Łukasz Osuszek

OPTYMALIZACJA KODU

Łukasz Osuszek

Optymalizacja kodu

www. e-naukowiec.eu

ISBN 978-83-936418-8-8

Lublin, 2013

Recenzja
Andrzej Radomski

Korekta
Grażyna M. Giersztyn

Skład i łamanie
Marta A. Kostrzewa

Projekt okładki
Radosław Bomba

ISBN 978-83-936418-8-8

Spis treści

Wprowadzenie....6

Zawartość książki....7

Rozdział I. Optymalizacja a inżynieria oprogramowania....8

Pomiar gęstości defektów....8

Unikanie błędów....9

Tolerancja błędów....11

Optymalizacja procesu developmentu i model CMM....12

Normy i standardy oprogramowania....14

Rozdział II. Przydatne narzędzia....16

Profilery....16

Narzędzia do śledzenia zajętości i tropienia wycieków pamięci....17

Rozdział III. Użyteczne techniki programistyczne....19

Proces optymalizacji....20

Główne założenie podczas optymalizowania programu....24

Wykorzystywanie zmiennych....24

Procedury lokalne....26

Strategie optymalizacyjne....27

Zmienne wskaźnikowe....29

Zbiory....30

Pętle....31

Konstrukcja Case....33

Typy danych....35

Optymalizacja działań matematycznych w Delphi....37

Operacje zmiennoprzecinkowe....38

Dobór właściwości komponentów....40

Blokowanie przerysowania....41

Rysowanie w oknach....41

Redukcja rozmiaru aplikacji....42

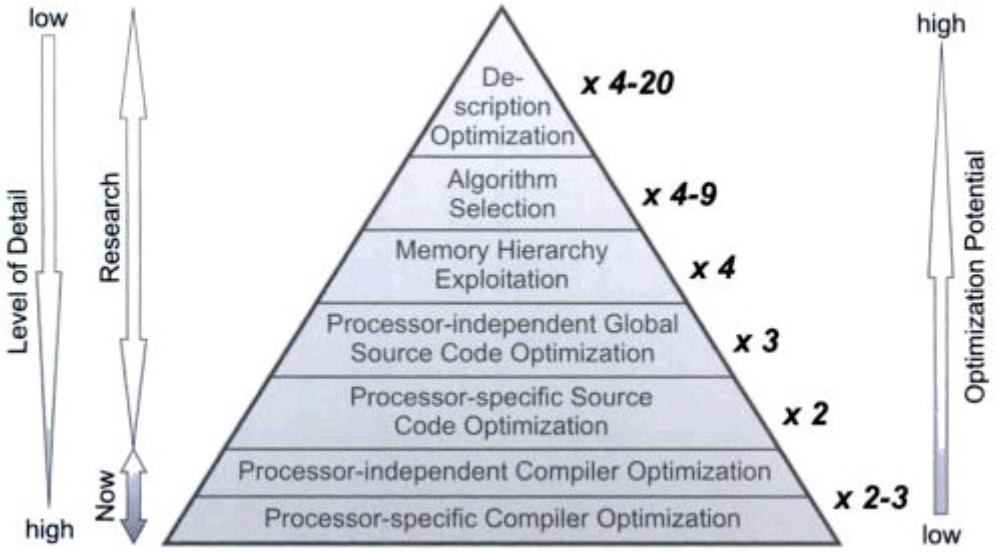
Optymalizacja kodu dla .NET....49

Praca ze zmiennymi typu string....49
Operator Null....51
Alokacja na stosie i stercie....52
Operator rzutowania AS....53
Stosowanie „using”....54
Dodawanie obiektów do kolekcji....55
Kopiowanie kolekcji do tablicy....56
Konwersja typów – najlepsze praktyki....57
Pętle....57
Rozdział IV. Podstawowe pojęcia związane z programowaniem wielowątkowym....59
Wprowadzenie....59
Obsługa wątków w środowisku Delphi....60
Wątek główny i wątek poboczny w środowisku Delphi....61
Niespodzianki oraz problemy związane z wątkami....63
Atomowość w stosunku do danych współdzielonych....65
Kończenie, przerywanie i niszczeniem wątków....68
Mechanizmy i obiekty wykorzystywane do synchronizacji....72
MREWS i zdarzenia....80
Zarządzanie priorytetami wątków....81
Rozdział V. Inicjatywa FastCode....84
Rozdział VI. Testowanie oprogramowania....85
Rodzaje testów....85
Testy modułowe (jednostkowe)....86
Testowanie z raportowaniem....87
Turbo Delphi Explorer i DUnit....88
Wykrywanie wycieków pamięci a DUnit....89
Uruchamianie testów DUnit z lini poleceń....90
Uruchamianie testów w trybie tekstowym....91
Obiekty imitacji (Mock Objects)....91
Testy akceptacyjne....93
Podsumowanie....94
Bibliografia....95

Wprowadzenie

W dzisiejszych czasach optymalizacja kodu jest jednym z wymogów tworzenia profesjonalnego oprogramowania. Skuteczna optymalizacja kodu jest sztuką. Obecnie zaobserwować można, że rozwój dziedziny optymalizacji postępuje najszybciej w obszarze usprawniania kodu źródłowego i algorytmów. Jeszcze w latach 90-tych więcej uwagi poświęcano optymalizacyjnym technikom kompilatorów. Niewątpliwą zaletą optymalizacji na poziomie kodu źródłowego jest generyczność i przenaszalność (w odróżnieniu od optymalizacji kompilatorowej, wykorzystującej możliwości konkretnej architektury sprzętowe). W książce tej przybliżono podejście optymalizacyjne na przykładzie kodu tworzonego w Delphi oraz C#. Jednak zawarte tu techniki mogą być z powodzeniem stosowane w innych środowiskach programowania. Kolejnym argumentem przemawiającym na korzyść optymalizacji na poziomie kodu jest łatwość sprawdzania poprawności działań polepszających kod. Algorytmy zapisane w języku maszynowym są znacznie trudniejsze do śledzenia i analizy niż kod stworzony w określonym środowisku.

Najbardziej podkreślaną zaletą optymalizacji kodu jest jej abstrakcyjność. Badania wskazują, że ogromny potencjał optymalizacyjny przesunięty jest w kierunku warstw abstrakcyjnych. Poniższy rysunek prezentuje obszary prac i rozwój dziedziny optymalizacji, jak również informacje o potencjale danej warstwy:



Rys. 1 Rozwój dziedziny optymalizacji. Falk Heiko

Dzisiejsze kompilatory nie operują na warstwie abstrakcyjnej. Nie rozumieją on informacji o całej strukturze danego programu, nie są w stanie użyć całego potencjału optymalizacyjnego. Stąd bardzo ważne jest podejście optymalizacyjne do kodu źródłowego. Zadaniem tej książki jest przybliżyć czytelnikowi najważniejsze techniki związane z optymalizacją kodu źródłowego.

Przedstawione dalej techniki są abstrakcyjne, niezależne od architektury sprzętu, na którym będą uruchamiane. Książka zawiera praktyczne przykłady zastosowania metod oraz technik optymalizacyjnych.

Zawartość książki

Rodział I – zawiera ważne z punktu widzenia inżynierii oprogramowania pojęcia wiążące się z optymalizacją procesu tworzenia oprogramowania. Pokazano w nim najnowsze trendy i osiągnięcia z dziedziny inżynierii oprogramowania wiążące się z procesem optymalizacyjnym.

Rodział II – przedstawia informacje o podstawowych narzędziach wykorzystywanych w procesie optymalizowania kodu źródłowego.

Rodział III – opisuje najpopularniejsze techniki programistyczne, używane w procesie optymalizacji.

Rozdział IV – jest kompendium wiedzy na temat na temat tworzenia aplikacji wielowątkowych. Zastosowanie mechanizmu wielowątkowości jest obecnie jednym z najpopularniejszych technik optymalizacji.

Rozdział V – zawiera informacje o inicjatywie FastCode związanej z optymalizacją kodu.

Rozdział VI – prezentuje informacje dotyczące technik testowania oraz popularnych narzędzi do sprawdzania poprawności kodu.

Rozdział I. Optymalizacja a inżynieria oprogramowania

W niniejszym rozdziale przedstawiono główne koncepcje i paradygmaty reprezentujące ostatnie osiągnięcia z zakresu zapewnienia i pomiaru jakości oprogramowania. Podkreślono tu ważną rolę pomiaru w zapewnieniu jakości i optymalizacji oprogramowania. Wyjaśniono zakres działań związanych z pomiarami cech istotnych dla zapewniania jakości na różnych etapach cyklu życia oprogramowania. W rozdziale tym przybliżono, w jaki sposób niemal wszystkie zróżnicowane działania metodyczne wywodzą się z podstawowych celów zarządzania: polepszenia oceny nakładów na produkcję oprogramowania i oceny wydajności programistów. Z tych prostych celów wywodzą się w szczególności szeroko zakrojone prace nad modelami i metrykami jakości oprogramowania. Przyjęto założenie, że dokładne miary wydajności i nakładów muszą uwzględniać jakość otrzymywanych wyników, a nie tylko ich wielkość. Rozdział ten zawiera przegląd i opis schematów pomiarów oprogramowania obejmujących najczęściej przytaczane metryki. Schematy te oparte zostały na pewnych fundamentalnych spostrzeżeniach z teorii pomiarów i na prostej klasyfikacji rozróżniającej procesy, produkty i zasoby.

Faza implementacji ma ogromny wpływ na niezawodność oraz wydajność i jakość oprogramowania. Często w trakcie tworzenia oprogramowania pojawia się problem znalezienia kompromisu pomiędzy efektywnością i niezawodnością oprogramowania. Problemy związane z niewystarczającą efektywnością są łatwo zauważalne. Efektywność jest także stosunkowo łatwa do poprawy poprzez np. optymalizację kodu kluczowych modułów systemu.

Natomiast zwiększenie niezawodności na etapie implementacji można osiągnąć dzięki detekcji oraz tolerancji błędów.

Pomiar gęstości defektów

Najczęściej stosowanym sposobem pomiaru jakości kodu, oznaczonego C, jest metryka gęstości defektów, definiowana jako:

```
(ilość defektów odkrytych w C) / (rozmiar C)
```

Rozrmar C jest zwykle określany w KLOC (tysiącach wierszy kodu). Choć gęstość defektów może być wskaźnikiem jakości, jeśli będzie stosowana konsekwentnie, nie jest rzeczywistą jakością oprogramowania. Metryka ta wiąże się z problemami, a mianowicie: nie jest wcale jasne, czy wyraża jakość programu, czy raczej rzetelność testowania tego programu jak również brak zgody w sprawie tego, co rozumiemy przez „defekt”. Ogólnie defektem może być albo usterka odkryta podczas przeglądu i testowania (która potencjalnie może doprowadzić do awarii w trakcie wykonania programu) albo awaria zaobserwowana podczas

działania tego programu. W niektórych opracowaniach defekty oznaczają po prostu awarie stwierdzone po przekazaniu oprogramowania do użytkowania, w innych oznaczają wszystkie znane usterki, w jeszcze innych zbiór usterek odkrytych po arbitralnie ustalonym punkcie w cyklu życia oprogramowania (np. po testowaniu możliwości). Różne organizacje stosują w tym zakresie bardzo zróżnicowaną terminologię: „intensywność usterek", „gęstość defektów" i „awaryjność" używane są niemal zamiennie. Gęstość defektów stała się *de facto* standardową miarą jakości oprogramowania w przemyśle informatycznym.

Z oczywistych powodów firmy bardzo niechętnie publikują dane o gęstości defektów w ich produktach, nawet gdy jest ona względnie niska. Nieliczne dostępne wzmianki na ten temat są formułowane w sposób uniemożliwiający identyfikację źródła, a ich forma uniemożliwia niezależne potwierdzenie wiarygodności tych danych. Niemniej jednak przedstawiciele firm często cytują odpowiednie liczby na konferencjach i w nieoficjalnych doniesieniach. Niezależnie od kłopotów ze stwierdzeniem zarówno prawdziwości tych liczb, jak i tego, co i w jaki sposób mierzono, istnieje zgoda w sprawie następujących ustaleń: w USA i w Europie średnia gęstość defektów (według liczby znanych defektów po przekazaniu oprogramowania do użytkowania) mieści się między 5 i 10 na KLOC. Liczby japońskie wydają się znacząco niższe (zwykle poniżej 4 na KLOC), jednak może to być wynikiem tego, że raporty pochodzą tylko z czołowych firm. Uważa się powszechnie, że gęstość defektów (w dostarczonym do użytkowania programie) poniżej 2 na KLOC to dobry wynik.

Do chwili obecnej jednym z bardziej wyśrubowanych miar jakości jest „cel jakościowy sześć sigma" firmy Motorola – polega na osiągnięciu „nie więcej niż 3,4 defektu na milion podstawowych jednostek wyjściowych z projektu". Przekłada się to na wyjątkowo niską gęstość defektów wynoszącą 0,0034 na KLOC. Oczywiście nawet niedościgły poziom zerowej gęstości defektów nie musi wcale oznaczać, że osiągnięta została bardzo wysoka jakość i kod wynikowy jest optymalny.

Unikanie błędów

Co zatem zrobić, aby tworzony przez nasz kod był jak najlepszy? Pełne uniknięcie błędów na etapie tworzenia kodu nie jest oczywiście możliwe. Pewne podejścia powalają jednak uniknąć wielu błędów. Podejścia takie to:

- Unikanie niebezpiecznych technik programistycznych.
- Zasada ograniczonego dostępu.
- Stosowanie kompilatorów sprawdzających zgodność typów.

Pewne techniki programistyczne zwiększają możliwość popełnienia błędów. Są to:

- Instrukcja ***goto*** (zaburza naturalną strukturę algorytmu).
- Obliczenia równoległe. Stosowanie wielowątkowości prowadzi do złożonych zależności czasowych, które mocno utrudniają uruchamianie oprogramowania. Wiele błędów nie ujawnia się w momencie śledzenia programu. Możliwe jest także równoczesne operowanie na tych samych danych przez różne procesy (więcej o tym zagadnieniu w kolejnych rozdziałach).

- Liczby zmiennopozycyjne. Obliczenia z wykorzystaniem liczb zmiennopozycyjnych wykonywane są ze skończoną dokładnością. Błędy zaokrągleń pojedynczych operacji mogą się kumulować, prowadząc do bardzo poważnych błędów obliczeń. Wiele błędów wiąże się także z porównywaniem liczb zmiennopozycyjnych. Dwa wyrażenia, które z matematycznego punktu widzenia dają dokładnie ten sam wynik, mogą dać nieznacznie różniące się rezultaty w przypadku realizacji komputerowej.
- Wskaźniki. Posługiwanie się wskaźnikami prowadzi do wielu błędów dostępu do pamięci. Wskaźniki mogą prowadzić także do sytuacji, w których te same zmienne mają różne etykiety.
- Rekursja. Stosowanie tej techniki utrudnia śledzenie programu. Często jest także przyczyną zapętlania się programu.

Niektóre z powyższych technik mogą się w pewnych sytuacjach okazać wskazane lub nawet niezbędne. Wielu błędów można także uniknąć dzięki stosowaniu zasady *ograniczonego dostępu*. Zasada ta znana jest również jako *need-to-know*. Jest to zasada stosowana w armii oraz innych organizacjach dbających o bezpieczeństwo mówi, że wiedzieć (czyli mieć dostęp do informacji) powinny tylko te osoby, które muszą wiedzieć. Wszystkie informacje są z założenia utajnione i prezentowane, kiedy zachodzi wyraźna potrzeba udostępnienia ich upoważnionemu. Zasada ta przeniesiona na programowania mówi, że dostęp do poszczególnych elementów programu powinny mieć tylko te składowe, które rzeczywiście odwołują się do określonych elementów np.: konkretna zmienna powinna być dostępna wyłącznie dla metod, które odczytują lub ustawiają jej wartość. Należy w miarę możliwości stosować wszędzie techniki ograniczania dostępu do składowych programu poprzez np. stosowanie funkcji prywatnych oraz stosowanie prywatnych pól i metod.

Często przyczyną błędów jest niepoprawne deklarowanie typów zmiennych parametrów funkcji i omyłkowe wykonywanie operacji na wyrażeniach o różnych typach. Pomyłek tych można uniknąć, jeżeli kompilator w ścisły sposób sprawdza zgodność i nie pozwala programiście na wykonanie błędnej operacji chyba, że konwersja zostanie jawnie wymuszona.

Wielu programistów ma niestety tendencję do preferowania języków, które pozwalają na automatyczną konwersję typu. Język C++ jak i wykorzystywany w Delphi Pascal jest natomiast językiem o ścisłej weryfikacji zgodności typów.

W Delphi, podobnie jak w innych językach obiektowych, klasa bazowa ma typ zgodny z klasami pochodnymi. Jest to oczywiście niezbędne z punktu widzenia pewnych mechanizmów programowania obiektowego, w pewnych sytuacjach może być jednak także przyczyną błędów.

Wiele kompilatorów zgodnie ze standardem danego języka programowania nie uznaje za błędy pewnych potencjalnie niebezpiecznych konstrukcji, między innymi naruszeń zgodności typów. Kompilatory te pozwalają jednak na generowanie ostrzeżeń w przypadku wykrycia takich konstrukcji. Z opcji tych należy korzystać w jak najszerszym zakresie. Wygenerowanych ostrzeżeń nie należy ignorować nawet, jeżeli ich pojawienie się nie przerywa kompilacji programu.

Tolerancja błędów

Jest oczywiste, że żadna z powyższych technik unikania błędów nie gwarantuje uzyskania programu w pełni bezbłędnego. Tolerancja błędów oznacza, że program działa poprawnie, a przynajmniej sensownie także wtedy, gdy zawiera błędy. Tolerancja błędów wymaga wykonywania przez program następujących zadań:

- wykrycia błędu
- wyjścia z błędu, czyli zakończenia pracy modułu, w którym wystąpił błąd w poprawny sposób
- ewentualnej naprawy błędu, to jest zmiany programu tak, aby zlikwidować wykryty błąd

Istnieją dwa główne sposoby automatycznego wykrywania błędów:

- sprawdzanie warunków poprawności danych
- porównywanie wyników różnych wersji modułu

Najczęściej na etapie analizy i projektowania określa się między innymi ograniczenia, jakie muszą spełniać poprawne dane oraz warunki poprawnego rozpoczęcia i zakończenia metod i procesów. Naruszenie tych warunków oznacza wystąpienie błędu. Poprawność tych warunków może być sprawdzana przez dodatkowe fragmenty kodu.

Błędy można też wykryć wykorzystując kilka modułów wykonujących te same operacje. W elektronice znana jest zasada *TMR* (ang. *triple-modular redundancy*), stosowana w przypadku układów o szczególnie wysokich wymaganiach wobec niezawodności, np. w zastosowaniach kosmicznych. Zgodnie z tą zasadą wykorzystuje się trzy kopie (lub inną nieparzystą liczbę kopii) układu, którego poprawne działanie jest szczególnie istotne. Wyjścia tych trzech kopii są porównywane. Jeżeli nastąpi awaria jednej z nich, jest bardzo mało prawdopodobne, aby pojawiła się ona jednocześnie w jednej z pozostałych kopii. Dlatego też wynik, który różni się od dwóch pozostałych jest uznawany za błędny. Oprogramowanie nie ulega awariom podobnym do awarii sprzętowych, tj. nie pojawiają się nim nowe błędy. Problemem są wyłącznie błędy istniejące w oprogramowaniu od momentu jego powstania. W związku z tym, powielanie tego samego modułu w kilku kopiach nie miałoby oczywiście sensu. Poszczególne wersje danego modułu powinny więc zostać zaimplementowane przez niezależne zespoły programistów. Stosowanie wielu wersji modułów narzuca oczywisty sposób wyjścia z błędu poprzez wybór jego poprawnych wyników generowanych przez większość wersji. Technikę taką nazywa się *programowaniem N-wersyjnym*.

Wykonanie kilku wersji modułu wiąże się jednak z dodatkowym narzutem czasu i zmniejsza wydajność systemu. Techniką pozbawioną tej wady jest stosowanie zapasowych modułów. W tym wypadku w danym momencie pracuje tylko jedna wersja modułu. Jej praca jest jednak monitorowana. W przypadku wykrycia niepoprawnych wyników aktywowany jest moduł zapasowy. Wyjście z błędu polega w tym wypadku na ponownym wykonaniu tych samych operacji przez zapasową wersję modułu. Naprawa błędu polega na odłączeniu wersji, w której wykryto błąd i zastąpieniu jej inną wersją.

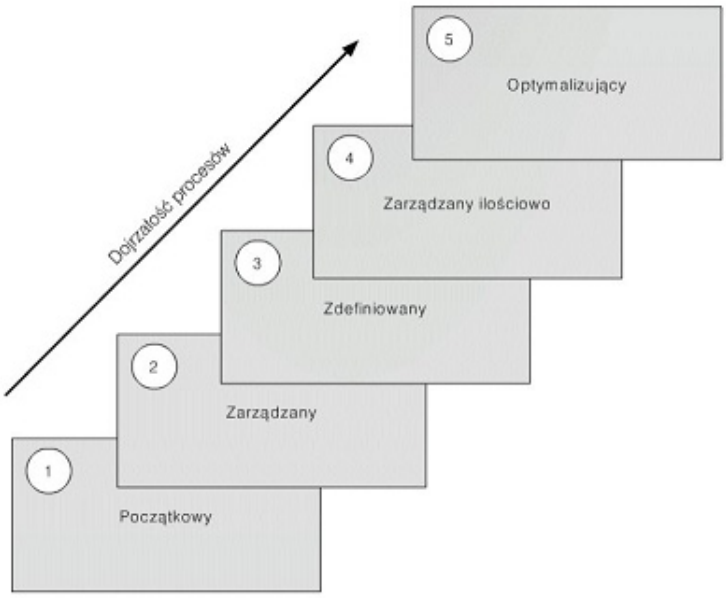
Jeżeli nie wykorzystuje się wielu wersji modułów, podstawowym sposobem wyjścia z błędu jest powrót do poprzedniego poprawnego stanu. Większość systemów baz danych lub transakcyjnych BPM zapew-

nia akceptowanie wyników transakcji tylko wtedy, gdy wszystkie operacje zakończą się sukcesem. Powrót do poprzedniego stanu jest więc wykonywany automatycznie przez system zarządzania bazą danych. W innych środowiskach niezbędne może być przechowanie dodatkowych informacji, które umożliwią powrót do poprzedniego stanu.

Optymalizacja procesu developmentu i model CMM

Z punktu widzenia inżynierii oprogramowania istnieje kilka sposobów na poprawienie jakości i optymalizację tworzonego oprogramowania. Do najbardziej popularnych należy zaliczyć SPI.

Poprawa procesów programowych (ang. *Software Process Improvement* – SPI) jest ogólnym terminem określającym ruch, którego podstawą jest przekonanie, że wszystkie zagadnienia jakości oprogramowania koncentrują się wokół ulepszenia procesu wytwarzania i eksploatacji oprogramowania. Centralnym elementem tego ruchu jest praca Instytutu Inżynierii Oprogramowania (ang. *Software Engineering Institute* w *Carnegie Melion University* w USA, promujący *model dojrzałości procesu wytwarzania* (ang. *Capability Maturity Model* – CMM). Opracowanie CMM zostało zlecone przez Departament Obrony Stanów Zjednoczonych w związku z problemami napotkanymi przy zakupach oprogramowania. Celem było uzyskanie metody oceny przydatności potencjalnych wykonawców. CMM jest pięciostopniowym modelem dojrzałości procesu organizacji wytwarzającej oprogramowanie (opartym w znacznym stopniu na koncepcjach *Total Quality Management* – TQM). Strukturę modelu ilustruje rysunek:



Rys. 2 Reprezentacja stała modelu CMMI. M. Chrapko

Dzięki rozbudowanemu kwestionariuszowi, uzupełniającym wywiadom i gromadzeniu różnych materiałów dowodowych, organizacje wytwarzające oprogramowanie można „przydzielić" do jednego z pięciu poziomów dojrzałości, opartych głównie na stopniu zdyscyplinowania ich procesów wytwórczych. Każdy poziom, za wyjątkiem pierwszego, charakteryzowany jest zestawem *kluczowych obszarów procesu* (ang. *Key Process Areas* – KPA).

Na przykład dla poziomu 2 wyróżniono następujące KPA: zarządzanie wymaganiami, planowanie

projektu, śledzenie projektu, zarządzanie podzleceniami, zapewnienie jakości i zarządzanie konfiguracją. KPA specyficzne dla poziomu 5 to zapobieganie defektom, zarządzanie zmianami technologii zarządzanie zmianami procesu.

Firmy powinny osiągnąć przynajmniej poziom 3, aby móc ubiegać się o kontrakty z Departamentu Obrony Stanów Zjednoczonych. Ta istotna motywacja komercyjna była pierwotnym powodem, dla którego CMM uzyskał taki rozgłos. W praktyce okazało się, że niewiele firm zdołało osiągnąć poziom 3; większość znajduje się wciąż na poziomie 1. Dopiero całkiem niedawno pojawiły się organizacje klasyfikowane na poziomie 5 – najbardziej znaną z nich jest część IBM odpowiedzialna za oprogramowanie dla programu promu kosmicznego NASA.

CMM wywiera wielki wpływ na całym świecie i wpływ ten spowodował znaczny wzrost zrozumienia i wykorzystania metryk oprogramowania. Powodem tego jest fakt, że metryki są istotne w odniesieniu do wszystkich KPA w całym modelu. Poniższa tabela przedstawia przegląd typów pomiarów sugerowanych dla każdego poziomu dojrzałości. Dobór pomiarów zależy od ilości informacji widocznej i dostępnej na każdym z poziomów. Pomiary na poziomie 1 zapewniają możliwość porównań dla prób poprawy procesów i produktów. Pomiary na poziomie
2 skupiają się na zarządzaniu projektem, podczas gdy na poziomie 3 dotyczą produktów pośrednich i finalnych uzyskanych w trakcie wytwarzania oprogramowania. Pomiary na poziomie 4 dotyczą charakterystyki samego procesu wytwarzania, w celu umożliwienia kontroli poszczególnych działań związanych z procesem. Proces na poziomie 5 jest na tyle dojrzały i tak starannie zarządzany, że umożliwia prowadzenie pomiarów w celu uzyskania informacji koniecznych do dynamicznej zmiany procesu w trakcie realizacji konkretnego projektu.

5 . Optymalizujący	Ulepszanie procesu	Proces plus informacje zwrotne w celu zmiany procesu
4. Zarządzany	Proces mieszany	Proces plus informacje zwrotne dla sterowania procesem
3. Zdefiniowany	Proces zdefiniowany i zinstytucjonalizowany	Produkt
2. Powtarzalny	Proces zależny od Indywidualności wykonawców	Zarządzanie projektem
1. Początkowy	Proces chaotyczny	Podstawowa

Tabela 1 Pomiar procesu na różnych poziomach dojrzałości. M. Chrapko

Mimo międzynarodowej akceptacji nie brak krytyków CMM. Najpoważniejszy zarzut dotyczy prawidłowości samej pięciostopniowej skali. Jak dotąd brak w pełni przekonujących dowodów na to, że firmy zakwalifikowane na wyższy poziom wytwarzają lepsze oprogramowanie. Istnieją także zastrzeżenia dotyczące metod oceny według modelu CMM. Europejskim projektem (finansowanym w ramach programu ESPRIT) blisko spokrewnionym z CMM: jest Bootstrap. Metoda Bootstrap także stanowi model do oceny dojrzałości procesu przygotowania oprogramowania. Podstawowa różnica polega na tym, że umożliwia on ocenę poszczególnych projektów (a nie całych organizacji), a wynikiem oceny jest liczba rzeczywista między 1 a 5.1 tak, na przykład, firma może zostać oceniona na 2.6, co oznacza, że jej dojrzałość jest „lepsza" niż poziom 2 (CMM), jednak nie dość dobra w świetle wymagań poziomu 3 CMM.

Najnowszym osiągnięciem na polu ulepszania procesu jest *SPICE* (ang. *Software Process Improvement and Capability Determination*). Jest to projekt międzynarodowy, którego celem jest opracowanie standardu oceny procesu wytwarzania oprogramowania, korzystając z najlepszych cech modeli CMM, Bootstrap i JSO9003.

Normy i standardy oprogramowania

Istnieje obecnie wiele norm krajowych i międzynarodowych pośrednio lub bezpośrednio związanych z zapewnieniem jakości oprogramowania. Ogólna krytyka tych norm wiąże się z faktem, że są one zbyt subiektywne i koncentrują się niemal wyłącznie na procesie wytwarzania, a nie na produkcie. Mimo tej krytyki, poniższe ogólne normy dotyczące zapewniania jakości oprogramowania wywierają znaczny wpływ na działalność związaną z metrykami oprogramowania.

Normy serii ISO 9000 i TicklT

W Europie, a także w coraz większym stopniu w Japonii, najważniejszy standard jakościowy, do którego aspirują firmy, opiera się na normie międzynarodowej ISO 9001. Ta ogólna norma produkcyjna określa zestaw dwudziestu wymogów dla systemu zarządzania jakością, obejmujących politykę, organizację, zakresy odpowiedzialności i przeglądy, a także działania kontrolne, które trzeba stosować w trakcie całego cyklu życia, aby otrzymać produkt wysokiej jakości. ISO 9001 nie dotyczy żadnego konkretnego sektora rynku. Jej wersją dotyczącą oprogramowania jest norma ISO 9003. Norma ISO 9003 jest także podstawą inicjatywy TickIT, sponsorowanej przez brytyjski Departament Handlu i Przemyślu (ang. *Department of Trade and Industry* – DTI). Firmy występują o przyznanie świadectwa TicklT (większość głównych firm w zakresie technologii informatycznych uzyskało już takie świadectwa). Co trzy lata podlegają one pełnej ponownej kontroli.

Poszczególne kraje posiadają własne normy oparte na serii ISO 9000. Na przykład w Wielkiej Brytanii takim odpowiednikiem jest seria BS 5750, a odpowiednikiem normy ISO 9001 w Unii Europejskiej jest EN 29001.

ISO 9126

Ocena produktów programowych – charakterystyki jakościowe i wytyczne ich stosowania

Jest to pierwsza norma międzynarodowa stanowiąca próbę zdefiniowania ram dla oceny jakości oprogramowania. Norma definiuje jakość oprogramowania jako:

„*Całość cech i właściwości produktu programowego wpływająca na jego zdolność do zaspokojenia określonych lub implikowanych potrzeb*".

ISO 9216 stwierdza, że jakość oprogramowania można ocenić przy pomocy sześciu cech: funkcjonalności, niezawodności, efektywności, używalności, pielęgnowalności i przenośności. W książce tej położono podstawowy nacisk na efektywność i optymalizację oprogramowania. Każda z nich jest dalej definiowa-

na jako zestaw atrybutów mających wpływ na odnośny aspekt oprogramowania; każda może być również uszczegółowiona za pomocą licznych poziomów cech niższego rzędu. Tak, na przykład, niezawodność jest definiowana jako:

„*zestaw atrybutów mających wpływ na zdolność oprogramowania do utrzymania właściwego mu poziomu sprawności w określonych warunkach przez określony okres czasu*";

natomiast przenośność jest definiowana jako:

„*zestaw atrybutów mających wpływ na zdolność oprogramowania tego by mogło być przenoszone z jednego środowiska do innego*".

Niektóre osoby dowodzą, że skoro cechy nadrzędne i cechy podrzędne nie zostały właściwie zdefiniowane, ISO 9126 nie zapewnia ram pojęciowych, w obrębie których strony posiadające różne punkty widzenia na jakość oprogramowania, np. użytkownicy, sprzedawcy, organizacje nadzorujące, mogą dokonywać porównywalnych pomiarów. Definicje atrybutów takich jak niezawodność, różnią się także od definicji spotykanych w innych uznanych normach. Niemniej jednak, ISO 9126 stanowi ważny etap na drodze rozwoju pomiarów jakości oprogramowania.

Rozdział II. Przydatne narzędzia

Profilery

Jednym z bardziej przydatnych narzędzi podczas operacji optymalizowania kodu jest profiler. Profilowanie jest formą dynamicznej analizy działania programu. Jest to sposób analizy zachowania oprogramowania w oparciu o dane zebrane w czasie wykonywania badanego obiektu (programu, lub jakiejś jego części np. funkcji etc.). Zwykle celem tej analizy jest określenie, które części programu można optymalizować – zwiększyć ogólną szybkość, zmniejszyć zapotrzebowanie na pamięć, a czasem jedno i drugie.

Najczęściej profiler bada jedynie – częstotliwość wywoływania funkcji oraz czas trwania danej partii kodu.

Częstym błędem początkujących developerów jest tworzenie programów do testowania wydajności, do analizy czasu trwania funkcji, czy liczby wywołań określonego kodu. Czyli tak naprawdę powielanie istniejących już rozwiązań.

Każdy większy projekt trzeba profilować. Poniższy cytat całkiem nieźle do tego przekonuje i co ważniejsze nie kłamie: „*Zwykle mniej niż 4% programu zużywa więcej niż połowę czasu jego wykonania.*”

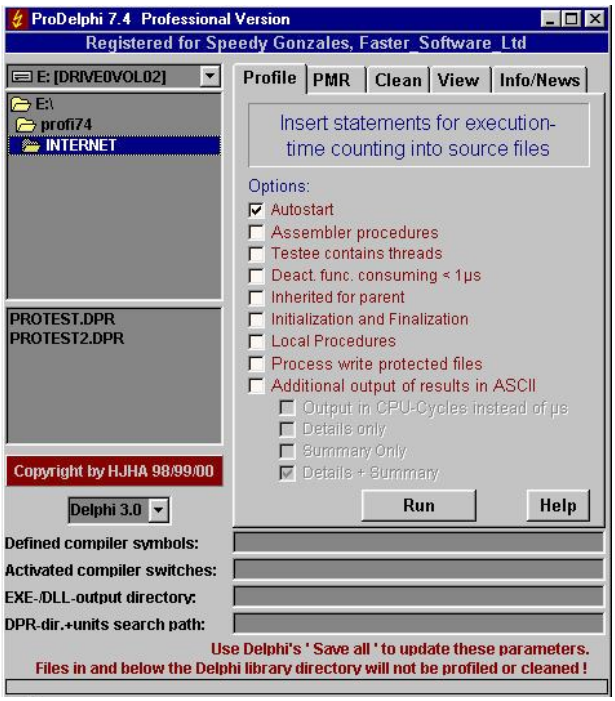
Profilowanie płaskie. Profil płaski służy do mierzenia czasu, każdej wykonanej funkcji programu, na tle całości. Nieużywane funkcje, pomimo iż zawarte w programie nie będą brane pod uwagę.

Graf wywołań. Graf wywołań ułatwia doszukanie się funkcji, które same w sobie nie są czasochłonne, ale wywołują inne (funkcje-dzieci) zjadające niezwykłe ilości czasu.

GpProfile	http://code.google.com/p/gpprofile2011/
MemProof	http://www.torry.net/pages.php?id=1526
ProDelphi	http://www.prodelphi.de/
Aqtime	http://smartbear.com/products/development-tools/performance-profiling/

Tabela 2 Zestawienie profilerów

Zdecydowanie najpopularniejszym narzędziem do profilowania kodu stworzonego w Delphi jest ProDelphi. W sposób najmniej inwazyjny i niezaburzający pracy oprogramowania (w tym koegzystencji z systemem operacyjnym) umożliwia śledzenie pracy programu.



Rys. 3 Ekran aplikacji ProDelphi

Dodatkowymi zaletami procesu profilowania są wykrywanie błędów, znajdywanie nazbyt „gorących miejsc" programu i łatwiejsza analiza oprogramowania o zbyt obszernym kodzie źródłowym.

Narzędzia do śledzenia zajętości i tropienia wycieków pamięci

Oprócz przemyślanego tworzenia kodu, wyciekom pamięci można zapobiegać używając aplikacji firm trzecich lub narzędzi zawartych w samym Delphi.

Wszystkie wersje Delphi od 2006 wzwyż posiadają ulepszony menadżer pamięci. Jednym z dodatków jest możliwość zarejestrowania i wyrejestrowania oczekiwanych wycieków, oraz opcjonalnie raportowanie nieoczekiwanych wycieków po zamknięciu programu.

Można również posiłkować się zewnętrznymi narzędziami:

EurekaLog	http://eurekalog.com/index_delphi.php
MemProof	http://www.scip.be/index.php?Page=ArticlesDelphi07&Lang=EN
FastMM	http://sourceforge.net/projects/fastmm/

Tabela 3 Zestawienie narzędzi do śledzenia zajętości pamięci

Wykorzystanie ostatniego z nich (FastMM) jest stosunkowo proste. Polega na dopisaniu jednej linijki do **dpr**:

```
program ProgramDoZbadania;
[...]
uses
```

```
  FastMM4,
  Forms,
```

Dodatkowo definiujemy, czy chcemy używać biblioteki **FullDebugMode** DLL

Przykładowe **Conditional defines**:

```
DEBUG;
FullDebugMode;
LogErrorsToFile;
LogMemoryLeakDetailToFile;
```

Raport o wyciekach pamięci zapisany zostanie w pliku tekstowym, w katalogu uruchamiania aplikacji. Poniżej przykładowy log:

```
--------------------------------2011/10/4 11:15:23---------------------------
A memory block has been leaked. The size is: 88

Stack trace of when this block was allocated (return addresses):
 402A9B [system.pas][System][@GetMem][2447]
 6098CE [Forms\Main.pas][Main][TForm1.SpeedRecClick][4666]
(...)
```

Przedstawiony przykład wskazuje na to, że program pobrał pamięć, ale bezpośrednio jej nie zwolnił:

```
6098CE [Forms\Main.pas][Main][TForm1.SpeedRecClick][4666]
```

Analiza modułu **Main.pas** pod wyszczególnioną linią (4666) wskazuje na winowajcę wycieku:

```
SpeedRec.Glyph:=TBitmap.Create();
```

Widać, że utworzony został obiekt **TBitmap**, który nie został potem zwolniony.

Rozdział III. Użyteczne techniki programistyczne

Osobiście wolę patrzeć na optymalizację kodu, jako na kompleksowy proces składający się z kilku faz. Im wcześniejsza faza tworzenia projektu podlega optymalizacji, tym lepsze wyniki są uzyskiwane. Nie należy jednak popadać w paranoję i optymalizować każdej linijki kodu. Preferowałbym tutaj raczej holistyczne podejście do tematu i zlokalizowania słabych punktów i tzw. „wąskich gardeł" w odniesieniu do całości projektu. Jeśli zlokalizujemy już problem nie należy od razu go poprawiać tylko przeanalizować następne kroki algorytmu i spojrzeć na problem szerzej. Często rozwiązanie problemu znaleźć można o poziom wyżej.

Jedno z najlepszych rozwiązań optymalizacyjnych w formie dowcipu opowiada Robert Lee, którego prace wniosły ogromny wkład w rozwój procesów optymalizacji kodu w Delphi:

Pacjent mówi do lekarza – Panie Doktorze, boli, kiedy poruszam ręką w ten sposób;
na co Doktor odpowiada – Więc proszę tak nie robić.

Doskonale oddaje on jedną z podstawowych zasad optymalizacji – czy naprawdę koniecznie musisz to wykonywać/z tego korzystać/to przetwarzać? To jest faza planowania optymalizacji w fazie algorytmicznej. Dopiero kolejnym krokiem jest faza implementacji, w której większość programistów dopiero zaczyna myśleć o optymalizacji. Algorytmy optymalizacyjne opisane w tej pracy bazują na algorytmie zastępującym – od wyższego poziomu abstrakcji do niższego bardziej szczegółowego.

Optymalizacja kodu w Delphi sprawia również, że staje się on bardziej przejrzysty i prostszy. Poza tym, stosując optymalizację algorytmów uzyskuje się spójność implementacji oraz ułatwienie zarządzania kodem.

1. Zlokalizowanie i zrozumienie problemu

Sama informacja o tym, że naszemu programowi pod względem szybkości działania bliżej do żółwia, niż zająca nie daje nam jeszcze dokładnej informacji, w którym miejscu zacząć optymalizację. Trzeba znaleźć winowajcę, czyli określić gdzie program zwalnia i dlaczego. W celu umiejscowienia wąskiego gardła najlepiej posłużyć się jednym z dostępnych na rynku programów profilujących (profilerów). Najbardziej popularne z nich to **TotalQA, GpProfile** czy dołaczany do pakietów Delphi Enterprise, na płycie z komponentami program ProDelphi. Zadaniem tych programów jest monitorowanie pracy aplikacji i zbieranie informacji na temat prędkości działania i wydajności poszczególnych składników programu. Możliwe jest śledzenie czasu wykonania poszczególnych procedur w podziale na milisekundy lub cykle procesora. Proces profilowania pozwala dokładnie znaleźć fragment kodu najbardziej zasobożerny, w celu jego późniejszej optymalizacji. Zdecydowanie zalecam korzystanie z tego typu narzędzi, ponieważ nie zawsze jesteśmy w stanie trafnie sami zlokalizować źródło problemu.

Doskonałą metodą do analizowania i profilowania tworzonego oprogramowania jest metoda *„dziel i zwyciężaj”* (ang. “*divide and conquer*”). Polega ona na podziale zadania (programu/algorytmu) na kilka mniejszych części, które dzieli się znowu i znowu, aż do rozebrania całego problemu na podstawowe podproblemy. Kiedy już do tego doprowadzimy możemy rozwiązać podproblemy, a następnie z powrotem poskładać je w jedną całość, aby otrzymać ostateczny wynik. Po zlokalizowaniu problemu przystępujemy do procesu optymalizacji:

2. Algorytm

Po pierwsze, należy dokładnie przeanalizować i prześledzić blokowe działanie algorytmu. Bez dokładnego „wczucia się” w istotę działania algorytmu dalsze prace nad optymalizacją wydają się mieć średni sens. Preferuje tutaj podejście holistyczne, pozwalające objąć całość algorytmu a nie jego poszczególne składniki.

Zachęcam do przeszukiwani zasobów sieciowych w celu odnajdywania rozwiązań lub algorytmów mogących mieć zastosowanie w rozwiązaniu problemów wydajnościowych. Doskonałym źródłem wiedzy są grupy dyskusyjne, przedstawiane tam algorytmy, pomimo że mogą przegrywać pod względem wydajności z naszymi rozwiązaniami zawsze dywersyfikują i otwierają oczy na inne rozwiązania. Nigdy nie należy odrzucać żadnej z możliwości i zawsze należy liczyć się ze zmianą koncepcji. Nie oznacza to również, że każdy znaleziony w sieci algorytm osiągający lepsza wydajność jest od razu lepszy od naszego. Algorytmy można porównywać dopiero wtedy, gdy oba są zoptymalizowane.

3. Kod

Kolejna fazą jest sprawdzenie implementacji algorytmu. W pierwszym podejściu powinniśmy stworzyć kod jak najbardziej uproszczony. To będzie nasza wersja „wyjściowa” kodu, nad którą będziemy praco-

wać. Jeszcze raz przyda się metoda „dziel i zwyciężaj” w przypadku bardziej skomplikowanych algorytmów, która podzieli je na mniej skomplikowane części. Proces ten nie tylko ułatwia nam zrozumienie implementacji i działania algorytmu. Ułatwia on również prace kompilatorowi. Często temu procesowi towarzyszy pierwszy wzrost wydajności aplikacji. Dla przykładu przeanalizujmy poniższe algorytmy:

```
  Procedure test1;
  for x:=0 to maxint shr 2 do begin
        z:=z+(ord(y<500)+1)*22;
        inc(y,ord(z mod 3=0)*2+1);
end;
end;
```

Jak się zaraz przekonamy optymalizacja kodu polegająca na zmniejszaniu ilości linijek i budowaniu skomplikowanych warunków logicznych prowadzi często do obniżenie wydajności algorytmu, a także do jego zaciemnienia. Po rozbiciu tego algorytmu na mniej skomplikowany uzyskujemy:

```
  Procedure test2;
  for x:=0 to maxint shr 2 do begin
    if y<500 then
      z:=z+44
    else
      z:=z+22;
    if z mod 3 = 0 then
      inc(y,3)
    else
      inc(y);
end;
end;
```

To przekłada się na wzrost czytelności i przejrzystości kodu, jak również na wzrost wydajności. Otóż, mamy tutaj wzrost wydajności o około 25%. Średni czas wykonania pierwszej pętli wynosił 18437 ms. Natomiast po zastosowaniu metody „dziel i zwyciężaj” czas ten zmniejszył się do 10859 ms. Więc o około siedem i pół sekundy! Któryś z niedowiarków może powiedzieć, że wyniki mierzone w ten sposób nie są obiektywne, ponieważ zależą od szybkości działania jednostki centralnej. No i rzeczywiście ziarnko prawdy w tym jest. Jest jednak pomiar, który bezdyskusyjnie wskazuje na słuszność tego rozwiązania – pomiar ilości cykli procesora przeznaczonych na wykonanie tego algorytmu. Jest to wielkość jednoznacznie określająca jak szybko dany algorytm zostanie wykonany. Niezależnie od tego na jakiej maszynie jest on wykonywany, cykl procesora to wielkość podstawowa. Prawda odnośnie mierzenia czasu wykonywania algorytmu jest taka, że określenie go w cyklach procesora ma wymiar uniwersalny; a przynajmniej na potrzeby przedstawionych tu badań możemy tak założyć (chociaż już Pentium potrafi dokonać zrównoleglenia 2 rozkazów, nowsze procesory są w tym lepsze, więc 1000 iteracji pętli złożonej z 10 instrukcji na 486 zostanie wykonane w 10000 cykli, a w Pentium teoretycznie może zostać wykonane w 10 + 999*5 cyklach. Na nowych procesorach może być jeszcze inaczej.) I tak na przykład, pomiary wymienionych wyżej procedur przedstawiają się następująco:

41.011.951.802 cykli procesora dla procedury pierwszej (wariant bez **if**)
oraz
23.252.333.042 cykli procesora dla procedury drugiej

Te wielkości na maszynie PIV z zegarem 3.06GHz przekładają się na czasy przedstawione wyżej. Ale jak pomiar czasu wykonywania algorytmów przedstawiony w cyklach procesora wskazuje na to, że jeżeli algorytmy uruchamiany byłby na szybszej jednostce centralnej stosunek wydajności drugiego algorytmu do pierwszego rósłby do prawie 50%?

Poza tym, rozpisany w ten sposób algorytm staje się dobrym punktem wyjścia do dalszego procesu optymalizacji. Często spotykanym rozwiązaniem stosowanym przez programistów jest zostawienie wersji kodu umieszczonej w komentarzach, jako wyjaśnienie działania algorytmu. Oczywiście zalecam zachowywanie wszystkich poszczególnych wersji algorytmu, ponieważ o dobrodziejstwie kopii bezpieczeństwa można by napisać wiele peanów.

4. Potrzebne narzędzia

Na pewnym etapie procesu optymalizacji niezbędne stają się narzędzia wspomagające ten proces. Pierwszą grupę z nich stanowią *code timery*. Jednym z najpopularniejszych narzędzi tego typu jest z_prof. Jest to małe, poręczne narzędzie napisane w Delphi umożliwiające pomiar czasu wykonywania algorytmów. Kolejna grupą narzędzi użytecznych w procesie optymalizacji są *testery* (ang. *test cases*). Zadaniem programów z tej grupy jest przeprowadzanie serii standardowych testów na aplikacji. Oczywiście, oprócz gotowych narzędzi możemy stosować własne odpowiedniki, mierzące czas czy wykonujące testy. Polecam to podejście, ponieważ często testy oferowane przez gotowe narzędzie, ze względu na swoją uniwersalność, nie są wystarczająco skuteczne. Równie użyteczną grupą narzędzi jest grupa *test bed* (dosł. *podłoże testowe*). Do narzędzi tego typu wprowadzamy nasze programy dzielone na mniejsze bloki funkcjonalno logiczne. Narzędzia sprawdzają program jakby był napisany w całości, po kolei testując wydzielone bloki algorytmu. Duża część procesu wytwarzania i testowania programu przebiega w środowisku emulatora, aby zminimalizować liczbę restartów komputera, a zmaksymalizować efektywność procesu tworzenia kodu.

Kolejnym, bardzo przydatnym narzędziem jest okno CPU. Nie oznacza to wcale, że od razu trzeba być ekspertem i 80 mnemoników assemblera znać na pamięć. Czasem wystarczy tylko umieć liczyć instrukcje.

Jednak zdecydowanie najpotężniejszymi i najczęściej używanymi „wspomagaczami" procesu optymalizacji są profilery. Na rynku istnieje ich bardzo wiele. Standardowo w pakiecie Delphi na płycie z partnerskim oprogramowaniem znajduje się **ProDelphi** – profiler napisany przez Helmutha J. H. Adolpha. Jest to bardzo przydatne narzędzie oferujące szeroki wachlarz funkcjonalności. Szerzej o narzędziach ułatwiających proces optymalizacji w dalszej części książki.

5. Ostatnie przygotowania

Nie pozostaje już nic innego niż, po podbudowie teoretycznej, wziąć się za praktyczniejsze aspekty optymalizacji. W dalszej części znajdują się bardziej praktyczne porady. Jest jednak kilka zasad sztuki optymalizacji, na które chciałbym zwrócić uwagę:

- nie zapominać o kopii bezpieczeństwa przed każdymi zmianami
- śmiało testować różne techniki
- nie zmieniać zbyt wiele w jednym podejściu
- nie spodziewać się gwałtownego wzrostu wydajności aplikacji po jednym zabiegu
- testować proponowane przez innych lub istniejące techniki/algorytmy ze swoimi
- pamiętać o tym, że czasem pogorszenie wydajności w jednym miejscu owocuje wzrostem wydajności całej aplikacji
- zawsze mieć otwarty umysł – im więcej rozwiązań wymyślisz, tym bardziej prawdopodobne jest, że osiągniesz lepsze rezultaty
- wykorzystywać każdą informację lub specjalistyczną wiedzę jaką posiadasz na temat problemu, nad którym pracujesz
- co pewien czas przyglądać się problemowi jako całości, żeby skupiać się nad właściwym problemem; czasem rozwiązanie problemu w jednym miejscu przesuwa problem w inne – trzeba co pewien czas monitorować całość aplikacji

Proces optymalizowania aplikacji jest procesem rekurencyjnym. Kiedy zakończyć proces optymalizacji? To dopiero ciężkie pytanie. W przypadkach, kiedy określony jest z góry poziom wydajności aplikacji jest to łatwe. Kiedy, powiedzmy, w wytycznych odnośnie projektu, pojawiła się informacja, że np. transport obrazu z serwera do klienta ma się odbywać w czasie krótszym niż 3 sek., po osiągnięciu tej wartości można zaprzestać procesu optymalizacji. Nie mniej jednak, większość zadań nie ma jasno sprecyzowanych wymagań i często podejście „im szybciej tym lepiej" powoduje zatracenie się programisty w niekończącym się procesie optymalizacji. Ogólnie rzecz biorąc, należy do problemu podejść raczej zdroworozsądkowo i według własnego uznania.

Generalnie optymalizację można podzielić na dwa typy: optymalizację kodu *sensu stricto* oraz techniki optymalizacyjne. Pisząc kod zawsze możemy rozwiązać dany zadanie na kilka sposobów. Niektóre z nich w istotny sposób wpływają na poprawę wydajności. Ten pasywny rodzaj optymalizacji zwany jest optymalizacją kodu *sensu stricto* (ang. *code style*). Aktywny sposób optymalizacji – usuwający określone wąskie gardła w aplikacji, sprowadza się do stosowania technik optymalizacyjnych. Poniżej zamieszam zbiór technik optymalizacyjnych.

Optymalizacja to nie tylko przyspieszanie pracy aplikacji. Często optymalizacja widziana jest jako poprawianie szybkości tworzenia kodu oraz jego debugowania. Oznacza to, ni mniej ni więcej, że nie oddajemy sobie przysługi tworząc szybki, ale nieczytelny i trudny do zrozumienia kod. Na szczęście tworzenie

optymalnego kodu w Delphi rzadko wymaga pisania nieczytelnego kodu, mało tego, często zoptymalizowany kod prezentuje się bardziej elegancko.

Główne założenie podczas optymalizowania programu

Najlepszym podejściem optymalizacyjnym jest podejście zstępujące. Żelazną zasadą optymalizacji jest: „*jeśli odpowiedź na to pytanie zajmuje zbyt dużo czasu, zmień pytanie*". Największą poprawę wydajności uzyskujemy zazwyczaj poprzez naniesienie zmian w części projektowej i algorytmicznej. Wraz ze schodzeniem coraz niżej, do coraz większego poziomu szczegółowości, musimy liczyć się z coraz mniejszymi możliwościami optymalizacji. Należy zapamiętać, że pierwszym krokiem w procesie optymalizacji jest spojrzenie na projekt jako całość i zstępowanie w dół, na kolejne poziomy szczegółowości.

Profilowanie kodu – czyli pomiar czasu jego wykonywania. Aby stwierdzić, że wykonaliśmy kawał dobrej roboty i algorytm wykonuje się o 30 sek. szybciej, należy przede wszystkim, zmierzyć początkowy czas wykonywania algorytmu. Każda kolejna modyfikacja kodu wiąże się z następnym pomiarem.

Bardzo dobrze w procesie optymalizowania sprawdza się używanie podglądu okna CPU. Oczywiście, analizowanie kodu mnemonicznego programu napisanego dla Windows nie jest niczym przyjemnym, ale jeszcze raz podkreślam, że nie trzeba być ekspertem w dziedzinie assemblera, żeby korzystać z tego narzędzia. W większości przypadków można sprawdzić różnice w efektywności rozwiązań, zliczając ilość wykonywanych instrukcji.

Wykorzystywanie zmiennych

Niezależnie od środowiska, w którym programujemy – pierwszą rzeczą, o której należy pamiętać jest fakt, że złożoność i komplikacja kodu utrudnia prace kompilatorowi. Naczelną zasadą jest tworzenie prostego kodu. Najlepiej, żeby w jednym algorytmie nie używać więcej niż 8 zmiennych. Nie należy w jednej pętli umieszczać zbyt dużo instrukcji. Fatalną praktyką jest deklarowanie zmiennych wewnątrz pętli, czyli tzw. *overloadowanie pętli*. Prowadzi ona do tego, że adresy zmiennych, indeksy tablic itd. w niej używane, są za każdą iteracją przeładowywane. Ogromne zyski optymalizacyjne płyną z rozdzielenia złożonych pętli na mniej złożone i przesuwaniu najbardziej wewnętrznych pętli do zewnętrznych bloków funkcjonalnych. Jeśli jest to zrobione dobrze, kod zyskuje również na czytelności.

Należy preferować lokalne zmienne nad zmienne o szerszym zasięgu. Lokalne zmienne to te przekazywane jako parametry, oraz deklarowane wewnątrz procedury/funkcji. Tylko i wyłącznie lokalne zmienne mogą być przekształcone na zmienne rejestrowe – a zmienna rejestrowa równa się szybkość! Czasem opłaca się najpierw skopiować globalne dane do lokalnych zmiennych jeszcze przed skorzystaniem z nich. Technikę ta najlepiej stosować wobec zmiennych pojawiających się w pętlach. Tego typu operacja zwiększa prędkość

kopiowania zmiennych, co z kolei przekłada się na zwiększenie wydajności.

Oto przykład dwóch procedur, z których jedna operuje na danych globalnych, a druga na zmiennych zadeklarowanych lokalnie:

```
var
  Form1: TForm1;
  x:integer;
  t:integer;
  i:integer;

implementation
[...]

procedure TForm1.global;
begin
  t:=gettickcount;
    for i:=0 to maxint do
      x:=x+1;
  caption:=inttostr(gettickcount-t);
end;

procedure TForm1.local;
var tt:integer;
    ii:integer;
    xx:integer;
begin
  tt:=gettickcount;
    for ii:=0 to maxint do
      xx:=xx+1;
  caption:=inttostr(gettickcount-tt);
end;
```

Ten prosty przykład bezapelacyjnie przedstawia wyższość stosowania zmiennych lokalnych nad zmiennymi globalnymi. Czas wykonywania procedury globalnej to około 4406 ms, natomiast czas wykonywania procedury **local** to około 1203 ms – co daje ponad 3,6-krotne przyspieszenie pracy algorytmu.

Jednak reguła preferowania zmiennych lokalnych nad globalne ma jeden wyjątek. Są to tablice, których elementami są typy proste. Jeżeli mamy tablice o stałym rozmiarze i stałych elementach, zadeklarowanie jej jako globalnej oszczędzi pracy rejestrom. Oszczędności te stają się dość znaczne, jeśli mamy do czynienia ze zdefiniowanymi stałymi strukturami.

Następna ważna reguła optymalizacyjna dotyczy parametrów przekazywanych do funkcji/ procedur. Często wykorzystywane w programie procedury nie powinny przyjmować więcej niż trzy parametry. Jest to magiczna liczba parametrów, jaka może być przekazywana przez rejestry. Stosując się do tej reguły idealnie utylizujemy rejestry procesora i ułatwiamy optymizerowi Delphi zadanie optymalizacji kodu. W tym miejscu należy przypomnieć, że w Delphi każda z metod lub procedur klasy lub obiektu posiada ukryty parametr **self**,

który jest każdorazowo przekazywany. Być może ta informacja zaskoczy część programistów Delphi, ale tak właśnie jest. Wystarczy przyjrzeć się fundamentom Delphi, a konkretnie modułowi **system**.**pas**. To właśnie w nim zdefiniowana jest podstawa każdej aplikacji w Delphi, czyli **TObjec**t, a śledząc dokładnie jego budowę odnajdziemy skrzętnie ukryty przed użytkownikami IDE parametr **self**. Zatem, w odniesieniu do często wykorzystywanych procedur lub funkcji klas bądź obiektów, najlepiej jest stosować dwa parametry.

Procedury lokalne

Nie jestem pewien gdzie po raz pierwszy zetknąłem się z pojęciem procedur lokalnych, czyli jednych procedur zagnieżdżonych wewnątrz kolejnych. Pamiętam jednak dokładnie, że autor zalecał to rozwiązanie jako korzystne z wielu strategicznych powodów. Otóż nie ma nic bardziej błędnego. Jeśli gdziekolwiek w algorytmie pojawia się tego typu „zjawisko" należy je jak najszybciej wyeliminować. Jeśli dodatkowo pojawia się ono w procedurze często wywoływanej w programie – możemy być pewni, że zlokalizowaliśmy jedno z potencjalnych „wąskich gardeł". Otóż główna zaleta procedur lokalnych to fakt, że zmienne zewnętrznej procedury widziane są przez wewnętrzną. Operacja ta wiąże się jednak ze skomplikowanymi operacjami na stosie. Tego typu konstrukcje znacznie obniżają wydajność aplikacji. Zamiast stosować tego typu „dziwolągi" proponowałbym z wewnętrznej procedury zrobić procedurę **unit scope**, a niezbędne parametry przekazać w postaci referencji. Na przykładzie przedstawionym poniżej możemy prześledzić, że rozwiązanie drugie, jest sprawniejsze od pierwszego o prawie 9 mln cykli procesora:

```
implementation
[...]

procedure zagwozdka;
var x:integer;

procedure inside;
      var i:integer;
      z:integer;
      begin
            for i:=0 to x do z:=0;
      end;

begin
  x:=maxint div 10;
  inside;
end;

procedure poprawionaWersja;
var x:integer;
begin
  x:=maxint div 10;
  inside(x);
end;
```

```
procedure inside(var x: integer);
var i:integer;
    z:integer;
begin
    for i:=0 to x do z:=0;
end;
```

Strategie optymalizacyjne

Jeśli chodzi o samo środowisko Delphi jest kilka technik, które zastosowane w programie zwiększają jego wydajność. Standardowe narzędzie optymalizacyjne w Delphi w specjalny sposób traktuje bloki kodu umieszczone między znacznikami **with**. Dlatego należy sięgać do tego mechanizmu. Należy jednak pamiętać, aby odwołanie w bloku **with** były jednoznaczne.

Jeśli chodzi o operację dodawania sprawa ma się podobnie. W niektórych wersjach Delphi implementacja polecenia **inc**() jest bardziej wydajna od zwykłego dodawania. I tak zwykłe dodawanie:

```
i := i + 4;
```

zamieniane jest na:

```
MOV     AX,[DS: i(0050)]
ADD     AX,0004
MOV     [DS: i(0050)],AX
```

i zajmuje 9 bajtów, natomiast operacja

```
inc(i,4);
```

przekłada się na:

```
ADD     [Word DS:+i(+0050)],+04
```

i zajmuje tylko 5 bajtów, co z kolei przekłada się na jej szybkość. To samo tyczy się operacji odejmowania i funkcji **Dec**.

Podobnie rzecz przedstawia się, jeśli idzie o operacje dzielenia i mnożenia. Operacje postaci:

`a = b *2` oraz `a = b div 2`

po zamianie odpowiednio na:

`a = b shl 1` oraz `a = b shr 1`

przełożą się na dość znaczący wzrost efektywności programu.

W przypadku deklarowania parametrów funkcji lub procedur należy dobrze przemyśleć funkcjonalność tworzonego kodu. Jeśli jest to możliwe, należy unikać parametrów przekazywanych jako wartość. Z uwagi na alokację pamięci dla tak przekazywanych parametrów radziłbym przekazywać taki parametr jako stałą. Korzystniejsze niż przekazywanie parametru jako wartości jest również przekazywanie parametru jako zmiennej.

Ważnym punktem optymalizacyjnym jest sprawdzenie listy używanych modułów. Teoretycznie linker jest inteligentnym narzędziem usuwającym dublujące się moduły, ale zapewniam, że manualnie jesteśmy w stanie zrobić to lepiej. Jeżeli w programie używamy danego modułu, używajmy go z głową. Jeśli korzystamy tylko z jednej funkcji danego modułu – korzystniej jest skopiować jej implementację do modułu użytkownika, który stworzyliśmy.

Duży wzrost wydajności możemy również osiągnąć modyfikując operacje związane z komponentami takimi jak **ListBox**, **Memo**, **treeView**, itd. Kiedy dodajemy lub modyfikujemy dużo pozycji szybkość działania komponentu drastycznie spada. Wynika to z faktu, że każde dodanie pozycji powoduje odświeżenie komponentu (jego przerysowanie) na ekranie. Jak łatwo policzyć dodanie 1000 pozycji do **ListBox** powoduje jego 1000 krotne przerysowanie na ekranie. To z kolei może potrwać aż do kilku minut. Jak widać, jeśli operujemy na kilku tysiącach pozycji tego typu działania wpływają znacznie na pogorszenie wydajności programu. Jednak twórcy graficznych komponentów w Delphi oferują nam magiczną metodę **BeginUpdate** dla każdego z wyżej wymienionych komponentów. Należy wywołać ją przed rozpoczęciem modyfikowania pozycji, a po ich zakończeniu należy wywołać **EndUpdate** w celu prezentacji zmian na ekranie. Wzrost wydajności rozwiązań z zastosowaniem tych funkcji jest znaczący i jednoznacznie wskazuje na opłacalność stosowania tego mechanizmu.

Jeśli używamy klasy **Tstrings** jako listy na której wykonujemy szereg operacji związanych z manipulacjami na łańcuchach – lepiej skorzystać z klasy **TstringHash** zaimplementowanej w Delphi w module **IniFiles**. Jest to według opisu klasa wykorzystywana wewnętrznie do optymalizacji wystukiwania w plikach ***.ini**. Jednak bez przeszkód możemy ją wykorzystywać do własnych algorytmów. W przypadku, kiedy operujemy na liście łańcuchów większej niż 100 pozycji – bardziej opłacalnym wariantem staje się skorzystanie z haszowanej listy łańcuchów. Wyszukiwanie ciągów znaków nie jest w niej zależne od ilości elementów!

W wersjach wcześniejszych niż D6 znany był bug, który odpowiedzialny był za bezmyślne pożeranie zasobów GDI. Problem spowodowany był błędem tkwiącym w implementacji wszystkich komponentów bazujących na klasie **TbuttonGlyph** (**TbitBtn**, **TspeedButton** itd.). Jeżeli ktokolwiek używa jeszcze Delphi w wersjach wcześniejszych niż D6 polecam mu modyfikację modułu **buttons**.**pas**. Więcej szczegółów na temat poprawy wydajności wspomnianych komponentów we wcześniejszych wersjach Delphi można znaleźć na portalu producenta.

Poważnym błędem programistycznym jest bezmyślne korzystanie z mechanizmu **forms-autocreation**. Czasem powoduje to przy starcie aplikacji ogromne zużycie zasobów systemowych oraz spowolnienie pracy. Należy tak dobrać tworzenie formatek aplikacji, aby przy starcie tworzone były jedynie niezbędne formy. Pozostałe należy załadować dynamicznie w aplikacji.

Ze względu na załadowanie pamięci korzystniej jest używać kontrolek „rysowanych" – to znaczy ta-

kich, które nie posiadają *uchwytu* (ang. *handle*). Wiąże się to z faktem, że nie są dla nich rezerwowane zasoby systemowe. I tak na przykład, kiedy wahamy się czy użyć komponentu **Panel** czy **Bevel**, bardziej wydajnym okaże się zastosowanie komponentu **Bevel** z racji tego, że nie jest on traktowany przez system jako nowe okno (nie jest mu przydzielany identyfikator – **handler**), jest tylko przerysowywany na formatce. Polecam również ustawienie **ParentFont** na wartość **True**. Spowoduje to, że instancja fontów nie będzie osobno ładowana dla każdego komponentu.

Korzystajmy z dyrektywy kompilatora **{$o+}** – która sprawia, że kompilator próbuje optymalizować nasz kod. Alokuje zmienne w rejestrach procesora etc. Jednak do celów analizy i debugowania aplikacji twórcy Delphi zalecają wyłączenie optymalizacji kodu.

Jeśli w algorytmie wykorzystujemy porównywanie warunków logicznych np.:

```
if (i>5) and (x<666) then
```

pamiętajmy, aby jako pierwszy umieścić warunek częściej spełniany (prawdziwy). W przypadku występowania kilku warunków logicznych można jeden test logiczny można rozbić na kilka.

Zmienne wskaźnikowe

Bardzo przydatną technika optymalizacyjna jest stosowanie zmiennych wskaźnikowych. Wielu z programistów Delphi podchodzi z nieufnością do zmiennych wskaźnikowych. Nieudolnie wykorzystywane wskaźniki prowadza do powstawania wycieków pamięci, błędów dostępu i innych równie przyjemnych nisko poziomowych zjawisk. Jednak wskaźniki odpowiednio używane stanowią użyteczne i potężne narzędzie optymalizacyjne. Nie należy popadać w skrajność i modyfikować wszystkich zmiennych w programie na typ wskaźnikowy. Zaletą zmiennych wskaźnikowych jest fakt, że zostają one przekształcone na zmienne rejestrowe. Całość procesu nie wymaga zmiany kodu aplikacji. Typ zmiennej wskaźnikowej informuje jedynie kompilator, że ten może traktować zmienne tego typu jako zmienne rejestrowe. Idealnym zastosowaniem dla zmiennych typu wskaźnikowego jest adresowanie skomplikowanych struktur danych. Ponieważ każda struktura (obiekt, klasa) w Delphi niejawnie traktowana jest jako wskaźnik.

Na niektórych wolniejszych maszynach warto również przeanalizować czy korzystniejsze okaże się zastosowanie tablic czy list (jedno lub dwukierunkowych). W przypadku procesorów starszych niż PIII należy zastanowić się nad sposobem dostępu do danych. Jeśli ma to być dostęp swobodny – należy skłaniać się ku rozwiązaniu tablicowemu. W przypadku dostępu sekwencyjnego należy jednak zastosować listy. Tablice pod względem wydajnościowym lepiej nadają się do przechowywania danych prostych typów, natomiast listy są lepsze dla większych danych. Ale na maszynach wyposażonych w procesory nowszych generacji nie ma tego typu wątpliwości. Dostęp do tablic jest zawsze szybszy.

Delphi ma rozbudowaną obsługę typu tablicowego. Środowisko oferuje cztery rodzaje tablic:

- statyczne – są to klasyczne pascalowe tablice (np. **tab:array[1..10] of string**)
- dynamiczne – wprowadzone do środowiska od wersji D4 (np. **tab:array of string**)
- wskaźnikowe – są to wskaźniki do tablic statycznych
- otwarte – są to tablice dynamiczne ściśle zarezerwowane do przekazywania parametrów

Implementacja tych typów tablic różni się dość znacznie. Ze względu na szybkość operacji preferowane są tablice statyczne i wskaźnikowe. Działania na nich są o wiele szybsze niż działania na strukturach o nie określonym na wstępie rozmiarze (jakimi są tablice dynamiczne i otwarte). Są jednak tysiące zastosowań, w których to właśnie elastyczne tablice dynamiczne nadają się lepiej niż statyczne lub wskaźnikowe (zarządzanie którymi nie należy do najprostszych). Tablice dynamiczne przypominają swoją filozofią typ **Ansistring**. Zmienna jest tak naprawdę wskaźnikiem do pierwszego elementu tablicy – dlatego konwersja tablicy dynamicznej na tablice wskaźnikową to tylko kwestia przypisania. Długość tablicy dynamicznej zapisana jest przed pierwszym elementem. Aby odczytać tą wartość wystarczy posłużyć się funkcjami **length** lub **high** (decato funkcja **high** wywołuje funkcję **length**). Jeżeli w danym algorytmie posługujemy się kilkukrotnie długością tablicy – lepszym rozwiązaniem jest zapamiętanie tej wielkości do lokalnej zmiennej w celu późniejszego odczytu.

Wykorzystywanie tablic otwartych jako parametrów wiąże się z pewnym nakładem czasowym, dlatego należy przekazywać je jako parametry **const** lub **var**. Przypominam jednak, że przekazanie tablicy jako parametru **const** nie blokuje możliwości zmiany jej zawartości – jedynie możliwość modyfikacji całej tablicy.

Do złych manier programistycznych należy również nadużywanie wyjątków. Sterowanie wykonywania algorytmu poprzez generowanie wyjątków wiąże się ze sporym spadkiem wydajności. W celu sterowania wykonywania programu proponowałbym zastosowanie instrukcji **exit**, **break** i **continue**.

Wiele problemów związanych z wydajnością swoje źródło ma w operacjach związanych z rzutowaniem typów. Z zagadnieniem tym bezpośrednio wiąże się technika nadpisywania „*overlay*" zmiennych używając dyrektywy **absolute**. Jest ona stosowana jako substytut rzutowowania typów. Niestety ten sposób uniemożliwia przekształcenie zmiennej na zmienną rejestrową. Ze względów optymalizacyjnych nie polecam korzystania z tej techniki; bardziej opłacalnym rozwiązaniem jest zrzutowanie zmiennej wyjściowej na nowy typ oraz zapisanie w ten sposób zmiennej.

Zbiory

Przyglądając się eksperymentom przeprowadzanym na zbiorach danych (**set**) można dojść do ciekawych wniosków. Otóż okazuje się że konstrukcja postaci: *s:=s+['a']* jest o dwa rzędy wielkości wolniejsza od polecenia **include**(s,'a'). Rozsądnym zatem wydaje się być natychmiastowe zastąpienie wszystkich operacji dodawania pojedynczych elementów do zbioru poprzez zastosowanie polecenia **include**. Zresztą obserwując implementacje podstawowych modułów Delphi zobaczymy, że wszystkie operacje związane z dodawaniem lub usuwaniem elementu zbioru (np. komponentu) wykonywane są z użyciem poleceń **include** i **exclude**.

Jeśli korzystamy z danych typu wyliczeniowego (np. *dniTyg=(poniedziałek,wtorek...)*) dobrze jest wymusić, aby wszystkie dane były 32-bitowe. Do tego celu należy posłużyć się dyrektywami kompilatora **{$z}.** I tak, jeśli używany przez nas typ ma więcej niż 256 elementów, zastosowanie odpowiedniej dyrektywy zwiększy wydajność programu.

Pętle

W przypadku analizowania algorytmu wykonującego operację na liczbach rzeczywistych przydatnym narzędziem dostępnym w Delphi jest podgląd stosu ośmiu rejestrów *FP* (ang. *floating point* – FP).

W przypadku małych pętli występujących w algorytmie można pokusić się o zastosowanie metody „skracania przebiegu pętli". Generalnie polega ona na wykonywaniu tego, co oryginalnie było wykonywane w kilku iterakcjach w obrębie jednego przebiegu pętli. Dzięki temu uzyskujemy zmniejszenie nakładów związanych z przeładowywaniem pętli. Polecam stosować tą technikę do pętli, w których kosztowne staje się każdorazowe przeładowywanie. Oto przykłady stosowania tej techniki:

```
implementation
[...]
const count=50000;
var i:integer;
    arr_i:array[0..count-1] of integer;
begin
  i := 0;
  while i < count do
  begin
    arr_i[i] := arr_i[i] + 1;
    inc(i);
  end;
```

Po zastosowaniu operacji skracania pętli kod wyglądałby tak:

```
const count=50000;
var i:integer;
    arr_i:array[0..count-1] of integer;
begin
  i := 0;
  while i < count do
  begin
    arr_i[i] := arr_i[i] + 1;
    arr_i[i+1] := arr_i[i+1] + 1;
    inc(i,2);
  end;
```

Z przeprowadzonych badań i analiz wynika, że granica opłacalności stosowania metody „skracania

przebiegu pętli" jest przekroczona przy współczynniku większym niż 4.

Jeżeli już wspomnieliśmy o pętlach kolejnym dobrym zwyczajem jest unikanie wewnątrz nich wyrażeń warunkowych i badania warunków logicznych. Większość testów logicznych wewnątrz pętli da się wyeliminować stosując technikę SPP lub podział pętli na dwie lub więcej.

Jedną z ważniejszych technik optymalizacyjnych związanych z pętlami jest redukcja ilości warunków pętli. Bardzo często spotykaną maniera programistyczna jest stosowanie pętli bazujących na kilku warunkach logicznych. Na przykład: jeśli jakiś warunek jest prawdziwy i indeks pętli jest mniejszy niż pewna wartość, wtedy wykonaj pętlę. W przypadku małych pętli (często składających się tylko z inkrementacji indeksu pętli) całość kosztów wykonania pętli to sprawdzanie warunków pętli. Prawie zawsze zmniejszenie liczby warunków pętli powoduje wzrost wydajności algorytmu. Fundamentalnym przykładem jest tutaj algorytm wyszukiwania odpowiedniego znaku w łańcuchu:

```
i := 1;
l := Length(s);
while ((i <= l) and (s[i] <> c)) do
  inc(i);
```

Proszę zwrócić uwagę, że wstawiając na koniec łańcucha poszukiwany znak (jest to przykład zastosowania tzw. wartownika) uzyskujemy połączenie obu warunków:

```
i := 1;
l := Length(s);
lastc := s[l];
s[l] := c;
while s[i] <> c do
  Inc(i);
s[l] := lastc;
```

W rezultacie uzyskujemy dwukrotny wzrost szybkości działania algorytmu. Technikę ta bardzo często stosuje się w połączeniu z SPP. W przykładach dołączonych do tego materiału załączam odpowiednik standardowej funkcji Delphi **findmax**, służącej do znajdowania maksymalnego elementu w tablicy. Dzięki zastosowaniu technik SPP oraz RIWP wyniki końcowe działania algorytmu są ponad dwukrotnie lepsze.

Część programistów uważa, że każdy algorytm napisany bezpośrednio w assemblerze jest szybszy i bardziej wydajny niż algorytm napisany w języku wyższego rzędu i tłumaczony na język maszynowy. Otóż okazuje się, że zdanie to jest prawdziwe tylko dla niektórych przypadków. Przy tworzeniu algorytmu w assemblerze należy pamiętać, że algorytm na pewnym poziomie szczegółowości bardzo mocno związany jest z danym sprzętem (procesor, rejestry, urządzenia peryferyjne) i algorytm osiągający doskonałe wyniki na jednej maszynie na innej może okazać się tzw. „wąskim gardłem" w aplikacji. Jeżeli jednak zdecydujemy się już na użycie czystego assemblera polecam dobrze zapoznać się materiałami zawartymi w napisanym przez Agnera Foga podręczniku optymalizacji assemblerowej.

Ciekawym tematem do rozważań do dyskusji jest pytanie: którą pętlę wybrać jeśli z góry wiemy ile

interakcji ma wystąpić? Najczęściej wybierana w takiej sytuacji jest pętla typu for. Jednak implementacja tej pętli w Delphi sprowadza się do translacji na pseudokod. I tak:

```
For i:=m to n do
      a[i]:=a[i]+b[i];
```

najczęściej zostaje przez kompilator zamieniona na kod postaci:

```
pa:=@a[m];
  pb:=@b[m];
  counter= m-n+1;
ifcounter>0 then
    repeat
  pa^:=pa^+pb^
  inc(pa);
  inc(pb);
  dec(counter);
until counter=0;
```

Jak widać tego typu translacja wprowadza pewien narzut czasowy związany ze zmiana wartości trzech zmiennych. Dlatego zastosowanie pętli for do algorytmu, w którym znana jest z góry ilość iterakcji nie jest jednoznacznie najlepszym rozwiązaniem. Czasem pętla rodzaju **while** bywa lepszą opcją. Dzieje się tak, jeśli wewnątrz pętli nie mamy do czynienia z tablicami lub mamy do czynienia jedynie z tablicami jednowymiarowymi o wielkości elementu nie przekraczającej 8 bajtów. W takim wypadku kod dla pętli **while** będzie bardziej efektywny. Natomiast do pętli operujących na wielowymiarowych tablicach o elementach większych niż 8 bajtów, lepiej nadaje się konstrukcja for. Również: jeśli wewnątrz pętli nie jest wykorzystywany indeks należy skłaniać się ku rozwiązaniu z for. Tak samo należy postąpić, jeśli granice indeksu zmieniają się dynamicznie w trakcie wykonywania programu. W celu poprawy wydajności pętli **while** należy jak najbardziej uprościć jej warunek.

Jeśli pętla **while** operuje na tablicach można zastosować dodatkową technikę poprawiającą wydajność algorytmu. Polega ona na przesunięciu zmiennej odpowiedzialnej za przechowywanie wartości indeksu. Rejestr procesora do tej pory przechowujący wartość indeksu jest zwalniany. W tym celu należy zmodyfikować wartość licznika pętli, tak żeby była liczona od wartości ujemnej do zera. Uwolnione rejestry procesora mogą wtedy przetrzymywać inne strategiczne dane.

Konstrukcja Case

Sporo można osiągnąć optymalizując algorytmy zawierające wyrażenie **Case**. Jest to dość skomplikowane wyrażenie z punktu widzenia kompilatora i ma on z nim dość dużo pracy. Najpierw lista wartości/zakresów zostaje posortowana (płynie z tego wniosek, że obojętne jest w jakiej kolejności sprecyzujemy

warunki). Następnie kompilator posługuje się drzewem porównań binarnych oraz skonstruowaną tablicą adresów skoków (**jump adress table**) sprawdzając warunki wyrażenia **case**. Decyzja o wygenerowaniu tablicy adresów skoków jest podejmowana na podstawie współczynnika gęstości odczytywanego z drzewa binarnego. Jeśli współczynnik jest wystarczająco duży generowana jest tablica, jeśli nie lista wartości jest dzielona na dwie połowy i każda z nich jest przeszukiwana oddzielnie. Czyli porównanie współczynnika gęstości powoduje wygenerowanie tablicy lub podział listy. Algorytm jest powtarzany, aż do obsłużenia wszystkich przypadków. Jak widać podział drzewa jest tutaj kluczowy z punktu widzenia procesu optymalizacyjnego. Jeśli więc jesteśmy w stanie pogrupować warunki, dobrym sposobem jest stworzenie osobnych konstrukcji **Case** dla każdego z zakresów.

Przykład:

```
Case x of
    100 : proc1;
    101 : proc2;
    102 : proc3;
    103 : proc4;
    104 : proc5;
    105 : proc6;
    200 : proc9;
    201 : proc10;
    202 : proc11;
    203 : proc12;
    204 : proc13;
  end;
```

należy zamienić na:

```
  Case x of
    100..105 :
  case x of
            100 : proc1;
       101 : proc2;
            102 : proc3;
            103 : proc4;
            104 : proc5;
            105 : proc6;
  end;
200..204 :
  case x of
            200 : proc9;
            201 : proc10;
            202 : proc11;
            203 : proc12;
            204 : proc13;
       end;
  end;
```

Jeśli wiem, że któryś z warunków wykonywany jest częściej w celach optymalizacyjnych należy przesunąć go poza obręb wyrażenia **Case**, np.:

```
if x=101 then
    procedura_czesto_wywolywana1
  else
    Case x of
      100 :proc1;
      102 :proc2;
...
end;
```

Typy danych

Obecnie kod generowany przez Delphi jest 32-bitowy. Dlatego najlepiej wszystkie zmienne występujące w programie deklarować jako 32-bitowe. Każdorazowe odwołanie się do zmiennych 16-bitowych (np. **shortint**, **word**) powoduje czasowe przełączenie procesora w tryb 16 bitowy. Wykorzystywanie 16-bitowych zmiennych znacznie wydłuża czas operacji. Stosunkowo lepiej w konfrontacji z 16 „bitowcami" wypadają zmienne 8-bitowe – **byte**, **smallint** etc. (oczywiście pod pewnymi warunkami, z których najważniejszy jest, aby nie łączyć ich ze zmiennymi 32-bitowymi). Jeśli już konieczne jest używanie mniejszych zmiennych – np. w celach kompatybilności ze wcześniejszymi rozwiązaniami, należy – kiedy to tylko możliwe – skonwertować je na zmienne 32-bitowe.

Przeważnie skomplikowane wyrażenia logiczne są rozbijane przez kompilatory na mniejsze, mniej złożone koniunkcje warunków logicznych. Zawsze należy próbować samemu rozłożyć dane skomplikowane wyrażenie na mniej złożone. Jako że trening czyni mistrza, już po pewnym czasie nasz sposób będzie wydajniejszy od tego wykonanego przez kompilator.

A oto przykład jak można zoptymalizować porównywanie zmiennych.

```
If (x > 0) and (x < 10) then
    Caption:='prawidłowa wartość';
```

lub

```
If (((ch > = 'a') and (ch < = 'z')) or ((ch > = '0') and (ch < = '9'))) then
  Caption:='prawidłowa wartość';
```

Otóż: w pierwszym i drugim przypadku w wyrażeniu logicznym porównywana jest tylko jedna zmienna. Wzrost wydajności (jak również przejrzystości) rozwiązania uzyskujemy stosując zbiory danych:

```
If x in [0..10] then
   Caption:='prawidłowa wartość';
```

```
If ch in ['0'..'9', 'a'..'z'] then
   Caption:='prawidłowa wartość';
```

Efektywność rozwiązania w największej mierze zależy od porównywanych zmiennych. I tak dla przykładu ze zmienną x: jeśli wiadomo, że wartość tej zmiennej najczęściej będzie należała do tego przedziału, lepiej jest zastosować zbiory.

Bardzo duży wkład w optymalizację pracy z łańcuchami (typ string) w Delphi wniosła grupa EFD systems. W internecie można znaleźć bibliotekę **HyperString** oferującą bogaty zestaw funkcjonalności związanych z typem string. Jest to rozwiązanie o wiele bardziej wydajne niż standardowe funkcje oferowane przez Delphi. W większości przypadków, kiedy zależy nam na zwiększeniu prędkości działania algorytmu operującego na łańcuchach, wystarczy użyć właśnie tej biblioteki.

Dość często spotykanym błędem jest tzw. podwójna inicjacja zmiennej łańcuchowej. Obecnie w Delphi wszystkie zmienne deklarowane jako string są, *de facto*, przez kompilator traktowane jako **Ansistring**. I tak każda zmienna typu **Ansisring** już podczas jest inicjalizowana podczas tworzenia jako pusty ciąg. Dlatego kod postaci:

```
var s:string
begin
s:=''
```

wprowadza niepotrzebną redundancję. Nie dotyczy to łańcuchów zwracanych jako rezultat funkcji.

Zachęcam do korzystania z bardzo wydajnego mechanizmu alokacji zmiennych łańcuchowych. Jednak należy korzystać z tego mechanizmu z głową, ponieważ zastosowanie go w nieodpowiedni sposób powoduje zbytnie obciążanie systemu. Proszę zwrócić uwagę na poniższy przykład.

```
s1:=''
for i:=0 to length(s2) do
   s1:=s1+s2[i]
```

Wadą tego algorytmu jest fakt, że pamięć dla zmiennej s1 jest realokowana w pętli. Każda iteracja wiąże się z operacją rozszerzenia pamięci dla zmiennej s1, co z kolei wiąże się ze stratą czasu. Zdecydowanie trafniejszym rozwiązaniem wydaje się:

```
setlength(s1, length(s2)-1);
for i:=0 to length(s2) do
    s1[i-1]:=s2[i];
```

Tutaj pamięć dla zmiennej s1 alokowana jest tylko raz. Jak widać stare, statyczne deklaracje zmiennych łańcuchowych z Pascala, zostały zamienione na nowe dynamiczne. Rodzi to nowe możliwości, ale stwarza też nowe problemy.

I tak na przykład, często powtarzanym błędem jest sytuacja z tworzeniem tymczasowych zmiennych

łańcuchowych za pomocą polecenia **copy**.

```
if copy(s1,5,5)=copy(s2,7,5) then
```

Również w tej sytuacji problemem jest alokacja pamięci dla łańcucha tymczasowego. Niedogodność tą można wyeliminować stosując np. takie rozwiązanie:

```
i:=1;
koniec := false;
repeat
  koniec := s1[i+4] <> s2[i+6];
  inc(i);
until koniec or (i>5);
```

Poza tym trzeba być świadomym, że polecenie **copy** zawsze na początku kopiuje całą zawartość string. Dlatego, jeżeli mamy do przeprowadzenia operację usunięcia części łańcucha, lepiej jest użyć polecenia **delete** niż **copy**.

Kolejną zasadą dotyczącą łańcuchów jest wystrzegać się używania typu **shortString**. Każde użycie tego typu wiąże się z dodatkowymi instrukcjami konwersji przeprowadzanymi w tle przez kompilator.

Optymalizacja działań matematycznych w Delphi

Mnożenie, potęgowanie, pierwiastkowanie, wszystkie te operacje trwają ułamek sekundy na obecnych maszynach, ułamek sekundy do czasu, gdy nie musimy wykonać tysięcy czy miliony iteracji danego działania. W takiej sytuacji odpowiedni dobór użytych funkcji może mieć duży wpływ na czas trwania całej operacji.

Podrozdział ten pokazuje, jakich błędów unikać pisząc w Delphi skomplikowane działania matematyczne, aby zoptymalizować czas wykonywania.

Power, Sqr, Sqrt

Funkcja **Power** przyjmuje 2 argumenty: pierwszym jest liczba, którą chcemy podnieść do potęgi, a drugim wykładnik potęgi. Należy dokładnie przemyśleć jej użycie w programie i – jeśli to możliwe – zastąpić innym, równoważnym wyrażeniem.

W sytuacji, w której podnosimy liczbę do potęgi drugiej, bądź wyciągamy pierwiastek stopnia drugiego bezwzględnie stosujmy funkcje **Sqr** oraz **Sqrt**. A co w przypadku, kiedy mamy do czynienia z pierwiastkiem 4 stopnia? Zagnieżdżone dwie funkcję **Sqrt** będą na pewno szybsze niż jeden **Power**.

Oto małe porównanie szybkości wykonywania się poszczególnych operacji:

Funkcja	Czas wykonania
Sqr(x)	0,0547ms
Sqrt(x)	0,6396ms
Power(x, 2)	0,8642ms
Power(x, 0.5)	3,9702ms

Tabela 4 Porównanie szybkości wykonywania sie poszczególnych operacji

I na koniec wynik testu z pierwiastkiem czwartego stopnia:

Funkcja	Czas wykonania
Sqrt(Sqrt(x))	1,7472ms
Power(x, 0.25)	4,5896ms

Tabela 5 Wynik testu z pierwiastkiem czwartego stopnia

Liczę, że te kilka porad pozwoli na optymalizację wielokrotnie wykonywanych działań matematycznych w Delphi. Oczywiście należy też pamiętać o doprowadzeniu danego równania do jak najprostszej postaci przed przystąpieniem do implementacji w programie.

Operacje zmiennoprzecinkowe

Bardzo dużo można też zyskać na optymalizacji operacji zmiennoprzecinkowych. Podstawową informacją dotyczącą pracy z liczbami zmiennoprzecinkowymi jest fakt, że zmienne typu **extended** są wewnętrznie przechowywane jako wielkości 80-bitowe!!! Każda operacja arytmetyczna wykonywana na zmiennych tego typu zwiększa czas wykonywania algorytmu. Jest to związane z dodatkowym czasem przeznaczonym na załadowanie i składowanie takich zmiennych.

Duże straty w wydajności spowodowane są również operacjami konwersji różnych typów zmiennoprzecinkowych – każdorazowo, gdy dokonujemy przypisania jednej zmiennej do drugiej lub przesyłamy zmienną jako parametr funkcji. W obu tych przypadkach zmienna jest ładowana na stos FP a następnie zapisywana jako nowy typ.

Jeżeli w programie zdecydujemy się na używanie stałych o wartościach zmiennoprzecinkowych, należy sprecyzować typ stałej. Jeśli tego nie zrobimy domyślnie, kompilator cześć ułamkową zapisze do zmiennej typu **extended**. Dlatego stałe zmiennoprzecinkowe najlepiej deklarować w ten sposób:

```
const
StOs:double = 3.417666;
```

W ten sposób zadeklarowana zmienna: po pierwsze – będzie zajmowała mniej pamięci; po drugie –

będzie szybsza.

Operacje na liczbach zmiennoprzecinkowych pochłaniają wiele czasu i zasobów. Szczególnie czasochłonne jest dzielenie. Jednak można i w tym przypadku zaoszczędzić trochę czasu. Domyślnie w Delphi dokładność operacji zmiennoprzecinkowych ustawiona jest na maksymalną dokładność. Jeśli wiemy, że w danym momencie algorytm nie wymaga tak dużej precyzji możemy zmienić dokładność obliczeń, co w znaczy sposób wpłynie na przyspieszenie pracy algorytmu. Dokonujemy tego zmieniając słowo kontrolne (**control word**) jednostki FPU. W Delphi wprowadzono specjalną funkcję umożliwiającą zmianę statusu słowa kontrolnego: Set8087CW oraz zmienną Default8087:

Typ	Składnia
Single	`Set8087CW(Default8087CW and $FCFF);`
Double	`Set8087CW((Default8087CW and $FCFF) or $0200);`
Extended	`Set8087CW(Default8087CW or $0300);`

Tabela 6 Zmiana słów kontrolnych środowiska Delphi

Od wersji Delphi 6 można również posługiwać się funkcją **SetPrecisionMode**, która ma o wiele prostszą składnię, ale tak naprawdę sprowadza się do wykonania jednej z operacji przedstawionej w tabeli powyżej.

Przypominam, że przyspieszenie dotyczy wyłącznie operacji dzielenia. Twórcy Delphi zalecają wykorzystanie zestawu tych funkcji przy obliczeniach dla OpenGL.

Kiedy musimy w kodzie programu zawrzeć zaokrąglanie liczb zmiennoprzecinkowych dostępnymi w Delphi funkcjami realizującymi to zadanie są **Trunc** i **Round**. Ze względu na prędkość działania proponuję skorzystać z funkcji **Round**, ponieważ jest o prawie 1/3 szybszy od **Trunc**.

Jednak – pomimo wachlarza funkcji związanych z działaniami na danych zmiennoprzecinkowych oferowanych przez Delphi – środowisko to nie oferuje żadnych optymalizacji dla tych operacji. Całość procesu optymalizacji spada na użytkownika. W operacjach na danych zmiennoprzecinkowych najbardziej czasochłonna operacją jest dzielenie. Operacja ta jest kilkadziesiąt razy wolniejsza niż operacja mnożenia. Jeżeli operacja dzielenia występuje w pętli należy przenieść ją poza pętlę.

Okazuje się również, że samo porównywanie, czy dana zmienna typu (**double**, **single** czy **extended**) jest równa zero, nie jest zoptymalizowana. Dlatego w celu przyspieszenia pracy algorytmu stosuje się konstrukcje zastępcze. Jeśli chodzi o porównywanie zmiennej typu single:

```
var sngl:single;
begin
if DWord(Pointer(sngl)) shl 1 = 0 then
```

Rozwiązanie to jest o wiele szybsze od zwykłego porównania *if sngl=0*. Natomiast, jeśli chodzi o porównywanie wartości typu double, sprawa jest nieco bardziej skomplikowana**:**

```
Var
dbl:double;
doubleData:tdoubledata absolute dbl;
```

```
begin
      if (DoubleData.hi shl 1 ) + DoubleData.Lo = 0 then
```

Dobór właściwości komponentów

Jak wiadomo, wyświetlenie zawartości jakiegoś komponentu na ekran jest bardzo czasochłonnym zadaniem. Z tego powodu warto przyjrzeć się właściwościom używanych komponentów. Przeanalizujmy kontrolkę **TLabel**.

Komponent ten ma trzy własności, które bezpośrednio wpływają na to, jak komponent będzie rysowany, lecz nie mają wpływu na ostateczny wygląd w większości sytuacji:

- **AutoSize** – ustawienie tej właściwości powoduje, że rozmiar komponentu dopasowuje się do zawartości tekstu. Jeśli rozmiar jest odpowiednio duży, a kontrolka nie ma reagować na położenie kursora myszy i nie ma innego tła, to różnic nigdy nie zobaczymy. Warto jednak nadmienić, że przewidzieć należy także fakt, że wpływ na rozmiar tekstu w głównej mierze – poza jego treścią – mają ustawienia czcionek u użytkownika.
- **ShowAccelChar** – włączenie spowoduje, że **label** stanie się aktywny na skróty klawiaturowe, a litera poprzedzona znakiem „&” zostanie podkreślona i stanie się klawiszem skrótu (w połączeniu z **Alt**). W większości przypadków jest to niepotrzebne, a czasem wręcz przeszkadza, gdy chcemy wyświetlać znaki ampersand (&) na etykiecie.
- **Transparent** – to ustawienie powoduje, że tło pod tekstem staje się przeźroczyste. Jeśli tło jest w tym samym kolorze co element na którym umieszczona została etykieta, to ustawienie to nie zmienia niczego wizualnie na formie.

Jak zatem wpływają poszczególne opcje na wydajność wyświetlania?

```
for x:=1 to 70000 do begin
  Label1.Caption:=Format('%d = %x',[x,x]);
  Label1.Repaint;
end;
```

Kod ten wyświetla 70 000 razy różny tekst w postaci liczby od 1 do 70 000 w postaci dziesiętnej oraz szesnastkowej, za każdym razem wymuszając przerysowanie. Wyniki zawarte są w poniższej tabeli. Przedstawia ona wszystkie możliwe kombinacje opisywanych ustawień oraz średni czas wykonywania jednego przebiegu pętli. Dodatkowo w prawej części przedstawione są różnice pomiędzy włączeniem a wyłączeniem danej opcji.

Transparent	AutoSize	ShowAccelChar	Czas [µs]	Różnice [µs]		
				ShowAccelChar	AutoSize	Transparent
NIE	NIE	NIE	**157**			
NIE	NIE	TAK	**159**	2		
NIE	TAK	NIE	**258**		101	

NIE	TAK	TAK	**260**	2	101	
TAK	NIE	NIE	**186**			29
TAK	NIE	TAK	**188**	2		29
TAK	TAK	NIE	**300**		114	42
TAK	TAK	TAK	**303**	3	115	43

Tabela 7 Wyniki działania komponentu TLabel z różnymi ustawieniami właściwości

Jak widać z powyższej tabeli, różnica pomiędzy skrajnymi przypadkami jest prawie dwukrotna! Najmniejszy wpływ na wydajność ma wyłączenie własności **ShowAccelChar** – możemy zaoszczędzić ok. 2 μs, czyli maksymalnie 1,3%. Włączenie przeźroczystości to strata już znacznie większa sięgająca 18%. Największy spadek wydajności zaobserwujemy włączając parametr **AutoSize** – możemy spowolnić w ten sposób wyświetlanie nawet o ponad 60%!

Warto jeszcze zwrócić uwagę na jeden fakt – domyślne ustawienia nie są najwydajniejszymi. Wyłączając automatyczne dopasowanie rozmiaru oraz przypisywanie skrótów zyskamy ok. 40% szybsze przerysowywanie komponentu **TLabel**. Wystarczy tylko pamiętać, by wcześniej nadać właściwy rozmiar uwzględniając przy tym możliwość wystąpienia większej czcionki u użytkownika.

Blokowanie przerysowania

Częstym niedopatrzeniem programistów jest zadbanie o to, czy komponent ma dokonać aktualizacji swej zawartości, czy też nie. Problem dotyczy wszystkich komponentów wizualnych, których zawartość bezpośrednio bazuje na typie **TStrings** (a więc i dziedziczący **TStringList**). Wszystkie obiekty tej klasy posiadają własność **BeginUpdate**, która określa, czy komponent ma dokonać przerysowania bądź innych działań związanych ze zmianą zawartości string-listy. Polecenia tego używa się przed blokiem programu, który dokonuje modyfikacji listy (dodanie, zmiana treści, usunięcie). To, czy lista jest w trakcie aktualizacji można odczytać poprzez zmienną **UpdateCount**. Po zakończeniu aktualizacji należy koniecznie wywołać metodę **EndUpdate**, która wywoła działania związane ze zmianą zawartości string-listy. Oczywiście tego typu zabiegów nie ma sensu wykonywać, jeśli dokonywana jest pojedyncza zmiana (te instrukcje są także wywoływane wewnętrznie przez metody **Add, Delete** itp.).

Rysowanie w oknach

Kolejny, bardzo często popełniany błąd wiążący się z problemem przerysowywania interfejsu, związany jest bezpośrednio z grafiką. Początkujący programiści próbują ułatwić sobie pracę poprzez rysowanie bezpośrednio na **Canvas** komponentu **TImage**. Nie trzeba się wówczas martwić przerysowaniem czy innymi aspektami. Niestety takie rozwiązanie jest bardzo powolne – szczególnie, jeśli rysowana jest większa ilość obiektów, lub następuje bardzo częste przerysowanie (np. wirtualny efekt przeciągania jakiejś figury). Dużo

szybszym jest rysowanie po obszarze wirtualnej bitmapy (czyli zmiennej typu **TBitmap**), a następnie – po dokonaniu wszystkich operacji rysowania, których wynik można przedstawić użytkownikowi – przerysowanie jej w całości na komponent wizualny (poleceniem **Draw**). Warto także pamiętać, że rysowanie dużo szybciej odbywać się będzie po komponencie **TPaintBox**, gdyż nie wywołuje on automatycznego przerysowania zawartości. Wadą jest to, że należy samemu obsłużyć zdarzenie **OnPaint**.

Redukcja rozmiaru aplikacji

W przypadku niezbyt złożonych aplikacji, można w znaczący sposób zmniejszyć rozmiar wynikowy nie używając bibliotek **VCL**. Zaprezentowane zostanie to na przykładzie prostego programu, który ma za zadanie wyświetlać co sekundę liczbę oraz całkowity rozmiar plików znajdujących się w koszu. Na formatce umieszczamy dwa komponenty **TLabel**, jeden **TButton** oraz **TTimer**. Oto kod programu:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  PSHQueryRBInfo = ^TSHQueryRBInfo;
  TSHQueryRBInfo = packed record
    cbSize: DWORD;
    i64Size: Int64;
    i64NumItems: Int64;
  end;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  DllVersion: Integer;
  SHQueryRBInfo: TSHQueryRBInfo;

implementation

{$R *.DFM}

function SHQueryRecycleBin(szRootPath: PChar; SHQueryRBInfo: PSHQueryRBInfo): HResult;
stdcall; external 'shell32.dll' Name 'SHQueryRecycleBinA';
```

```
function GetDllVersion(FileName: String): Integer;
var
  InfoSize, Wnd: DWORD;
  VerBuf: Pointer;
  FI: PVSFixedFileInfo;
  VerSize: DWORD;
begin
  Result   := 0;
  InfoSize := GetFileVersionInfoSize(PChar(FileName), Wnd);
  if InfoSize <> 0 then
  begin
    GetMem(VerBuf, InfoSize);
    try
      if GetFileVersionInfo(PChar(FileName), Wnd, InfoSize, VerBuf) then
        if VerQueryValue(VerBuf, '\', Pointer(FI), VerSize) then
          Result := FI.dwFileVersionMS;
    finally
      FreeMem(VerBuf);
    end;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  DllVersion := GetDllVersion(PChar('shell32.dll'));
  if DllVersion >= $00040048 then
  begin
    FillChar(SHQueryRBInfo, SizeOf(TSHQueryRBInfo), #0);
    SHQueryRBInfo.cbSize := SizeOf(TSHQueryRBInfo);
    SHQueryRecycleBin(nil, @SHQueryRBInfo);
    Label1.Caption := 'Całkowity rozmiar plików w koszu: ' + IntToStr(SHQueryRBInfo.
i64Size) + ' bajtów';
    Label2.Caption := 'Liczba plików w koszu: ' + IntToStr(SHQueryRBInfo.i64NumItems);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled := False;
  Close;
end;
end.
```

Standardowo, po skompilowaniu program zajmuje **298 kB**. Jest to dość znaczna wielkość, zważywszy na skromną logike programu. Optymalizację proponuję zacząć od zainstalowania otwartej biblioteki **KOL&MCK** (można ja pobrać z http://kolmck.net/). Przy tworzeniu nowego projektu postępujemy zgodnie ze wskazówkami znajdującymi się w pobranej paczce. Oprócz wspomnianego pliku ściągamy również zamienniki standardowych bibliotek **System**, **SysUtils** oraz **Classes** (zakładka **System**). W Delphi w polu **Conditional Defines** (zakładka **Directories/Conditionals**) do parametru **KOL_MCK** dodajemy **SMALLEST_CODE**. Na formie umieszczamy: 2x **TKOLLabe**l oraz po jednym **TKOLButton** i **TKOLTimer**. Część odpowiadająca za funkcjonowanie programu wygląda analogicznie jak w przypadku VCL:

```
{ KOL MCK } // Do not remove this line!
{$DEFINE KOL_MCK}
unit Unit1;

interface
```

```
{$IFDEF KOL_MCK}
uses Windows, Messages, KOL {$IFNDEF KOL_MCK}, mirror, Classes, Controls, mckCtrls,
mckObjs, Graphics {$ENDIF (place your units here->)};
{$ELSE}
{$I uses.inc}
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
{$ENDIF}

type
  PSHQueryRBInfo = ^TSHQueryRBInfo;
  TSHQueryRBInfo = packed record
    cbSize: DWORD;
    i64Size: I64;
    i64NumItems: I64;
  end;

type
  {$IFDEF KOL_MCK}
  {$I MCKfakeClasses.inc}
  {$IFDEF KOLCLASSES} {$I TForm1class.inc} {$ELSE OBJECTS} PForm1 = ^TForm1; {$ENDIF
CLASSES/OBJECTS}
  {$IFDEF KOLCLASSES}{$I TForm1.inc}{$ELSE} TForm1 = object(TObj) {$ENDIF}
    Form: PControl;
  {$ELSE not_KOL_MCK}
  TForm1 = class(TForm)
  {$ENDIF KOL_MCK}
    KOLProject1: TKOLProject;
    KOLForm1: TKOLForm;
    Timer1: TKOLTimer;
    Button1: TKOLButton;
    Label1: TKOLLabel;
    Label2: TKOLLabel;
    procedure Timer1Timer(Sender: PObj);
    procedure Button1Click(Sender: PObj);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1 {$IFDEF KOL_MCK} : PForm1 {$ELSE} : TForm1 {$ENDIF} ;
  DllVersion: Integer;
  SHQueryRBInfo: TSHQueryRBInfo;

{$IFDEF KOL_MCK}
procedure NewForm1( var Result: PForm1; AParent: PControl );
{$ENDIF}

implementation

{$IFNDEF KOL_MCK} {$R *.DFM} {$ENDIF}

{$IFDEF KOL_MCK}
{$I Unit1_1.inc}
{$ENDIF}

function SHQueryRecycleBin(szRootPath: PChar; SHQueryRBInfo: PSHQueryRBInfo): HResult;
stdcall; external 'shell32.dll' Name 'SHQueryRecycleBinA';

function GetDllVersion(FileName: String): Integer;
var
  InfoSize, Wnd: DWORD;
  VerBuf: Pointer;
```

```
  FI: PVSFixedFileInfo;
  VerSize: DWORD;
begin
  Result   := 0;
  InfoSize := GetFileVersionInfoSize(PChar(FileName), Wnd);
  if InfoSize <> 0 then
  begin
    GetMem(VerBuf, InfoSize);
    try
      if GetFileVersionInfo(PChar(FileName), Wnd, InfoSize, VerBuf) then
        if VerQueryValue(VerBuf, '\', Pointer(FI), VerSize) then
          Result := FI.dwFileVersionMS;
    finally
      FreeMem(VerBuf);
    end;
  end;
end;

procedure TForm1.Timer1Timer(Sender: PObj);
begin
  DllVersion := GetDllVersion(PChar('shell32.dll'));
  if DllVersion >= $00040048 then
  begin
    FillChar(SHQueryRBInfo, SizeOf(TSHQueryRBInfo), #0);
    SHQueryRBInfo.cbSize := SizeOf(TSHQueryRBInfo);
    SHQueryRecycleBin(nil, @SHQueryRBInfo);
    Label1.Caption := 'Całkowity rozmiar plików w koszu: ' + Num2Bytes(Int64_2Dou-
ble(SHQueryRBInfo.i64Size));
    Label2.Caption := 'Liczba plików w koszu: ' + Int64_2Str(SHQueryRBInfo.i64Nu-
mItems);
  end;
end;

procedure TForm1.Button1Click(Sender: PObj);
begin
  Timer1.Enabled := False;
  PostQuitMessage(0);
end;
```

Przedstawioną aplikację możemy zbudować również nie używając MCK (graficznego interfejsu dla KOL). Możemy zastosować jedynie standardowe funkcjonalności biblioteki KOL:

```
program KOL_Trash;

uses Windows, KOL;

type
  PSHQueryRBInfo = ^TSHQueryRBInfo;
  TSHQueryRBInfo = packed record
    cbSize: DWORD;
    i64Size: I64;
    i64NumItems: I64;
  end;

var
  DllVersion: Integer;
  SHQueryRBInfo: TSHQueryRBInfo;
  W, B, L1, L2 : PControl;
  T : PTimer;

function SHQueryRecycleBin(szRootPath: PChar; SHQueryRBInfo: PSHQueryRBInfo): HResult;
stdcall; external ,shell32.dll' Name ,SHQueryRecycleBinA';
```

```
function GetDllVersion(FileName: String): Integer;
var
  InfoSize, Wnd: DWORD;
  VerBuf: Pointer;
  FI: PVSFixedFileInfo;
  VerSize: DWORD;
begin
  Result   := 0;
  InfoSize := GetFileVersionInfoSize(PChar(FileName), Wnd);
  if InfoSize <> 0 then
  begin
    GetMem(VerBuf, InfoSize);
    try
      if GetFileVersionInfo(PChar(FileName), Wnd, InfoSize, VerBuf) then
        if VerQueryValue(VerBuf, '\', Pointer(FI), VerSize) then
          Result := FI.dwFileVersionMS;
    finally
      FreeMem(VerBuf);
    end;
  end;
end;

procedure TimerTick(Dummy: Pointer; Sender: PTimer);
begin
  DllVersion := GetDllVersion(PChar('shell32.dll'));
  if DllVersion >= $00040048 then
  begin
    FillChar(SHQueryRBInfo, SizeOf(TSHQueryRBInfo), #0);
    SHQueryRBInfo.cbSize := SizeOf(TSHQueryRBInfo);
    SHQueryRecycleBin(nil, @SHQueryRBInfo);
    L1.Caption := 'Całkowity rozmiar plików w koszu: ' + Num2Bytes(Int64_2Double(SH-
QueryRBInfo.i64Size));
    L2.Caption := ,Liczba plików w koszu: , + Int64_2Str(SHQueryRBInfo.i64NumItems);
  end;
end;

procedure CloseClick(Dummy: Pointer; Sender: PControl);
begin
  T.Enabled := False;
  PostQuitMessage(0);
end;

begin
  W := NewForm(Applet, 'Kosz').SetClientSize(350,250).CenterOnParent;
  L1 := NewLabel(W, '').SetPosition(40,40).AutoSize(True);
  L2 := NewLabel(W, '').SetPosition(40,80).AutoSize(True);
  B := NewButton(W, 'Zamknij').SetPosition(100,180).SetSize(133, 33);
  B.OnClick := TOnEvent(MakeMethod(nil, @CloseClick));
  T := NewTimer(1000);
  T.OnTimer := TOnEvent(MakeMethod(nil, @TimerTick));
  T.Enabled := True;
  Run(W);
end.
```

Oprócz standardowego kodu dodana została funkcja **Num2Bytes**, która zwraca rozmiar plików kosza w przejrzystym formacie. W tym przypadku, po skompilowaniu rozmiar kodu wynikowego to **18,5 kB**. Nie osiadajmy w tym momencie na lurach. Sprawdźmy jak wielki będzie kod wynikowy z zastosowaniem WinAPI:

```
program API_Trash;

uses Windows, Messages;

type
  I64 = record Lo, Hi: DWORD;
  end;

type
  PSHQueryRBInfo = ^TSHQueryRBInfo;
  TSHQueryRBInfo = packed record
    cbSize: DWORD;
    i64Size: I64;
    i64NumItems: I64;
  end;

var
  Wnd: TWndClass;
  Msg: TMsg;
  L1, L2: HWND;

function SHQueryRecycleBin(szRootPath: PChar; SHQueryRBInfo: PSHQueryRBInfo):
HResult; stdcall; external 'shell32.dll' Name 'SHQueryRecycleBinA';
function StrFormatByteSize64(dw: I64; szBuf: PChar; uiBufSize: UINT): PChar; std-
call; external 'shlwapi.dll' name 'StrFormatByteSize64A';

function GetDllVersion(FileName: string): Integer;
var
  InfoSize, Wnd: DWORD;
  VerBuf: Pointer;
  FI: PVSFixedFileInfo;
  VerSize: DWORD;
begin
  Result   := 0;
  InfoSize := GetFileVersionInfoSize(PChar(FileName), Wnd);
  if InfoSize <> 0 then
  begin
    GetMem(VerBuf, InfoSize);
    try
      if GetFileVersionInfo(PChar(FileName), Wnd, InfoSize, VerBuf) then
        if VerQueryValue(VerBuf, '\', Pointer(FI), VerSize) then
          Result := FI.dwFileVersionMS;
    finally
      FreeMem(VerBuf);
    end;
  end;
end;

function WndProc(Wnd: HWND; uMsg: UINT; wPar: WPARAM; lPar: LPARAM): LRESULT; st-
dcall;
var
  Buffer: array[0..255] of Char;
  DllVersion: integer;
  SHQueryRBInfo: TSHQueryRBInfo;
begin
  Result := 0;
  case uMsg of
    WM_CREATE:
    begin
      CreateWindow('BUTTON', 'Zamknij', WS_CHILD or WS_VISIBLE, 100, 180, 133,
33, Wnd, 14, hInstance, nil);
      L1 := CreateWindow('STATIC', '', WS_CHILD or WS_VISIBLE, 40, 40, 300, 25,
Wnd, 0, hInstance, nil);
```

```
            L2 := CreateWindow('STATIC', '', WS_CHILD or WS_VISIBLE, 40, 80, 300, 25,
Wnd, 0, hInstance, nil);
            DllVersion := GetDllVersion(PChar(,shell32.dll'));
            if DllVersion >= $00040048 then SetTimer(Wnd, 1, 1000, nil);
          end;

          WM_TIMER:
          begin
            FillChar(SHQueryRBInfo, SizeOf(TSHQueryRBInfo), #0);
            SHQueryRBInfo.cbSize := SizeOf(TSHQueryRBInfo);
            SHQueryRecycleBin(nil, @SHQueryRBInfo);
            StrFormatByteSize64(SHQueryRBInfo.i64Size, Buffer, 255);
            SetWindowText(L1, PChar('Całkowity rozmiar plików w koszu: ' + Buffer));
            wvsprintf(Buffer, '%lu', @SHQueryRBInfo.i64NumItems);
            SetWindowText(L2, PChar(,Liczba plików w koszu: , + Buffer));
          end;

          WM_COMMAND: if wPar = 14 then
          begin
            KillTimer(Wnd,1);
            PostQuitMessage(0);
          end;

          WM_DESTROY: PostQuitMessage(0);

          else Result := DefWindowProc(Wnd, uMsg, wPar, lPar);
        end;
      end;

      begin
        with Wnd do
        begin
          lpfnWndProc := @WndProc;
          hInstance := hInstance;
          lpszClassName := 'XPU';
          hbrBackground := COLOR_WINDOW;
          hIcon := LoadIcon(0, IDI_APPLICATION);
          hCursor := LoadCursor(0, IDC_ARROW);
        end;
        RegisterClass(Wnd);
        CreateWindow('XPU', 'Kosz', WS_VISIBLE or WS_TILEDWINDOW, (GetSystemMetrics(SM_
CXSCREEN) div 2)-350, (GetSystemMetrics(SM_CYSCREEN) div 2)-250, 350, 250, 0, 0, hIn-
stance, NIL);

        while GetMessage(msg, 0, 0, 0) do
        begin
          TranslateMessage(msg);
          DispatchMessage(msg);
        end;
      end.
```

Przypominam, że zaczynaliśmy od **298 kB**. Eksperyment z **WinApi** dał rezultaty w postaci kodu wynikowego o wielkości **8 kB**. Zachęcam, zatem do przeanalizowania kodu pod względem zastępowania standardowych komponentów VCL ich odpowiednikami dającymi lepsze rezultaty.

Optymalizacja kodu dla .NET

Poniższy rozdział prezentuje wskazówki, dzięki którym możliwe będzie optymizacja aplikacji tworzonych z wykorzystaniem frameworku. .NET. Oprócz generycznych technik optymalizacji przedstawionej w poprzednich rozdziałach, .NET daje spore spektrum technik optymalizacyjnych związanych z właściwościami tego środowiska.

Struktura kodu pośredniego jest bardzo prosta – kompilator C# nie stosuje praktycznie żadnych optymalizacji. To reguła w świecie.NET – kompilator JIT podczas kompilacji kodu pośredniego do kodu natywnego i tak dokonuje swoich optymalizacji, dlatego kompilatory języków nie muszą tego robić na poziomie kompilacji kodu języka do kodu pośredniego. Tym bardziej uzasadnione jest stosowanie technik wpływających na poprawę wydajności aplikacji.

Jednym z podstawowych prawideł optymalizacji kodu stworzonego w .NET jest preferowanie pojedynczych większych assemblies, niż tworzenie zbioru mniejszych. W przypadku, kiedy mamy do czynienia ze zbiorem mniejszych encji ładowanych w tym samym czasie, powinniśmy rozważyć połączenie ich w jedną większą całość. Nadmiarowy koszt związany z uruchomieniem wielu encji wiąże się z:

- kosztem ładowania modelu metadanych do każdej z assemblies
- pracą z różnymi stronami w pamięci w prekompilwanych obrazach dla CLR
- czasem kompilacji JIT
- sprawdzaniem polityk bezpieczeństwa

W przypadku uruchomienia jednej większej aplikacji, istnieje większe prawdopodobieństwo optymalziacji natywnego obrazu assembly z pośrednictwem narzędzia Native Generator Utility (**Ngen.exe**). Oznacza to, że newralgiczne dane będą ułożone w bardziej optymalną strukturę w pamięci i manipulacje na nich nie będą wymagały zbyt wielu operacji przemieszczania się między stronami pamięci. Należy jednak pamiętać, że nie zawsze istnieje możliwość grupowania aplikacji.

Praca ze zmiennymi typu string

Pisząc aplikację na platformę .NET prawie zawsze korzystamy z obiektów (zmiennych) typu string. Są one idealne dla operacji tekstowych, gdyż mają wbudowaną całą funkcjonalność, którą można wykorzystać. Jednak obiekty typu string różnią się od innych obiektów tym, że zmienna nie przechowuje referencji do obiektu, ale cały obiekt. Po wykonaniu poniższego kodu w zmiennej tekst dalej będzie tekst **„Hello”**, chociaż takie zachowanie jest nietypowe dla obiektów.

```
string tekst=”Hello”;
string str1;
str1=tekst;
str1+=” World”;
```

Dzieje się tak, gdyż każda operacja na zmiennej typu string tworzy kopię obiektu (zmienna tekst dalej przechowuje starą wartość, bo str1 operuje tylko na kopii). Jednak ciągłe tworzenie nowych kopii jest mało wydajne i przy operacjach na tysiącach zmiennych typu string wydajność drastycznie spada. Z tego powodu podczas bardzo wielu operacji na tekstach należy stosować klasę **StringBuilder** – ich szybkość jest wielokrotnie większa niż w przypadku stosowania zmiennych typu string. By pokazać zalety klasy **StringBuilder** zbudowałem aplikację tworzącą w pętli (o określonej przez użytkownika liczbie powtórzeń) wynikowy tekst. Im większa będzie liczba operacji, tym bardziej widoczna będzie przewaga klasy **StringBuilder** nad standardowymi obiektami klasy string. Funkcja **TestString** tworzy string wynikowy dodając wielokrotnie tekst określony w zmiennej **txt**:

```
public static string TestString(string txt,int count)
{
     string wynik=string.Empty;
     for(int i=0;i<count;i++)
     {
          wynik+=txt;
     }
     return wynik;
}
```

Funkcja **TestStringBuilder** wykonuje te same operacje, ale korzysta z klasy **StringBuilder**:

```
public static string TestStringBuilder(string txt,int count)
{
System.Text.StringBuilder text=new System.Text.StringBuilder (txt.Length*count);
     for(int i=0;i<count;i++)
     {
          text.Append(txt);
     }
     return text.ToString();
```

Zbyt częste stosowanie operacji łączenia (ang. *concatenation*) skutkuje wieloma operacjami alokacji i zwalniania zasobów. Wynika to z faktu, że każda operacja zmiany wartości zmiennej wiąże się z utworzeniem nowej zmiennej i zwolnieniem przez garbage collector starej.

- w przypadku konkatenacji zmiennych literowych – są one składane przez kompilator bezpośrednio w czasie kompilacji String *str = „Hello” + „world”*
- kiedy zdecydujemy się na konkatacje zmiennych alfanumerycznych (litery, cyfry) CLR dokona konkatenacji w czasie uruchomienia – w efekcie zastosowanie operatora „+” spowoduje stworzenie wielu obiektów typu string
- w celu konkatenacji złożonych łańcuchów lub w przypadku kilkukrotnych wywołań operacji łączenia łańcuchów należy użyć **StringBuilder**

```
String str = "Some Text";
```

```
for ( ... loop several times to build the string ...) {
str = str + " additional text ";
}

StringBuilder strBuilder = new StringBuilder("Some Text ");
for ( ... loop several times to build the string ...) {
strBuilder.Append(„ additional text „);
}
```

W przypadku manipulacji na zmiennymi łańcuchowymi, kiedy dokładnie znamy liczbę złożeń, możemy użyć operatora „+”. Budujemy jedno proste wyrażenie obejmujące wszystkie składowe ciągi:

```
String str = str1+str2+str3;
```

Jeżeli operacja konkatenacji wywoływana jest tylko w jednym wyrażeniu konieczne jest tylko jedno wywołanie **String.Concat**. Na skutek takiego działania nie są tworzone zmienne pomocnicze (temporary) dla kombinacji składanych łańcuchów.

W przypadku, kiedy z góry nie znamy ilości złożeń i zmiennych łańcuchowych do połączenia, należy zdecydować się na użycie **StringBuilder**.

```
for (int i=0; i< Results.Count; i++)
{
  StringBuilder.Append (Results[i]);
}
```

Klasa **StringBuilder** po inicjacji ma domyślnie wielkość ustawioną na 16. Łańcuchy mniejsze przechowywane są w obiekcie **StringBuilder**. Domyślna wielkość bufora może zostać powiększona:

```
constructor.
public StringBuilder (int capacity);
```

Pracę można kontynuować bez dodatkowej alokacji zasobów aż do momentu skonsumowania całej puli zasobów. W rezultacie obiekt **StringBuilder** jest często efektywniejszy niż użycie zwykłego typu **String**. Jeżeli operacja konkatencji przekroczy zaalokowany bufor, zostanie on automatycznie powiększony. Po przekroczeniu wielkości bufora 16 zostaje zaalokowany nowy bufor o rozmiarze 32, a zawartość starego zostaje skopiowana do nowo zaalokowanego bufora. Stary bufor zostaje zwolniony za pomocą garbage collectora.

Operator Null

Jest to jeden z bardzo przydatnych operatorów w językach programowania. Najczęściej wykorzystywany jest podczas sprawdzania wartości zmiennej, kiedy na przykład chcemy przypisać pewną wartość zmiennej typu **string**, ale jeśli zmienna jest pusta (**null**) i chcemy ją wyzerować. Oczywiście zadanie to moż-

na zrealizować na wiele sposobów, np. z wykorzystaniem operatora logicznego **if**:

```
string name = value;
if (value == null)
{
  name = string.Empty;
}
```

lub w następujący sposób:

```
string name = (value != null) ? value : string.Empty;
```

Zapis ten wydaje się być bardziej spójny, ale nie jest on optymalny. C# wniosło nowy, koalescencyjny operator (**??**):

```
string name = value ?? string.Empty;
```

Możliwe jest stworzenie metody **helpera**, aby mogła być wykorzystana z operatorem **??**.

```
public static class StringUtility
   {
          public static string TrimToNull(string source)
          {
                  return string.IsNullOrWhiteSpace(source) ? null : source.Trim();
  }
}
```

Zatem operator ten może być wykorzystany w następujący sposób:

```
string name = StringUtility.TrimToNull(value) ?? „None Specified”;
```

Alokacja na stosie i stercie

Niniejsza sekcja powinna rozwiać wszystkie wątpliwości na temat alokacji obiektów: czy wydajniejsze jest alokowanie obiektów na stosie (struktury) czy na stercie (klasy)? Przeanalizujemy czasy alokacji, inicjalizacji obiektów oraz wykonywania na nich operacji. Na początek stworzymy jedną klasę (typ referencyjny) i strukturę (typ wartości). Typy referencyjne alokowane są na stercie, a typy wartości na stosie. Podane niżej wyniki testów były przeprowadzone dla 100 000 obiektów:

```
for(int i=0;i<count;i++)
klasy[i]=new KlasaTestowa();
```

Rysuje się tutaj przewaga struktur nad klasami. Tworzenie obiektu klasy polega na utworzeniu na stosie zmiennej referencyjnej, zaalokowaniu na stercie pamięci na obiekt, przypisaniu zmiennej referencyjnej zaalokowanego obiektu. Natomiast tworzenie obiektu struktury polega na utworzeniu tego obiektu na stosie. Stąd znaczna różnica w czasach wykonania. Przeanalizujmy inicjalizację składowych obiektów:

```
for(int i=0;i<count;i++)
     klasy[i].zmienna=5;
```

Tu również struktury mają przewagę nad klasami. Wykonanie takiej operacji zajęło dla klas 24,94 ms, a dla struktur 10,62 ms. Ostatni test prezentuje przekazywanie obiektów jako argumentów funkcji.

```
for(int i=0;i<count;i++)
     OperacjaNaKlasie(klasy[i]);
```

Tutaj także szybciej zachowywały się struktury, ale nieznacznie (klasy 18,86 ms; struktury 14,04 ms). Wnioskiem podsumowującym jest zatem, że klasy przeznaczone są dla „dużych" obiektów, składających się z wielu zmiennych i długo wykorzystywanych, natomiast struktury do małych obiektów, które często są tworzone i niszczone.

Operator rzutowania AS

Za przykład nieoptymalnego zastosowanie operatora **as** tutaj posłuży poniższy algorytm:

```
if (employee is SalariedEmployee)
{
  var salEmp = (SalariedEmployee)employee;
  pay = salEmp.WeeklySalary;
}
```

Jest to przykład, w którym typ sprawdzany jest dwukrotnie. Po raz pierwszy przy użyciu **is**, po raz drugi przy rzutowaniu. Zdecydowanie lepszą praktyką jest zastosowanie operatora **as**:

```
var salEmployee = employee as SalariedEmployee;
if (salEmployee != null)
{
  pay = salEmployee.WeeklySalary;
}
```

Stosowanie „using”

Komenda **using** to jeden z najlepszych sposobów na zarządzanie zasobami, a uściślając – najlepszy sposób usuwania (**dispose**). Ta konstrukcja wprowadza efektywny blok try/finalny, gdzie referencja do obiektu użytego przez **using** jest zadeklarowana. Poniższe dwa przykłady są merytorycznie tożsame i prezentują tą samą funkcjonalność

```
MyClass obj = null;
try
{
     obj = new MyClass();
     obj.DoSomething();
}
finally
{
     if (null != obj)
          obj.Dispose();
}

using (MyClass obj = new MyClass())
{
     obj.DoSomething();
}
```

W obu przypadkach **obj** jest tworzony lokalnie. Na pierwszy rzut oka widać, że zastosowanie **using** wymaga mniejszej ilości tworzonego kodu i uwalnia nas od żmudnego procesu zwalniania zasobów obiektu. **Using** można używać w konstrukcjach zagnieżdżonych – podobnie jak **try/finally**. Poniższy przykład prezentuje użycie **DataReader** do wiązania z siatką ASP.NET.

```
protected DataGrid MyGrid;

private void Page_Load(object sender, System.EventArgs e)
{
     using (SqlConnection dbCxn = new SqlConnection("connection
string"))
     {

          dbCxn.Open();

          string sql = "SELECT * FROM MyTable";
          using (SqlCommand dbCmd = new SqlCommand(dbCxn, sql))
          {

               using (DataReader reader =
dbCmd.ExecuteReader(CommandBehavior.CloseConnection))
               {
                    MyGrid.DataSource = reader;
```

```
                    MyGrid.DataBind();
                }
            }
        }
    }
```

W tym przykładzie – w przypadku napotkania jakichkolwiek problemów podczas podłączania do bazy danych – siatka prezentacyjna nie zostanie podłączona. Wszystkie obiekty zostaną prawidłowo usunięte i zwolnione.

Pytaniem, na które należy odpowiedzieć jest, co z obiektami, które nie wspierają interfejsu **Idispose**? Niestety specyfika **using** wymaga, aby obiekty poprawnie impelnetowały i obsługiwały ten interfejs. Użycie **using** nie jest wymagane – ale z całą pewnością należy do najlepszych praktyk programistycznych i szczerze je polecam.

Dodawanie obiektów do kolekcji

Często w programach dodajemy wiele elementów do kolekcji. Często robimy to za pomocą funkcji **Add**() nie przykładając większej uwagi do funkcji **AddRange**(). Pierwsza funkcja jest prostsza w użyciu, jednak nie oznacza to, że jest bardziej wydajna. Funkcję **AddRange**() najczęściej wykorzystujemy, kiedy dodajemy tablicę do kolekcji:

```
ArrayList cols=new ArrayList(count);
Obiekt[] tab=new Obiekt[count];
for(int i=0;i<count;i++)
tab[i]=new Obiekt();
```

Najpierw tworzymy tablicę, którą następnie dodamy do kolekcji:

```
for(int i=0;i<count;i++)
     cols.Add(tab[i]);
cols.AddRange(tab);
```

Znacznie wydajniejsze jest dodawanie metodą **AddRange**(). Zauważmy także, że w przypadku **AddRange** od razu określana jest liczba elementów w kolekcji **cols**, dzięki czemu kod jest bardziej wydajny.

Zatem, jeśli możemy zapisać wartości, które chcemy dodać do kolekcji w tablicy, zróbmy to za pomocą funkcji **AddRange**(). Dodatkowo przy tworzeniu kolekcji ustawmy jej rozmiar na tyle elementów, ile planujemy przechowywać.

Kopiowanie kolekcji do tablicy

Najwydajniejsze w przypadku scenariusza kopiowania zawartości kolekcji do tablicy, będzie wykorzystanie metody **ICollection.CopyTo,** implementowanej przez wszystkie klasy kolekcji, albo metody ToArray implementowanej przez kolekcje **ArrayList, Stack** i **Queue**. Metody **ICollection**. **CopyTo** i **ToArray** wykonują w zasadzie tę samą funkcję: płytkie kopiowanie elementów zawartych w kolekcji do tablicy. Zasadniczą różnicę stanowi to, że **CopyTo** kopiuje elementy kolekcji do istniejącej tablicy, a **ToArray** tworzy nową tablicę.

Metoda **CopyTo** wykorzystuje dwa argumenty: tablicę i indeks. Tablica stanowi cel operacji kopiowania i musi być typu odpowiedniego dla elementów kolekcji. Jeśli typy nie są zgodne lub nie ma możliwości bezpośredniej konwersji typów elementów kolekcji na typy elementów tablicy, zostanie wywołany wyjątek **System.InvalidCastException**. Pierwszym elementem tablicy, do której elementy kolekcji zostaną skopiowane, jest indeks. Jeśli indeks jest równy lub większy niż długość tablicy – albo – jeśli liczba elementów kolekcji przekracza pojemność tablicy, zostanie wywołany wyjątek **System.ArgumentException**. Poniższy przykład ilustruje kopiowanie zawartości **ArrayList** do tablicy przy użyciu metody **CopyTo**.

```
ArrayList list = new ArrayList(3);
list.Add(“Raz”);
Ust.Add(”DWA”);
list.Add(“trzy”);
string[] array1 = new string[3];
list.CopyTo(arrayl,0);
```

Klasy **ArrayList, Stack**, i **Queue** również implementują metodę **ToArray**, która automatycznie tworzy tablicę odpowiedniego rozmiaru dla pomieszczenia kopii wszystkich elementów kolekcji. Po wywołaniu **ToArray** bez argumentów, zostanie zwrócony obiekt niezależnie od typu obiektów zawartych w tej kolekcji. Można przekazać obiekt **System.Type**, określający typ tablicy, którą metoda **ToArray** powinna utworzyć (trzeba będzie zmodyfikować zwróconą tablicę do uzyskania odpowiedniego typu). Poniższy przykład pokazuje wykorzystanie metody **ToArray** do listy **ArrayList**, której utworzenie pokazano w poprzednim przykładzie.

```
object[] array2 = list.ToArrayO;
string array3 = (string[])list.ToArray(System.Type.GetType(“System.String”));
```

Konwersja typów – najlepsze praktyki

W przypadku algorytmu, którego zadaniem będzie konwersja między typami danych, najlepiej zdecydować się na wykorzystanie metod *statycznych* (ang. *static*) klasy **System.BitConverter**. Dostarczają one wygodnego mechanizmu, który umożliwia konwersję większości podstawowych typów wartości w tablice bajtowe i odwrotnie (z wyjątkiem liczb dziesiętnych – **decimal**). Aby dokonać konwersji **decimal** na tablicę bajtową, należy zapisać **decimal** instancji **System.IO.MemoryStream**, wykorzystując obiekt **System. IO. BinaryWriter**, a następnie wywołać metodę **MemoryStream.ToArray**. Aby z tablicy bajtowej utworzyć **decimal**, należy utworzyć z niej obiekt **MemoryStream** i wczytać decimal z **MemoryStream**, posługując się instancją **System.IO.BinaryReader**.

Metoda statyczna (*static*) **GetBytes** klasy **BitConverter** pozwala na przeciążenie większości typów o wartości standardowej, które zwraca w postaci zakodowanej tablicy bajtowej. Obsługiwane są typy danych: **bool**, **char, double, short, int, long, float, ushort, uint** i **ulong**. **BitConverter** zapewnia również zestaw metod statycznych (*static*), umożliwiających konwersję tablic bajtowych na każdy ze standardowych typów wartości; noszą one nazwy **ToBoolean, ToUInt32, ToDouble**, itd. Poniższy kod demonstruje zastosowanie **BitConverter** do konwersji typów **bool** i **int** na tablicę bajtową i odwrotnie. Drugim argumentem dla każdej z metod **ToBoolean** i **ToInt32** jest przesunięcie przy *podstawie zerowej* (ang. *zero-basedoffset*) wewnątrz tablicy bajtowej, wskazujące miejsce, od którego **BitConverter** powinien rozpocząć przetwarzanie bajtów, aby uzyskać wartość danych.

```
byte[] b = nuli;
b = BitConverter.6etBytes(true); Console.WriteLine(BitConverter.ToString(b));
Console.WriteLine(BitConverter.ToBoolean(b,0));
b = BitConverter.GetBytes(3678); Console.WriteLine(BitConverter.ToString(b));
Console.WriteLine(BitConverter.ToInt32(b,0));
```

BitConterter nie zapewnia konwersji typu decimal. Można natomiast dokonać konwersji decimal na tablicę bajtową przy użyciu MemoryStream i BinaryWriter, co ilustruje poniższy kod:

```
public static byte[] DecimalToByteArray (decimal src) {
  using (MemoryStream stream = new MemoryStream()) {
          using (BinaryWriter writer = new BinaryWriter(stream)) {
                  writer.Write(src);
                  }
          }
  }
```

Pętle

Wiele pytań stawianych przez programistów dotyczy stosowania pętli **for** i **foreach**. Czym te pętle się różnią i jaka jest ich wydajność? Na pytania te odpowie prosty program, który wyświetla zawartość kolekcji

(wykorzystując obie pętle).

```
public static void TestFor(StringCollection cols)
{
     int count=cols.Count;
     string txt=string.Empty;
     for(int i=0;i<count;i++)
     {
          txt=cols[i];
     }
}
```

Zauważmy, że liczbę elementów w kolekcji pobieramy przed wejściem do pętli i zapamiętujemy w zmiennej **count**, dzięki czemu jeszcze bardziej optymalizujemy kod.

```
public static void TestForeach(StringCollection cols)
{
     string txt=string.Empty;
     foreach(string str in cols)
     {
          txt=str;
     }
}
```

Przy niewielkiej liczbie elementów wydajność obu pętli jest zbliżona, jednak przy większej liczbie jest wyraźnie zauważalna. Pętla **foreach** posiada dodatkowe ograniczenia (np. powinna być wykorzystywana tylko do odczytu elementów).

Wniosek: jeśli operujesz na wielu elementach, korzystaj z pętli **for**.

Rozdział IV. Podstawowe pojęcia związane z programowaniem wielowątkowym

Wprowadzenie

Jako pierwsze zdefiniujemy pojecie „procesu". Większość użytkowników platformy WIntel rozumie proces jako wykonywany program, który koegzystuje, dzieli zasoby *CPU* (ang. *Central Processing Unit* – CPU) oraz pamięci z innymi programami. W środowisku programistów proces znany jest jako wykonanie kodu programu w taki sposób, że każdy proces jest unikalny. Wykonanie każdego procesu odbywa się w izolacji. Zasoby używane przez proces (pamięć, dysk, urządzenia I/O. czas procesora) są wirtualizowane (mechanizm wirtualizacji wprowadza system operacyjny) tak, że każdy proces ma własny zbiór zasobów, których nie wykorzystują inne procesy. Proces wykonuje kolejne moduły kodu. Moduły te mogą być rozłączne, mogą być również współdzielone (np. wewnątrzprocesowe obiekty COM, które mogą być wywoływane z kontekstu dowolnego procesu).

Następnym pojęciem, które wymaga wyjaśnienia jest *wątek* (ang. *thread*). Wątki zostały stworzone w momencie, gdy pożądane stało się tworzenie aplikacji, które wykonywać miały kilka akcji jednocześnie. W przypadku, gdy jakieś akcje wywoływały opóźnienia w procesach jednowątkowych (np. oczekiwanie na reakcje użytkownika), często pożądane było, aby nadal wykonywane były jakieś akcje współbieżnie. Wątek to obiekt systemu operacyjnego, który reprezentuje wydzieloną część kodu w ramach procesu. Każda aplikacja Win32 ma *wątek główny* (ang. *primary thread*), który może tworzyć inne wątki, tzw. *wątki poboczne* (ang. *secondary threads*). Przykładem programu *wielowątkowego* (ang. *multithreading*) jest *powłoka systemu Windows* (ang. *windows shell*). Wystarczy kliknąć ikonę „Mój komputer" i otworzyć kilka okien. W jednym z nich wywołać np. proces kopiowania. Okno z aktywnym procesem kopiowania jest niedostępne, jednak pozostałe okna są gotowe do użycia.

Mechanizm wątków pozwala na jednoczesną realizację różnych funkcji aplikacji. Jednak równoczesne wykonywanie różnych komend jest tylko pozorem. Rzeczywistość przedstawia się inaczej. System po prostu szybko przełącza procesor pomiędzy poszczególne wątki, dzieje się to tak szybko, że sprawia wrażenie, iż wszystkie zadania wykonywane są jednocześnie.

Po co używać wątków ?

Wątki nie zmieniają semantyki programu – zmieniają tylko czas wykonywania operacji. Są one najczęściej wykorzystywane do rozwiązywania problemów powiązanych ze sobą. Oto najczęstsze przykłady wykorzystywania wielowątkowości:

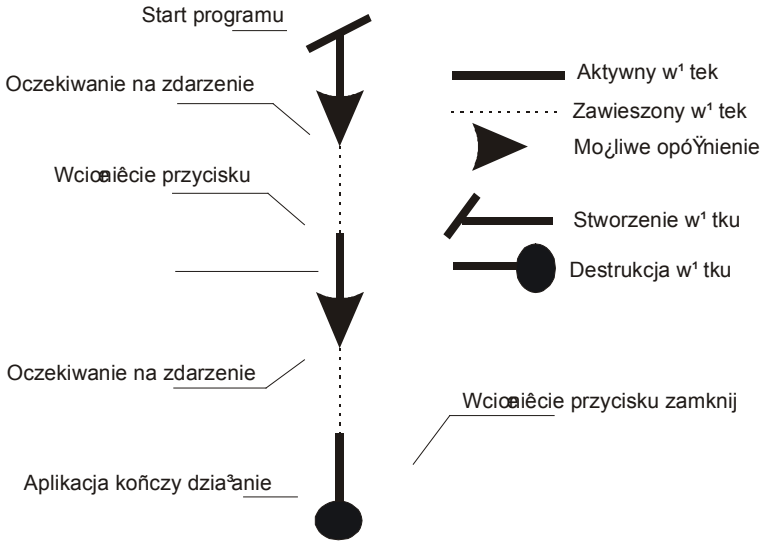
- podczas wykonywania długich obliczeń kiedy okno aplikacji nie odpowiada
- podczas wykonywania procesów w tle

- programowanie obiektów automatyzacji
- programowanie równoległe

Używanie wielowątkowości w aplikacji usprawni pracę programu tj. pozbędziemy się powstających często opóźnień. Spowoduje także prawidłowe wykorzystanie systemów wieloprocesorowych lub skalarnych, ponieważ aplikacja z jednym tylko wątkiem nie zrobi użytku z dwu czy więcej procesorów). Użycie wielowątkowości może usprawnić podział czasu procesora – szczególnie wtedy, gdy dokonamy w sposób właściwy ustawienia priorytetów wątków.

Obsługa wątków w środowisku Delphi

Wyobraźmy sobie następującą przykładową aplikację: wywoływany jest formularz z jednym przyciskiem, po naciśnięciu którego zostaje wyświetlony komunikat powitalny. Aplikacja ta ma tylko jeden wątek – **VCLthread**. Działanie tej aplikacji przedstawione jest na rys.1. Postęp wykonania aplikacji przedstawiony jest jako linia pionowa, czas płynie w dół.



Rys. 4 Diagram działania aplikacji

Diagram ten ilustruje porządek zdarzeń w czasie i stan wątku pomiędzy tymi zdarzeniami. Z diagramu tego wynika, że:

- Wątek tej aplikacji nie jest wykonywany ciągle tj. cały czas. Mogą wystąpić dość długie okresy, w których aplikacja nie otrzymuje żadnych „bodźców" i nie wykonuje w ogóle operacji. Zasoby zajęte przez aplikację są cały czas okupowane, ale kod nie jest wykonywany przez procesor.
- Aplikacja rozpoczyna swoje działanie i wykonuje główny wątek – wywołuje formularz (okno) aplikacji, a następnie w wyniku braku innych zadań przechodzi do fragmentu kodu w *VCL* (ang. *Visual Component Library*) zwanego *pętlą obsługi komunikatów* (ang. *message loop*). Żąda od systemu następnych *zdarzeń*. Gdy te się nie pojawiają, system operacyjny *zawiesza wątek* (ang. *suspending*

thread).

- W późniejszym czasie użytkownik naciska przycisk w celu wyświetlenia powitania. System budzi zawieszony wątek i przekazuje do niego zdarzenie wskazujące, że przycisk został wciśnięty. Wątek główny jest znów aktywny.
- Wzbudzenie procesu i jego zawieszenie może zdarzyć się jeszcze kilkukrotnie.

Wątek główny i wątek poboczny w środowisku Delphi

Win 32 API wprowadza bardzo wiele funkcji wspomagających wielowątkowość w aplikacjach. Delphi ma bardzo użyteczną i wygodną klasę TThread. Klasa ta ułatwia tworzenie i zarządzanie wątkami, poza tym pomaga uniknąć wielu pułapek związanych z wielowątkowością. Nowy wątek możemy utworzyć klikając w **menu: File->New**. Otworzy się okno dialogowe, z którego należy wybrać pozycję **ThreadObject**. Następnie zostaniemy zapytani o nazwę klasy pochodnej i po jej podaniu zostanie utworzony nowy moduł z wątkiem. Załóżmy, że klasa wątku będzie się nazywała **TSample**. Definicja klasy **TSample** w nowym module będzie wyglądała następująco:

```
Type
TSample = class(TThread)
private
protected
procedure Execute; override;
end;
```

Jak widać definicja wątku musi zawierać przynajmniej definicję metody **Execute**(), która jest wykorzystywana przy inicjowaniu wykonania wątku. Utworzenie nowego wątku i jego uruchomienie może nastąpić np. w wyniku zdarzenia wywołanego przez kliknięcie na przycisk. Oto przykład :

```
      uses Unit2; // moduł wątku
      procedure TForml.ButtonlClick(Sender : TObject);
var Nowy : TSample;
      begin
      Nowy := TSample.Create(False);
      end;
```

Tutaj ujawnia się ciekawa własność wątków - utworzenie nowego wątku nie musi być równoznaczne z jego wykonaniem. Wszystko to dzięki parametrowi konstruktora **TThread.Create**(), który może przybrać wartość **True** (wątek w stanie zawieszenia) lub **False** (automatyczne wykonanie). Jeżeli wątek jest w stanie zawieszenia, to jego uruchomienie nastąpi w momencie wywołania metody **TThread.Resume**(). W odwrotnej sytuacji, kiedy wątek jest uruchomiony wystarczy użyć metody **TThread.Suspend**() a nastąpi zawieszenie wątku. A jak zakończyć taki wątek? Należy tutaj wspomnieć, że zakończenie wykonywania instrukcji zawartych w procedurze **Execute**() jest równoznaczne z zakończeniem wątku. Jeżeli chcemy, by obiekt wątku

został automatycznie zwolniony, należy w procedurze **Execute**(), najlepiej na jej samym początku, umieścić następującą instrukcje przypisania:

```
FreeOnTerminate := True;
```

By zakończyć wątek natychmiast należy wywołać metodę **Terminate**. A oto przykład aplikacji mającej wątek główny (VCL) oraz wątek poboczny (Tthread). Zadaniem tej aplikacji jest określenie czy dana liczba jest liczbą pierwszą. Procedura uruchamiająca wątek poboczny (**Tpoboczny** klasy **Tthread**) z głównego wątku VCL ma postać:

```
procedure Tforma.Odpal(Sender: TObject);
var NewThread: Tpoboczny;
begin
 NewThread := Tpoboczny.Create(True);
 NewThread.FreeOnTerminate := True;
  try NewThread.Liczba := (liczba do testów};
      NewThread.Resume;
  except on EConvertError do begin
 NewThread.Free;
 ShowMessage('wprowadź liczbę');
  end; end; end;
```

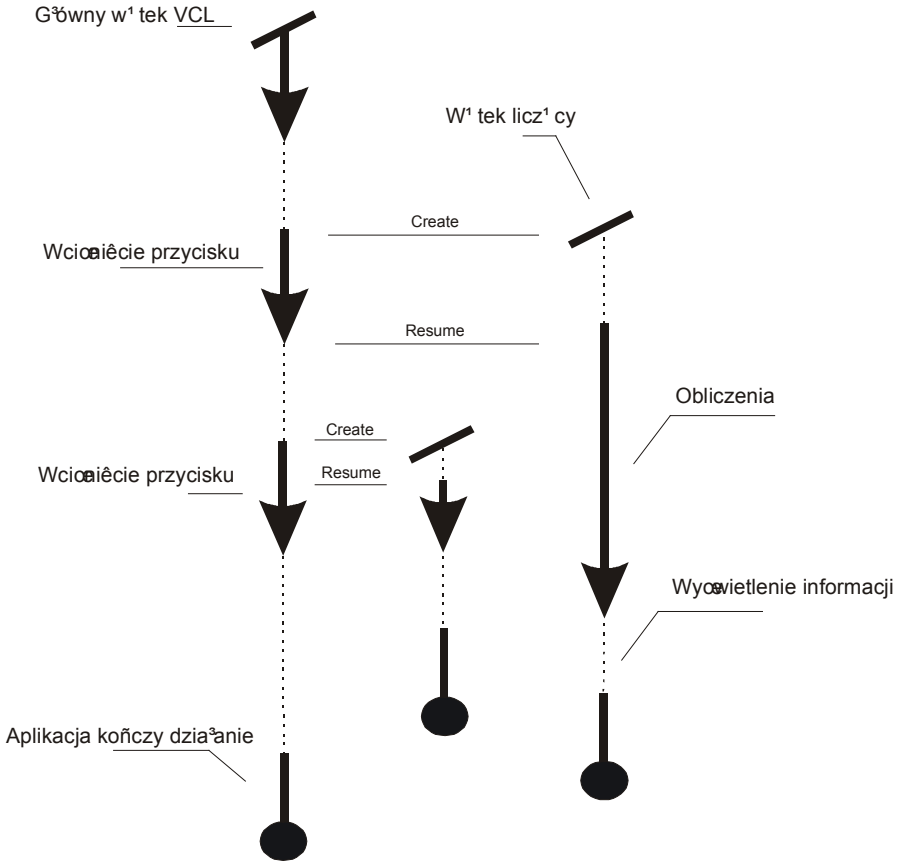
A oto jak wygląda wątek poboczny sprawdzający czy dowolna liczba naturalna jest liczbą pierwszą:

```
interface
uses Classes;
type TPoboczny = class(TThread)
private Fliczba: integer;
protected function CzyPierwsza: boolean;
procedurę Execute; override;
public property Liczba: integer write Fliczba; end;
implementation

uses SysUtils, Dialogs;
function Tpoboczny.CzyPierwsza: boolean;
var iter: integer;
begin
  result := true:
  if FLiczba <= 2 then exit;
  for iter :- 2 to Fliczba - l do begin
      if (FTestNumber mod iter) = O then begin
         result := false;
      end
  end;
end;

procedure Tpoboczny.Execute;
begin
  if Czypierwsza then ShowMessage(IntToStr(Fliczba) + 'jest liczbą
pierwszą.') else
```

```
    ShowMessage (IntToStr(Fliczba) + 'nie jest');
end;
```

Za każdym razem, gdy wywoływana jest procedura Odpal, wywoływany jest nowy wątek.

Inicjalizowane jest jego pole **Fliczba**. W zależności od danych wejściowych, wątek zaczyna sprawdzać czy podana liczba jest pierwszą. Po obliczeniu wyświetla informujący o tym komunikat. Zauważmy, że te wątki są współbieżne, niezależnie od tego czy aplikacja zostanie wykonana na maszynie jedno czy wieloprocesorowej. Dodatkową zaletą tej aplikacji jest fakt, że nie limituje ona liczby stworzonych wątków. Czas wykonywania wątku pobocznego zależny jest jedynie od wielkości liczby, dla której przeprowadzane są obliczenia (jeśli np. najpierw badamy dużą liczbę, powiedzmy 1.000.000 i zaraz po niej zbadamy liczbę 2, najpierw zostanie wyświetlony wynik dla liczby mniejszej). Na rys. 2 przedstawiono przebieg sytuacji:



Rys. 5 Schemat pracy aplikacji dla liczb 1.000.000 I 2

Niespodzianki oraz problemy związane z wątkami

W tym miejscu po raz pierwszy hydra synchronizacji międzywątkowej podnosi swój łeb. Otóż, odkąd wątek główny wskrzesił wątek poboczny, nie może on nic powiedzieć o stanie tego wątku (i *vice versa*). Taki stan rzeczy prowadzi do powstawania potencjalnie niebezpiecznych sytuacji trzech rodzajów:

- Kłopotliwy start. Środowisko Delphi doskonale radzi sobie z tworzeniem wątków. Przed wykonaniem wątku pobocznego, wątek rodzic często chce dokonać ustawień pewnych stanów, pól wątku

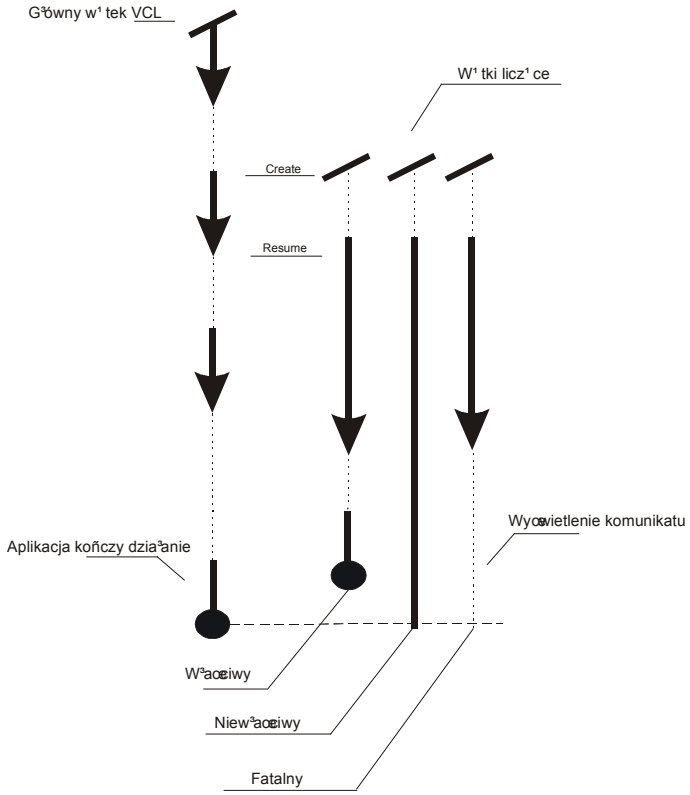
potomnego. Poprzez stworzenie wątku zawieszonego mamy pewność, że kod wątku nie zostanie wykonany dopóki nie dokonamy wszystkich ustawień (w tym przykładzie było to ustawienie własności **FreeOnTerminate** oraz pola liczba).

- Komunikacja i problemy z nią związane. Powstają wtedy, gdy mamy dwa (lub więcej) wykonywane wątki i chcemy, aby komunikowały się one między sobą. Jeżeli nie mamy stworzonej odpowiedniej synchronizacji, czy kontroli współbieżności, możemy być pewni, że nie dane nam będzie doświadczyć współdzielenia zasobów między wątkami ani wykonywać operacji graficznych w osobnych wątkach. Jednak twórcy oprogramowani z Redmont wspaniałomyślnie wyposażyli systemy Windows w szereg mechanizmów umożliwiających nam komunikowanie się między wątkami.

- Niszczenie wątków. Wątki, podobnie jak inne obiekty środowiska Delphi, zajmują pamięć i zasoby systemowe, więc bardzo istotne jest rozsądne ich niszczenie. Generalnie są możliwe dwa podejścia do tego zagadnienia:

o Pozwolić wątkowi zająć się tym samodzielnie. To podejście stosowane jest głównie do dwóch rodzajów wątków – takich, które zwracają do wątku rodzica określone dane jeszcze przed zniszczeniem i takich, które nie mają żadnych ważnych dla innych procesów danych w czasie niszczenia. W powyższych przypadkach możemy ustawić własność **FreeOnTerminate** na **True**. Spowoduje to, iż wątek „sam po sobie posprząta".

o W drugim sposobie wątek rodzic czyta dane z wątku dziecka, a później niczym wyrodna matka pozbywa się dziecka i zwalnia wszystkie zajmowane przez nie zasoby.



Rys. 6 Scenariusze działania wątków pobocznych aplikacji

Rysunek 6 ilustruje, jak fatalne w skutkach może być podejście pierwsze. Widać, że mogą się tu wydarzyć dwie niepożądane sytuacje. Pierwsza: gdy próbujemy zamknąć program, podczas gdy jeden z wątków jeszcze prowadzi obliczenia. Druga: gdy próbujemy zamknąć program podczas, gdy jeden z wątków jest zawieszony. Pierwsza sytuacja nie jest jeszcze zagrożeniem dla stabilności systemu – po prostu aplikacja zakończy swoje działanie bez konsultowania się z wątkiem pobocznym. Delphi oraz Windows „posprzątają" po nim. Jednak poważny kłopot sprawia drugi przypadek. Wątek spoczywa gdzieś zawieszony „wewnątrz" podsystemu komunikatów. Jednak z własnego doświadczenia autora niniejszej pracy wynika, że środowisko Delphi znakomicie radzi sobie i w tym przypadku, jakkolwiek pozostawiania wątków takim stanie powinno się unikać.

Atomowość w stosunku do danych współdzielonych

W celu zrozumienia, jak wątki współpracują ze sobą należy wyjaśnić pojęcie atomowości. Akcja lub sekwencja akcji jest atomowa wtedy, gdy jest niepodzielna. Kiedy wątek przygotowuje akcję atomową oznacza to, że dla innych wątków akcja ta widziana jest jako jeszcze nie rozpoczęta lub jako wykonana. Nie jest możliwe, aby jeden wątek „przyłapał" inny w trakcie takiej akcji. Rozważmy następujący fragment kodu:

```
var
zmienna:integer //zmienna globalna
begin
zmienna:= zmienna + 1;
end;
```

Niestety nawet tak trywialny fragment może doprowadzić do fatalnych w skutkach następstw podczas modyfikowania zmiennej przez osobne wątki. Wyrażenie to rozkłada się na trzy operacje na poziomie asemblera:

- skopiowanie zawartości zmiennej z pamięci do rejestru procesora
- dodanie wartości 1 do rejestru procesora
- przepisanie zawartości rejestru do zmiennej w pamięci

Na komputerze jednoprocesorowym wykonanie tego kodu przez kilka wątków może spowodować problemy. Przyczyną tego jest *planowanie* (ang. *scheduling*) operacji. W komputerach maszynach jednoprocesorowych w danym momencie może być aktywny tylko jeden wątek. Mechanizm zwany **Win32 Scheduler** zajmuje się przełączaniem między nimi (około 18 razy na sekundę). *Program szeregujący* (ang. *Scheduler*) może zatrzymać działanie jednego wątku lub uruchomić inny. System operacyjny nie czeka na pozwolenie zawieszenia pacy jednego wątku i wykonanie innego, przełączenie następuje w dowolnych chwilach czasowych. Ponieważ przełączenie może nastąpić między dwiema dowolnymi instrukcjami procesora, może

ono nastąpić w miejscu „strategicznym” programu. Wyobraźmy sobie dwa wątki W1 i W2 wykonywane w komputerze jednoprocesorowym. Istnieje prawdopodobieństwo, że program bezie się wykonywał poprawnie a proces przełączenia ominie punkty krytyczne programu. Schemat takiej sytuacji wyglądałby następująco:

Instrukcje wykonywane przez W1	**Instrukcje wykonywane przez W2**	**Wartość zmiennej**
<inne operacje>	Wątek zawieszony	1
Przeczytanie zmiennej z pamięci do rejestru procesora	Wątek zawieszony	1
dodanie wartości 1 do rejestru procesora	Wątek zawieszony	1
Przepisanie zawartości rejestru do zmiennej w pamięci	Wątek zawieszony	2
<inne operacje>	Wątek zawieszony	2
Przełączenie wątków	**Przełączenie wątków**	2
Wątek zawieszony	<inne operacje>	2
Wątek zawieszony	Przeczytanie zmiennej z pamięci do rejestru procesora	2
Wątek zawieszony	dodanie wartości 1 do rejestru procesora	2
Wątek zawieszony	Przepisanie zawartości rejestru do zmiennej w pamięci	3
Wątek zawieszony	<inne operacje>	3

Tabela 8 Optymistyczny scenariusz działania wątków W1 i W2 w komputerze jednoprocesorowym

Niestety istnieje wysokie prawdopodobieństwo, że program ten wykona się w następujący sposób:

Instrukcje wykonywane przez W1	**Instrukcje wykonywane przez W2**	**Wartość zmiennej**
<inne operacje>	Thread Suspended	1
Przeczytanie zmiennej z pamięci do rejestru procesora	Thread Suspended	1
dodanie wartości 1 do rejestru procesora	Thread Suspended	1
Przełączenie wątków	**Przełączenie wątków**	**1**
Wątek zawieszony	<inne operacje>	1
Wątek zawieszony	Przeczytanie zmiennej z pamięci do rejestru procesora	1
Wątek zawieszony	dodanie wartości 1 do rejestru procesora	1

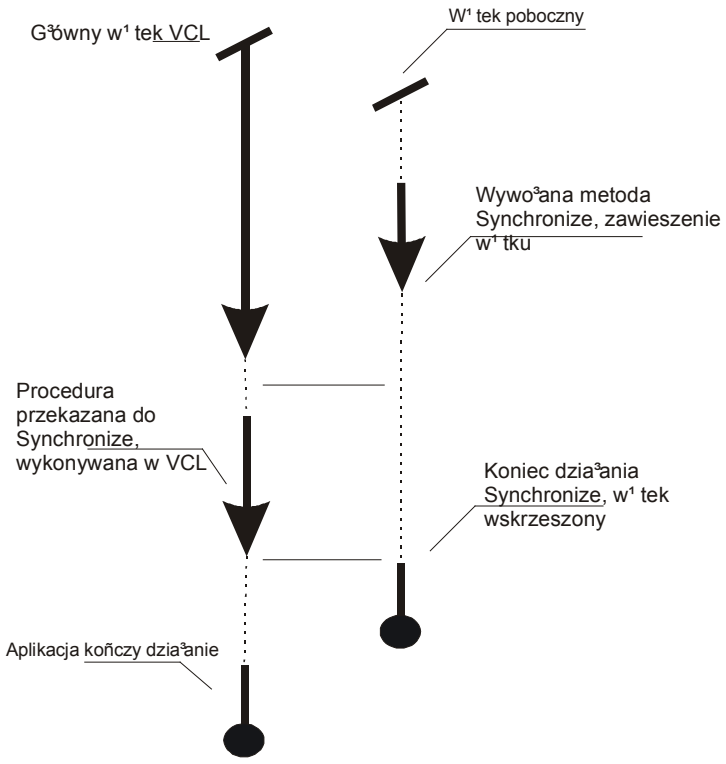
Wątek zawieszony	Przepisanie zawartości rejestru do zmiennej w pamięci	2
Przełączenie wątków	**Przełączenie wątków**	**2**
Przepisanie zawartości rejestru do zmiennej w pamięci	Wątek zawieszony	2
<inne operacje>	Wątek zawieszony	2

Tabela 9 Pesymistyczny scenariusz działania wątków W1 i W2 w komputerze jednoprocesorowym

Sytuacja taka nosi nazwę *warunkowego wyścigu* (ang. *race condition*[1]). Wątek VCL nie ma zabezpieczeń przed tego typu pułapkami. Przełączanie między wątkami może doprowadzić do różnego rodzaju sytuacji błędnych (naruszenie więzi miedzy komponentami, uszkodzenie licznika odwołań itp). Problemy tego typu nie występują na maszynach wieloprocesorowych.

Metoda Synchronize klasy TThread

Metoda **Synchronize** jest rozwiązaniem proponowanym przez twórców środowiska Delphi. Metoda ta rozwiązuje wszystkie wspomniane do tej pory problemy. W metodzie tej przyjmuje się jako parametr bezparametrową metodę, która jest wykonywana. Mamy pewność, że kod tej metody zostanie wywołany jako rezultat wywołania metody **Synchronize**. Metoda ta daje nam również pewność, że nie nastąpi żaden konflikt z głównym wątkiem VCL. Metoda wywoływana wtedy, gdy używany jest mechanizm synchronizacji potrafi „zrobić wszystko" to co wątek VCL. Dodatkowo może ona modyfikować dane własnego wątku. Na poniższym diagramie zilustrowano co się dzieje później:



Rys. 7 Ilustracja działania metody Synchronize

1 W literaturze spotkać można również tłumaczenie „niebezpieczeństwo wyścigu"

Kiedy wywoływana jest metoda **Synchronize**, wątek poboczny jest zawieszany. Teraz następuje oczekiwanie, aż wątek główny VCL znajdzie się w stanie całkowitej *jałowości* (ang. *idle*). Kiedy już VCL osiągnie stan jałowości, bezparametrowa metoda przekazana jako parametr **Synchronize**, zostaje wykonana w kontekście wątku głównego VCL. To daje pewność, że nie zdarzy się żaden konflikt z VCL. Żaden konflikt nie wystąpi także w wątku pobocznym, gdyż jest ona zawieszony w bezpiecznym miejscu. Kiedy już metoda zostanie wykonana, wątek główny jest wolny i może zająć się swoją normalną pracą. Inaczej sprawa przedstawia się z synchronizacją wątków pobocznych. Załóżmy, że mamy dwa wątki poboczne W1 i W2. Konieczne jest wywołanie metody **Synchronize** z obu wątków podczas dostępu do danych współdzielonych, w efekcie czego modyfikowaniem danych zajmuje się wątek główny, a wątki poboczne synchronizują się z nim, kiedy chcą odczytać lub zapisać dane współdzielone. To podejście nie jest jednak zbytnio efektywne.

Kończenie, przerywanie i niszczeniem wątków

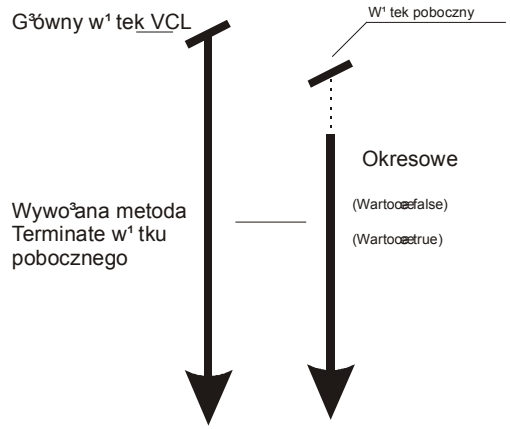
Oto dwie główne sprawy, na które należy zwrócić uwagę rozważając kwestie związane z kończeniem, przerywaniem i niszczeniem wątków:

- „czyste" wychodzenie z wątku i uprzątnięcie zasobów zajmowanych przez wątek
- odczyt wartości z wątku po jego zakończeniu

Jeżeli wątek poboczny nie zwraca do wątku głównego żadnych danych po zakończeniu swej pracy, lub jeśli używa technik komunikacji po zakończeniu pracy opisanych wcześniej, nie istnieje potrzeba, aby angażować wątek główny w „sprzątanie" po zakończonym wątku pobocznym. Wystarczy ustawić tylko własność **FreeOn Termiante** na wartość **True** i pozwolić, aby wątek poboczny sam zajął się delokacją swoich zasobów. Rozwiązanie takie jest akceptowalne jedynie w przypadku, gdy wątek operuje tylko na danych w pamięci operacyjnej. W przypadku, gdy zapisuje on dane do pliku lub innego urządzenia I/O, taka sytuacja jest niedopuszczalna.

Przygotowanie przerwania działania wątku

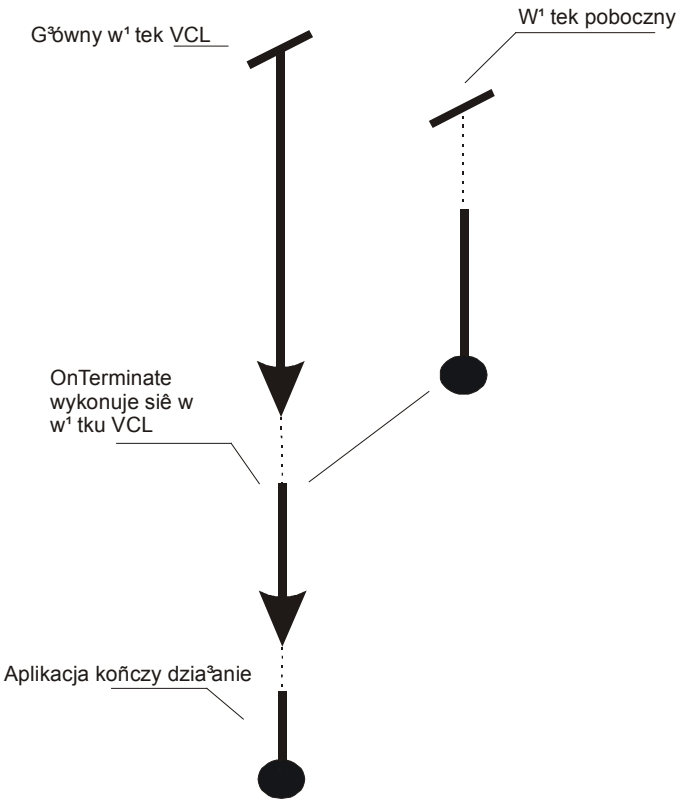
Istnieją sytuacje, w których jeden wątek musi powiadomić drugi, że ten powinien zakończyć pracę. Z taką sytuacją mamy często do czynienia wtedy, gdy wątek przygotowuje długie obliczenia, a użytkownik decyduje o tym czy należy zamknąć aplikację lub zakończyć wątek. Klasa **TThread** wprowadza mechanizm do obsługi takich sytuacji – metodę **Terminate** i własność **Terminated**. Kiedy wątek jest tworzony jego właściwość **Terminated** ma wartość **False**. Kiedy wywoływana jest procedura **Terminate**, wartość tej własności ustawiana jest na **True**. Dzieje się tak, ponieważ wątki co pewien czas sprawdzają wartość tej własności. Ustawienie tej własności należy interpretować jako „zakończ wątek najszybciej, jak to możliwe". Na rys. 5 przedstawiono tą sytuację.



Rys. 8 Działanie metody Termianate

Zdarzenie OnTerminate

Występuje zawsze wtedy, gdy wątek rzeczywiście kończy działanie. Zdarzenie to nie następuje w momencie, gdy wywoływana jest metoda **Terminate**. Zdarzenie to jest bardzo użyteczne, dlatego jest wywoływane w kontekście wątku głównego tak jak metoda wywołana w procedurze **Synchronize**. Dlatego jest to właściwe miejsce na umieszczenie operacji związanych z zakończeniem działania wątku. Jest to również doskonałe miejsce do przeprowadzenia transferu danych z kończącego pracę wątku pobocznego do wątku głównego VCL.



Rys. 9 Wykorzystanie zdarzenia OnTerminate

Jak widać z rysunku 9 zdarzenie **OnTerminate** działa podobnie jak metoda **Synchronize** i jest prawie identyczne semantycznie, jak użycie metody **Synchronize** na końcu pracy wątku. Najczęściej

spotykanym zastosowaniem tego zdarzenia jest stworzenie w procedurze obsługi licznika aktywnych wątków aplikacji (zamknij, jeśli tylko VCL jest aktywny).

Metoda WaitFor

Wspomniane wyżej zdarzenie jest użyteczne jedynie w przypadku wątków, które automatycznie realizują proces „sprzątania po sobie". A co w przypadku, kiedy musimy mieć pewność, że wszystkie wątki poboczne są nieaktywne? Odpowiedzią na to pytanie jest właśnie metoda **WaitFor**. Metoda te jest użyteczna wtedy, gdy:

- wątek główny VCL musi komunikować się z wątkiem pobocznym po tym, jak zakończył on swoje działanie aby modyfikować jego dane
- trzeba spowodować przerwanie działania w momencie, gdy opcja zamknięcia aplikacji jest niedostępna

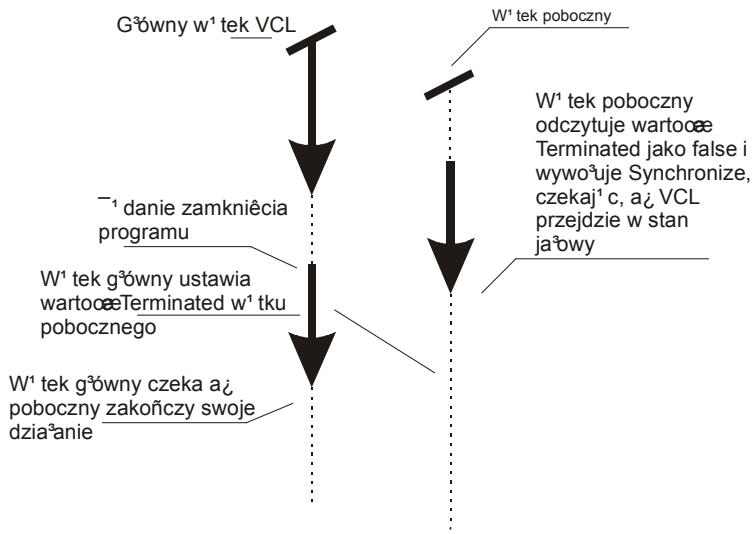
W praktyce, kiedy wątek W1 wywołuje metodę **WaitFor** wątku W2, wątek W1 zawiesza swoje działanie, dopóki wątek W2 się nie wykona. Kiedy wątek W1 wznowi działanie, jest pewne, że można czytać wyniki z wątku W2 a obiekt **Tthread** reprezentujący W2 może zostać zniszczony.

Główna zaleta metody **WaitFor** jest jednak równocześnie jej największą wadą: zawiesza ona działanie wątku głównego tak, że ten nie może odbierać komunikatów. Oznacza to, że aplikacja nie może wykonywać operacji związanych z obsługą pętli komunikatów (aplikacja nie przerysowuje okna, nie można zmienić rozmiarów formularza, aplikacja nie reaguje na zewnętrzne czynniki). Łudząco przypomina to sytuację, w której dany program nie odpowiada („zawiesza się"). Z sytuacją tą mamy do czynienia jedynie w przypadku przerywania egzekucji wątku w sposób anormalny. Czas, który główny wątek spędza w tym stanie zależy głównie od tego, jak często wątek pracownika sprawdza swoja właściwość **Terminated**.

Zjawisko zakleszczenia

W aplikacjach, w których użyto metod **WaitFor** oraz **Synchronize** istnieje duże prawdopodobieństwo wystąpienia zjawiska *zakleszczenia* (ang. *deadlock*). Zakleszczenie to fenomen, przez który w aplikacji nie występują żadne błędy algorytmiczne, jednak proces wykonania programu zostaje wstrzymany. Zjawisko to występuje, gdy wątki oczekują na siebie w sposób cykliczny. Na przykład wątek W1 czeka aż W2 zakończy swoje działanie, podczas gdy W3 czeka na W4. Na końcu wątek W4 oczekuje na wątek W1. Niestety W1 nie może wykonać wszystkich operacji, bo jest zawieszony. W niektórych przypadkach zakleszczenie może nastąpić między dwoma wątkami, jeśli wątek poboczny wywołuje metodę **Synchronize** na krótko przed tym jak wątek główny wywoła metodę **WaitFor**. Wtedy wątek poboczny będzie czekał na wątek główny, aż ten powróci do pętli obsługiwania komunikatów, natomiast wątek główny czeka, aż wątek roboczy zakończy

obliczenia. Rezultatem takiej sytuacji jest zjawisko zakleszczenia. Jest też prawdopodobne, że główny wątek VCL wywoła metodę **WaitFor** na krótko przed tym, jak wątek poboczny wywoła metodę **Synchronize**. Taka sytuacja również może doprowadzić do zakleszczenia.



Rys. 10 Zjawisko zakleszczenia

Najlepszym sposobem na uniknięcie pułapek związanych z zakleszczeniem, jest po prostu nie używanie w jednej aplikacji metod **Synchronize** i **WaitFor**. Metoda **WaitFor** może więc być zastąpiona obsługą zdarzenia **OnTerminate**.

Ograniczenia synchronizacji

Metoda **Synchronize** ma kilka wad, które czynią ją nieodpowiednią za wyjątkiem stosowania w naprawdę prostych aplikacjach wielowątkowych:

- metoda ta użyteczna jest tylko w przypadku komunikacji wątku pobocznego z wątkiem głównym VCL
- metoda **Synchronize** wymaga, aby wątek poboczny czekał aż wątek główny VCL znajdzie się w stanie całkowicie jałowym
- jeśli aplikacja często używa tej metody, to wątek VCL staje się „wąskim gardłem” aplikacji i program zamiast usprawnić swoje działanie dzięki zastosowaniu wielowątkowości osiąga odwrotny skutek
- jeśli wywołujemy tą metodę do komunikacji między dwoma wątkami pobocznymi, oba wątki mogą być zawieszane oczekując na wątek główny VCL
- metoda ta może doprowadzić do zakleszczenia.

Niezmiernie ważne jest to, aby pamiętać „po co" używamy wątków w aplikacji. Najczęstszym powodem stosowania wątków pobocznych w aplikacji jest chęć, aby aplikacja pozostała „czujna" na bodźce zewnętrzne podczas wykonywania długich obliczeń lub blokowania dostępu do urządzeń I/O oraz przyspieszenie działania aplikacji. Oznacza to, że wątek główny VCL powinien wykonywać jedynie krótkie fragmenty obliczeń, oraz zająć się komunikacją z użytkownikiem. Reszta kodu zajmującego się „głównymi obliczeniami" powinna znaleźć się w watkach pobocznych.

Wiele wątków komunikuje się z VCL tylko w prosty sposób tj. przekazując strumienie danych, bądź wykonując zapytanie na bazie danych i zwracając wyniki. Wróćmy na chwilę do pojęcia atomowości. Wyobraźmy sobie, że mamy strumień, do którego dane zapisywane są przez wątek poboczny; dane te co pewien czas odczytywane są przez wątek główny. Pytanie brzmi: czy musimy mieć pewność, że wątek główny i wątek poboczny nie są uruchamiane w tym samym czasie? Odpowiedź brzmi NIE! Wszystko czego potrzebujemy, to upewnienie się, że tylko jeden wątek w określonym czasie modyfikuje dane współdzielone – to czyni operację atomową. Własność ta zwana jest **wzajemnym wykluczaniem**. Jest kilka podstawowych prymitywów synchronizacji służących do zapewnienia tej własności. Najprostszym z nich jest **Mutex**. System Windows oferuje wiele procedur związanych z mutexami – m.in. *sekcję krytyczną* (ang. *critical section*). Firma Borland od wersji 4 wyposażyła System Delphi w klasę zawierającą sekcje krytyczne.

Mechanizmy i obiekty wykorzystywane do synchronizacji

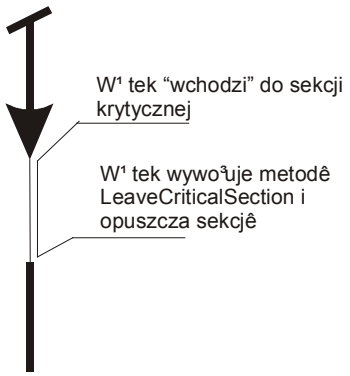
Sekcje krytyczne

Mechanizm ten daje nam możliwość zapewnienia wzajemnego wykluczania się. W kontekście sekcji krytycznej mamy następujące operacje:

- *InitializeCriticalSection*
- *DeleteCriticalSection*
- *EnterCriticalSection*
- *LeaveCriticalSection*
- *TryEnterCriticalSection* (operacja obsługiwana tylko pod systemami z jądrem NT)

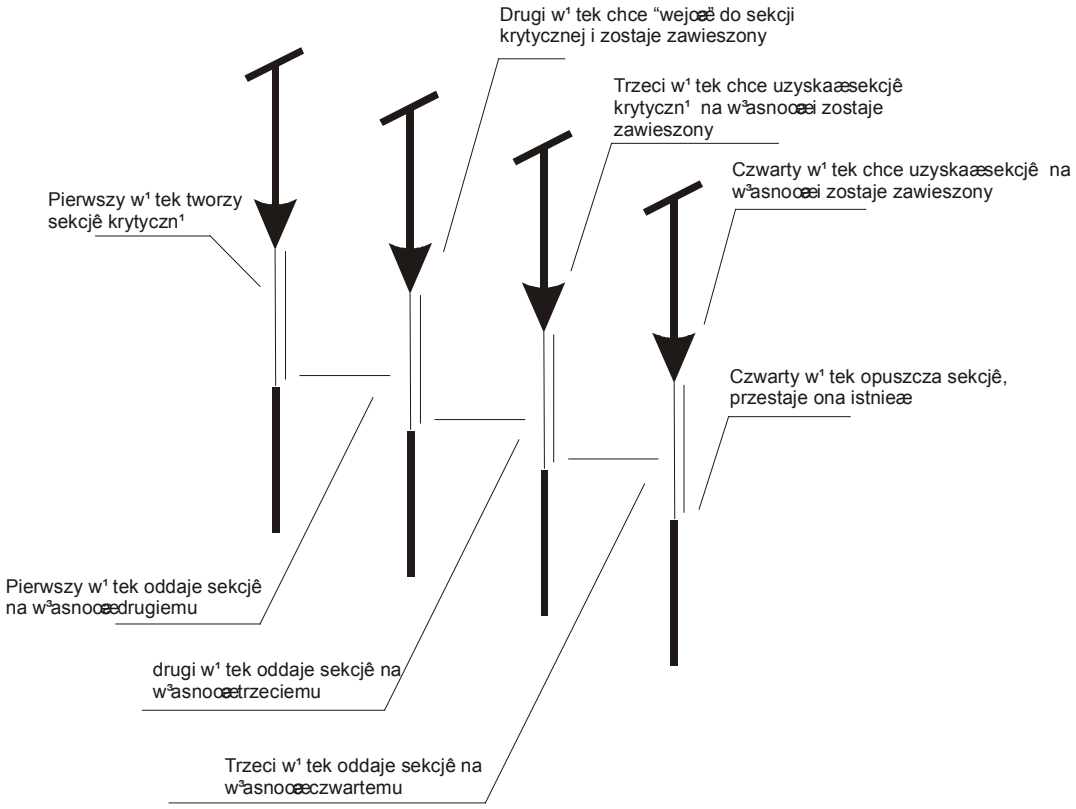
Utworzenie sekcji (**InitializeCriticalSection**) oraz jej usunięcie (**DeleteCriticalSection**) są analogiczne do tworzenia oraz destrukcji obiektu. Sensownym rozwiązaniem jest stworzenie i usunięcie sekcji krytycznej w jednym wątku, zazwyczaj najdłużej „żyjącym". Oczywiście każdy wątek, który chce uzyskać synchronizowany dostęp do danych musi mieć wskaźnik lub uchwyt tego obiektu. Raz stworzony obiekt sekcji krytycznej może być używany do kontroli dostępu do danych współdzielonych. Głównymi operacjami wykonywanymi na sekcji są **EnterCriticalSection** oraz **LeaveCriticalSection** (w literaturze zdarza się też użycie pojęć WAIT i SIGNAL lub LOCK i UNLOCK). Na początku po stworzeniu sekcji krytycznej żaden z wątków nie ma jej na własność. Aby uzyskać sekcje na własność, wątek musi wywołać operację **Enter-**

CriticalSection. W przypadku, gdy sekcja nie jest w posiadaniu innego wątku, otrzymuje do niej prawo własności. Po otrzymaniu prawa własności wątek właściciel wykonuje operacje na zasobach współdzielonych, po czym opuszcza sekcję wywołując **LeaveCriticalSection**.



Rys. 11 Użycie sekcji krytycznej

Rysunek 11 ilustruje opisaną wcześniej sytuację. Ważną własnością sekcji krytycznej jest fakt, że tylko jeden wątek w danym czasie może w niej przebywać. W przypadku, kiedy wątek próbuje wejść do sekcji, która jest wykonywana przez inny wątek, zostanie zawieszony i wskrzeszony dopiero wtedy, gdy wątek przebywający w sekcji krytycznej ją opuści. To zapewnia wzajemne wykluczanie wątków w dostępie do danych współdzielonych. W oczekiwaniu na wejście do sekcji krytycznej zawieszonych może zostać więcej wątków, więc jak widać operacja **EnterCriticalSection** może być wykorzystywana do synchronizacji pracy kilku wątków. Na rysunku 9 przedstawiono sytuację, gdy kilka wątków próbuje jednocześnie dostać się do sekcji krytycznej.



Rys. 12 Kilka wątków próbujących jednocześnie dostać się do sekcji krytycznej

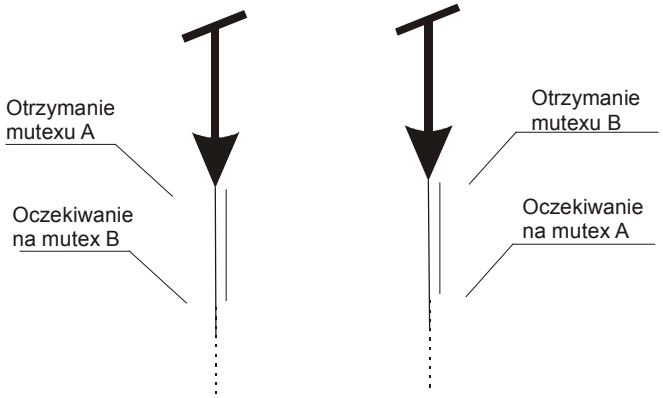
Jak widać na rysunku tylko jeden wątek może wykonywać sekcję w danym czasie, Problem wyścigu warunkowego i wzajemnego wykluczania został rozwiązany. Jakie korzyści daje to programistom w środowisku Delphi? Otóż główny wątek VCL nie musi znajdować się w stanie jałowym, aby wątek poboczny mógł zmodyfikować dane współdzielone. Wątek wykonujący się w sekcji krytycznej „nie wie” i nie zajmuje się tym jaki wątek obecnie ją wykonuje (nie ma rozróżnienia na wątek główny i obiekty **TThread**). Można używać metody **WaitFor** (prawie) bez obawy wystąpienia zakleszczenia. Aby uniknąć tego niepożądanego zjawiska wystarczy wywołać tą metodę z wątku głównego, gdy jest on akurat wewnątrz sekcji krytycznej.

Obiekt Mutex

Mutexy pracują dokładnie w ten sam sposób jak sekcje krytyczne. Jedyna różnica to ta, że użycie sekcji krytycznych ogranicza się jedynie do jednego programu. W przypadku jednego programu używającego kilku wątków, sekcja krytyczna jest obiektem optymalnym. Jeśli jednak chodzi o tworzenie aplikacji rozproszonych opartych na technologii COM należy, używać mutexów. Dlatego Win32 API wprowadza większy zakres funkcji obsługujących mutexy. Oto kilka, najważniejszych z naszego punktu widzenia, funkcji związanych z obsługą tego obiektu:

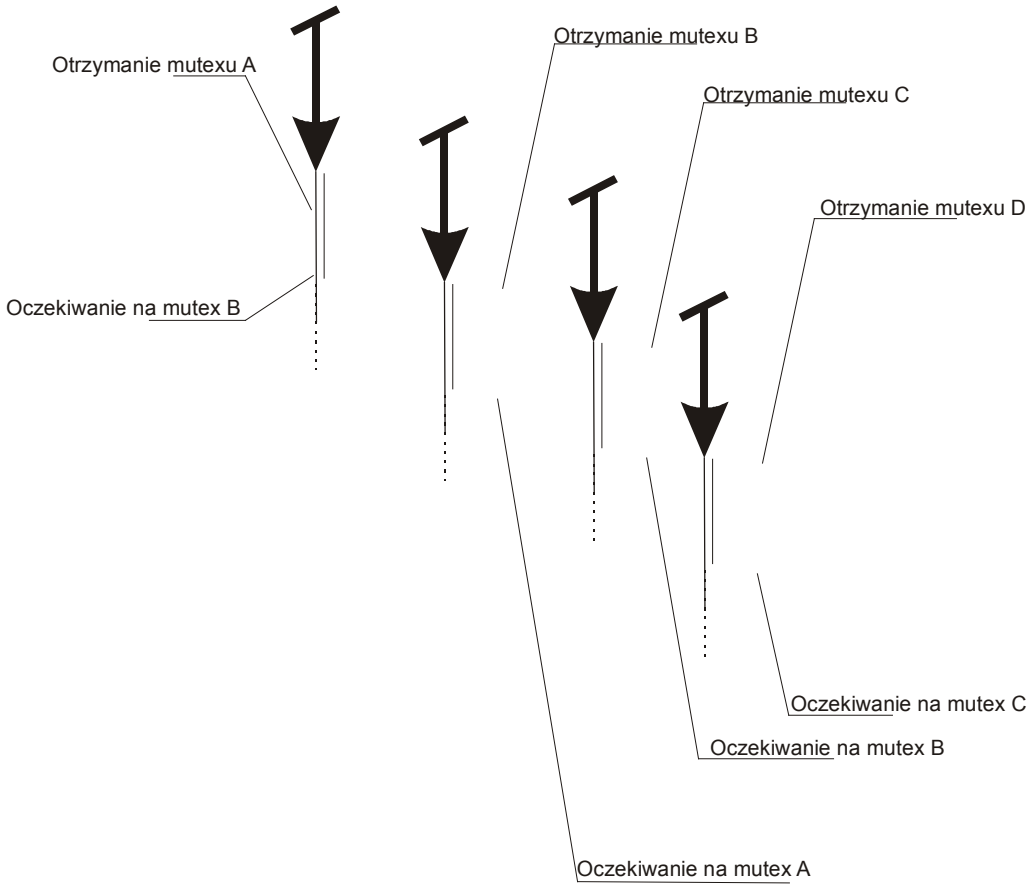
- *CreateMutex / OpenMutex*
- *CloseHandle*
- *WaitForSingleObject(eX)*
- *ReleaseMutex*

W przypadku, gdy program ma więcej niż jeden mutex, istnieje prawdopodobieństwo wystąpienia zjawiska zakleszczenia. Najczęściej spotykaną sytuacją jest zależność cykliczna w kolejce wątków oczekujących na mutex. W literaturze zjawisko wzajemnego wykluczania rozwiązano za pomocą mutexów na podstawie ucztujących filozofów. Jak już wcześniej wspominano zakleszczenie występuje wtedy, gdy wątki oczekują aż inny wątek zwolni obiekt synchronizacji. Najprostszą ilustracją tego problemu jest sytuacja między dwoma watkami: jeden z nich uzyskuje mutex A przed uzyskaniem mutexu B, drugi najpierw pozyskuje mutex B, a potem A.



Rys. 13 Zjawisko zakleszczenia wywołane nieprawidłowym użyciem obiektów Mutex (wersja I)

Oczywiście istnieje też inny, bardziej złożony sposób na zakleszczenie programu:



Rys. 14 Zjawisko zakleszczenia wywołane nieprawidłowym użyciem obiektów Mutex (wersja II)

Sytuacja tak jak ta przedstawiona na rysunku 14 jest sytuacją niedopuszczalną w aplikacji. Jest kilka sposobów na usunięcie problemów zakleszczenia programów, na skutek błędnego użycia obiektów mutex.

Unikanie zakleszczenia wątków przez określenie czasu oczekiwania

Funkcje z zestawu WinAPI związane z mutexami nie wymagają, aby wątek czekał na pozyskanie mutexu w nieskończoność. Funkcja **WaitForSingleObject** określa czas, jaki wątek oczekuje na otrzymanie mutexu. Po przekroczeniu określonego czasu oczekiwania, wątek jest wskrzeszany i zwrócony zostaje błąd z kodem informującym o przekroczeniu czasu oczekiwania. Jeżeli wątek używa metody należy stworzyć odpowiednią obsługę tego błędu – przez ponowienie próby pozyskania, bądź przez rezygnację. Dla porównania – użytkowanie sekcji krytycznej nie daje tej możliwości.

Unikanie zakleszczenia przez organizowanie dostępu do mutexów

Załóżmy, że mamy aplikację z mutexami M1, M2, M3...Mn, gdzie mutexy te mogą być pozyskiwane przez wątki programu. Wątek stara się pozyskać mutex Mi, jeśli nie jest właścicielem wątku o wyższym priorytecie (Mi+1). Wyjaśnimy to na przykładzie. Dla wygody posłużymy się terminami zamykanie i otwieranie obiektów. Terminologia ta wydaje się odpowiednia do rozważania sytuacji, w której przyporządkowany programowi mutex, ma zapewnić atomowy dostęp do danych. Wątek najpierw staje się właścicielem mutexu, potem modyfikuje dane współdzielone a następnie opuszcza mutex. Operacja ta bardzo przypomina obiektowy model programowania. W tym sensie **Obiekt**.**Lock** może być rozpatrywana jak ekwiwalent metody

EnterCriticalSection(sekcja) lub **WaitForSingleObject**(mutex, INFINITE).

Niech dana jest struktura danych w postaci listy. Każdy z obiektów tej listy ma własny mutex. Wątki operujące na tej strukturze mają wykonywać następujące czynności:

- czytają jedną pozycję listy przez zablokowanie jej, odczyt wartości i odblokowanie
- zapisują dane do obiektu przez zablokowanie, zapisanie danej i odblokowanie go
- porównują dwie pozycje przez znalezienie ich w liście, zablokowanie i porównanie wartości

Wyobraźmy sobie trzecią sytuację. Wątek ma porównać pozycje X z pozycją Y w liście. Jeśli wątek będzie zawsze przeprowadzał blokowanie w następujący sposób: najpierw pozycja X, potem pozycja Y; może dojść do zakleszczenia w momencie, gdy inny wątek będzie miał porównać pozycje Y z X. Prostym i zarazem doskonałym rozwiązaniem jest blokowanie jako pierwszej pozycji o niższym indeksie.

Należy również wspomnieć o funkcji **WaitForMultipleObject**(eX). Funkcja ta daje programiście możliwość pozyskania kilku obiektów synchronizacji (np. mutexów) jednocześnie. Umożliwia ona wątkowi oczekiwanie, aż obiekt (lub grupa obiektów) osiągnie status Signaled
(w przypadku mutexów „nie zajęty"), by później uzyskać do niego dostęp. Docenić tę metodę można już w sytuacji, gdzie dwa wątki ubiegają się o pozyskanie mutexów M1 i M2. Nie jest ważne, w jakiej kolejności nastąpiły żądania. Mutexy zostaną pozyskane automatycznie. W tym przypadku zakleszczenie nie może wystąpić.

To podejście ma jednak poważną wadę. Ponieważ wszystkie obiekty synchronizacji muszą mieć status **Signaled** przed tym, jak mogą zostać pozyskane, jeżeli jakiś wątek czeka na wiele obiektów przez długi czas, nie uzyska on prawa do własności, jeśli inne wątki starają się uzyskać te same obiekty na własność w sposób prostszy (pojedynczo). Optymalnym rozwiązaniem dla tego przykładu jest wprowadzenie mutexu dla całej listy, w celu możliwości jej chwilowego zamknięcia. Wszystkie operacje muszą wpierw dokonać zablokowania listy. Wątek może chcieć zmodyfikować zawartość obiektu w liście. Operacja ta powinna wyglądać w ten sposób:

- zamknięcie listy
- znalezienie obiektu w liście
- zablokowanie obiektu,
- odblokowanie listy
- wykonanie modyfikacji zawartości obiektu
- odblokowanie obiektu

To rozwiązanie jest idealne, ponieważ operacje na obiekcie mogą być czasochłonne, jednak nie blokują one w tym czasie listy. Tymczasem inne watki mogą dokonywać operacji na innych obiektach listy. Oto algorytm porównywania dwóch obiektów listy:

- zamknięcie listy
- odnalezienie pierwszego obiektu
- zamknięcie pierwszego obiektu
- znalezienie drugiego obiektu
- zablokowanie drugiego obiektu
- odblokowanie listy
- porównanie obiektów
- zwolnienie obu obiektów (w dowolnym porządku)

Jak widać takie rozwiązanie całkowicie eliminuje możliwość powstania zakleszczenia.

Semafory

Semafory są jeszcze jednym użytecznym prymitywem ułatwiającym proces synchronizacji wątków. Semafory działają w ten sam sposób jak mutexy. Są one jednak o wiele bardziej elastyczne i pozwalają tworzyć bardziej skomplikowane mechanizmy.

Podobnie jak mutexy semafory charakteryzują się tym, że w miejsce określania czy dany semafor ma właściciela określa się tylko wartość licznika. Jeśli ta wartość jest większa od 0 oznacza to, że semafor jest zajęty (mutex jest specjalnym rodzajem semafora, takim, którego licznik przyjmuje tylko wartości 1 lub 0). Semafory mogą być traktowane jak muteksy, które mogą mieć więcej niż jednego właściciela w danym momencie. Funkcje API operujące na semaforach są łudząco podobne do tych operujących na mutexach. Poniższa tabela prezentuje w jaki sposób instancje używające mutexów można przekonwertować do instancji używających semaforów:

Mutexy	Semafory
`Mtx:=CreateMutex(nil,FALSE,<nazwa>);`	`Smfr:= CreateSemaphore(nil,1,1,<nazwa>);`
`Mtx:=CreateMutex(nil,TRUE,<nazwa>);`	`Smfr:= CreateSemaphore(nil,0,1,<nazwa>);`
`WaitForSingleObject(Mtx,INFINITE)`	`WaitForSingleObject(Smfr,INFINITE);`
`ReleaseMutex(Mtx);`	`ReleaseSemaphore(Smfr,1);`
`CloseHandle(Mtx);`	`CloseHandle(Smfr);`

Tabela 10 Konwersja operacji z użyciem mutexów na operacje z użyciem semaforów

Sytuacja, gdy semafory mają wartość licznika większą niż jeden, jest analogiczne do sytuacji, w której mutex ma kilku właścicieli równocześnie. Dlatego mutexy najczęściej działają wespół z mechanizmem sekcji krytycznej, która zezwala na wykonywanie fragmentu kodu czy dostępie do danego obiektu tylko określonym wątkom. Ze względu na swoje możliwości najczęściej semafory wykorzystuje się przy tworzeniu klas kontrolujących przepływ danych. Rozpatrzmy następujący przykład: załóżmy, że stworzyliśmy prostą strukturę

danych współdzielonych, umożliwiającą kontrolę przepływu danych. Struktura ta, to kolejka typu *FIFO* (ang. *First in First Out*). Każdy z elementów tej kolejki ma wskaźniki (P)obierz i (W)staw. Na buforze możliwe są do wykonania cztery operacje:

Stworzenie bufora	Zostaje stworzony bufor wraz z mechanizmem synchronizującym, po czym następuje inicjacja.
Wstawienie elementu	Umieszcza obiekt w buforze w sposób bezpieczny wątkowo. Jeśli umieszczenie elementu w buforze nie jest chwilowo możliwe (np. bufor jest pełny), to wątek usiłujący dokonać operacji wstawiania jest zawieszany aż do chwili, gdy bufor jest w stanie przyjąć więcej danych.
Pobranie elementu	Operacja ta pobiera element z kolejki. Jeśli wykonanie tej operacji jest niemożliwe (pusty bufor), to wątek pobierający zostaje zablokowany aż do czasu w którym bufor umożliwi pobranie z niego danych.
Zniszczenie bufora	Uwalnia wszystkie wątki oczekujące w buforze i niszczy bufor.

Tabela 11 Operacje na buforze

Jak widać w aplikacji manipulującej danymi współdzielonymi można użyć mechanizmu mutexów. Jednakowoż, wszystkie potrzebne nam operacje blokowania kiedy bufor jest pełny lub pusty możemy zrealizować za pomocą semaforów. Mało tego rozwiązanie to zwolni nas z konieczności sprawdzania zakresu danych (wielkości bufora) oraz sprawdzania liczby elementów w buforze. Do bufora wprowadzamy parę semaforów zliczających liczbę elementów (pu)stych i (pe)łnych w buforze.

Rozważmy sytuację, w której na buforze operują dwa wątki. Jeden wątek stara się zapisać element do bufora, drugi natomiast stara się go stamtąd pozyskać. Jak widać, watek zapisujący i czytający dane działają w pętli. Wątek piszący do bufora stara się umieścić element w buforze. W tym celu najpierw wywołuje metodę Wait semafora (pu) i jeśli licznik tego semafora jest równy zero, wtedy wątek zostaje zablokowany aż do momentu, gdy bufor nie będzie już pełny i możliwy będzie zapis do niego. Kiedy już watek zapisze nowy element do bufora zwiększa on licznik elementów (pe)łnych (gdy w kolejce do bufora jest uśpiony wątek pobierający dane zostaje on teraz wzbudzony). Odwrotnie sprawa ma się z wątkiem pobierającym element. Jest on blokowany, gdy licznik semafora zliczającego elementy (pe)łne sięgnie zera. Natomiast zwolnienie (pobranie) elementu z bufora powoduje zwiększenie (pu). Z takiego rozwiązani płyną następujące korzyści:

- żaden wątek nie może produkować zbyt dużej liczby elementów – przez co nie mamy tu do czynienia ze stale zwiększającą objętość strukturą przechowującą dane. W ten sposób osiągamy zmniejszenie zajętości pamięci – bufor ma skończone granice
- zmniejszenie zajętości procesora – unikniecie sytuacji pisania pętli dla wątków oczekujących na zwolnienie/ zapełnienie bufora. Kiedy wątek nie ma żadnego zadania do wykonania wtedy jest wstrzymywany

Dodatkowo dokonamy ilustracji tej idei na przykładzie sekwencji zdarzeń. Niech dany będzie bufor z

czterema miejscami na elementy – początkowo te miejsca są puste.

Reakcje wątku czytającego	**Reakcje wątku piszącego**	u	e
Stworzenie wątku	Wątek nieaktywny		
Wait (pe) – zostaje zawieszony			
	Wait (pu) – przechodzi dalej		
	Element dodany Signal (pe)		
	Wait (pu) – przechodzi dalej		
	Element dodany Signal (pe)		
	Wait (pu) – przechodzi dalej		
	Element dodany Signal (pe)		
	Wait (pu) – przechodzi dalej		
	Element dodany Signal (pe)		
	Wait (pu) – zostaje zawieszony		
Wait (pe) – przechodzi dalej			
Element dodany Signal (pu)			
Wait (pe) – przechodzi dalej			
Element dodany Signal (pu)			
Wait (pe) – przechodzi dalej			
Element dodany Signal (pu)			
Wait (pe) – przechodzi dalej			
Element dodany Signal (pu)			
Wait (pe) – zostaje zawieszony			

Tabela 12 Działanie aplikacji dwuwątkowej wykorzystującej semafory

Chyba każdy, kto budował w swym życiu aplikacje wielowątkowe wie, że najwięcej problemów sprawia zawsze część realizująca niszczenie obiektów i „sprzątanie". Warto więc uważnie przyjrzeć się co tak naprawdę dzieje się w systemie, gdy zamykamy semafor lub mutex. Czy zamknięcie uchwytu powoduje odblokowanie (wzbudzenie wszystkich wątków oczekujących na dany mutex/semafor)? Otóż po przeprowadzonym przeze mnie śledztwie okazało się, że zamknięcie uchwytu danego obiektu synchronizacji nie odblokowuje żadnych wątków oczekujących na ten obiekt. Wszystkie czynności zwolnienia zasobów systemowych powinien wykonać programista.

MREWS i zdarzenia

Na koniec omówimy dwa najbardziej zaawansowane mechanizmy synchronizacji. Jak do tej pory nasuwa się wniosek, że wszystkie wcześniej poznane mechanizmy można skonstruować przy użyciu dwóch podstawowych „cegiełek" mianowicie: mutexów i semaforów (pod warunkiem, ze mamy sporo wolnego

czasu i lubimy go marnować). W programowaniu wielowątkowym bardzo często spotykamy się z pewnymi problemami, których nie da się rozwiązać za pomocą mechanizmów opisanych do tej pory. W celu rozwiązania takich właśnie problemów opracowano mechanizm synchronizacji **MREWS** (ang. *Multi Read Exclusive Write Synchronizer*) oraz *zdarzenia* (ang. *events*). Mechanizmy te oferowane są przez system Win32 API.

Konstruowanie klasy bezpiecznej wątkowo w systemie Delphi

Czasami stworzenie klasy bezpiecznej wątkowo jest jedynym sensownym rozwiązaniem. Kod umieszczony w bibliotekach DLL korzystający z niektórych zasobów systemowych musi zawierać mechanizm synchronizacji pomimo tego, że nie zawiera żadnego obiektu klasy TThread. Biblioteka taka musi zawierać klasy bezpieczne wątkowo.

Enkapsulacja wątkowo bezpiecznej klasy. Jest to najprostszy typ klasy wielowątkowej. Często jest tak, że uczynienie klasy bezpiecznej wątkowo polega jedynie na dodaniu obiektu synchronizacji (np. mutex) i dwóch metod składowych Zablokuj i Odblokuj.

Klasa kontrolująca przepływ danych. Jest to poszerzona klasa opisana wyżej, dodatkowo zawierająca klasy bufora: listy, stosy i kolejki. Dodatkowo klasa ta może automatycznie zajmować się przepływem danych między wątkami. Przykładem takiej klasy jest Winsocket z WinAPI.

Klasa monitorująca. Kolejny krok naprzód w stosunku do poprzedniej klasy. Klasa ta dopuszcza równoczesny dostęp do danych, wymagających bardziej wyrafinowanych mechanizmów blokujących. Przykładowo, silniki baz danych *BDE* (ang. *Borland Database Engine*) należą do tej klasy. Mechanizmami charakterystycznymi dla tej klasy są transakcje **Rollback**, **Commit**. Innym przykładem implementacji tego typu klasy jest system obsługi plików w systemie Windows.

Zarządzanie priorytetami wątków

Program szeregujący Win32 Scheduler dzieli czas procesora między aktywne wątki w systemie. Aby tego dokonać musi wiedzieć ile czasu procesora dany wątek zamierza zużyć. Większość systemów operacyjnych przydziela priorytety do wątków w celu rozpoznania, który wątek powinien dostać ile czasu procesora. W przypadku systemu Windows priorytet wątku obliczany jest „w locie" na podstawie kilku składników. Niektóre priorytety mogą być bezpośrednio zadawane przez programistę. Do składników tych zaliczamy: *klasę priorytetu procesu* (ang. *priority class*), *poziom priorytetu wątku* (ang. *priority level*), z których wyliczany jest *priorytet bazowy wątku* (ang. *base priority*), a w rezultacie *priorytet wznoszący* (ang. *boost priority*). Klasa priorytetu dla większości aplikacji środowiska Delphi jest ustawiona na wartość nor-

malną (z wyjątkiem np. wygaszaczy ekranu, które mogą mieć ustawioną tą klasę na wartość idle – jałową). Nie ma potrzeby zmiany priorytetu klasy podczas gdy proces jest już wykonywany. Poziom priorytetu wątku może być ustawiany z wnętrza klasy przyporządkowanej procesowi. Jest to bardzo użyteczne. Można tego dokonać za pomocą funkcji API **SetThreadPriority**.

Dopuszczalnymi wartościami dla tej funkcji są:

```
THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_NORMAL, THREAD_
PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_LOWEST oraz THREAD_PRIORITY_IDLE
```

Priorytet wznoszący jest używany przez Win 32 Scheduler do obserwacji (kontroli) zachowań aktywnych wątków. Dzięki temu priorytetowi zagospodarowywany jest czas procesora miedzy działające w systemie wątki.

Jaki priorytet powinien mieć napisany przez nas wątek? Generalnie, większość aplikacji napisanych w środowisku Delphi skupia się na osiągnięciu maksimum „reagowalności" (natychmiastowe odpowiedzi na bodźce zewnętrzne), dlatego rzadko zwiększa się priorytet wątku ponad normalny. O dziwo, obniżenie priorytetu wątku może okazać się bardzo użyteczne – wzrasta „reagowalność" aplikacji.

Poprawienie efektywności

Wszystkie do tej pory omówione operacje na danych współdzielonych były operacjami z *wyłącznością* (ang. *exclusive*). Operacje zapisu czy odczytu, niezależnie od liczby wątków, były wykonywane pojedynczo (tylko jeden wątek w danym czasie). Najlepsze poprawienie wydajności aplikacji uzyskujemy w sytuacji, gdzie jednocześnie mamy wiele wątków czytających współdzielone dane (przy czym częstotliwość czytania jest duża) i tylko klika wątków zapisujących dane (przy czym częstotliwość pisania jest mała). Prymitywem, który ułatwia efektywne budowanie aplikacji wielowarstwowych jest mechanizm MREWS.

Większość mechanizmów synchronizacji obsługuje cztery podstawowe operacje: **StartRead**, **StartWrite**, **EndRead**, **EndWrite**. Wątek, który chce przeczytać dane wysyła sygnał **StartRead**, zaś po przeczytaniu danych wywołuje metodę **EndRead**. Istnieje możliwość obsługiwania innych żądań odczytu równocześnie (miedzy parą **StartRead** a **EndRead** mogą się pojawić inne pary wywołań tych metod, natomiast nie jest możliwe wywołanie w tym czasie **StartWrite**). Co do zapisu – tylko jedna operacja może być wykonywana w danej chwili, wszystkie wątki zgłaszające chęć odczytu lub zapisu danych zostaną zablokowane.

Jest kilka sposobów implementowania synchronizacji (**MREWS)**. Każdy obiekt synchronizacji musi mieć następujące składniki:

sekcję krytyczną	do ochrony danych współdzielonych
licznik wątków aktywnych	zgłaszających chęć czytania
licznik wątków czytających	wykonujących operacje odczytu
licznik wątków aktywnych	zgłaszających chęć zapisu
licznik wątków zapisujących	wykonujących operacje zapisu

parę semaforów	semafor odczytu (ReaderSem) i semafor zapisu (WriterSem)
sekcję krytyczna	wymuszająca wyłączność operacji zapisu

Tabela 13 Składniki obiektu synchronizacji. O. W.

Wątki mogą znajdować się w dwóch stanach. Pierwszy z nich to stan aktywności, w którym wątek zgłasza chęć zapisu lub odczytu danych. Kiedy to się już zdarzy wątek może zostać zablokowany – w zależności od tego czy aktualnie wykonywane są jakieś operacje zapisu lub odczytu. Kiedy zostaje odblokowany, to wykonuje operację odczytu lub zapisu, a następnie zwalnia zasoby współdzielone aktualizując przy okazji liczniki wątków. Jeżeli wątek jest ostatnim wątkiem zapisującym lub odczytującym, to odblokowuje on wszystkie pozostałe wątki zablokowane wskutek wykonywanej przez niego operacji zapisu czy odczytu.

Rozdział V. Inicjatywa FastCode

Projekt FastCode udostepnia zoptymalizowane i poprawione funkcje oraz komponenty Delphi. Udostępnione w nim funkcje są szybszymi wersjami ich odpowiedników ze środowiska Delphi (Runtime Library) jak również kontrolek VCL. Projekt rozpoczął w 2003 roku Dennis Christensen. Ma on charakter otwartego projektu, dla którego kontrybuować może każdy programista.

Projekt zorganizowany jest jako konkurs podzielony na kilka kategorii. Każda z kategorii dotyczy optymalizacji pewnej funkcji pod różnymi kątami. W obrębie projektu udostępnione są narzędzia do analizy wydajności i walidacji funkcji zgłaszanych na konkurs. Za każdą poprawną funkcję dodaną do projektu użytkownik dostaje punkty. Wyniki publikowane są pod koniec każdego roku, a zwycięzcy nagradzani są przez Borland, Codegear i Embracadero.

Większość z funkcji umieszczonych w FastCode to kod stworzony w assemblerze. Embracadero wspiera inicjatywę FC, dodatkowo wybrane funkcje dodane zostają do kodu bazowego Delphi.

Jakość kodu z projektu FC jest bardzo wysoka. Testowanie odgrywa istotną rolę w procesie publikacji materiałów. Z racji tego, że większość kodu tworzona jest w assemblerze, proces walidacji odbywa się na konfiguracjach. Funkcje testowane są na w różnych systemach operacyjnych i na różnym sprzęcie.

Definitywnie FastCode jest inicjatywą wartą przyjrzenia się. Opublikowane tam materiały można wykorzystywać również w swoim oprogramowaniu.

Rozdział VI. Testowanie oprogramowania

Ostatnim elementem tworzenia optymalnego oprogramowania jest jego finalne testowanie. Testowanie oprogramowania jest nie tylko procesem, służącym do poszukiwania błędów oraz badania jakości oprogramowania, przekłada się również na zmniejszenie kosztów produkcji oprogramowania. Testując program i usuwając błędy wykryte w początkowych etapach powstawania programu, można zaoszczędzić znacznie więcej niż w przypadku wykrycia błędu w późnej fazie projektu albo w programie, który jest już zainstalowany w środowisku produkcyjnym.

Jak się okazuje automatyzacja procesu testowania jest ważnym elementem procesu rozwoju oprogramowania zgodnym z metodykami lekkimi. Jednym z celów takich działań są oszczędności – oszczędności czasu i budżetu projektu. Metodyki lekkie zalecają opracowywanie i implementację zbioru testów jeszcze przed powstaniem właściwego kodu źródłowego aplikacji. Zmiany dokonywane podczas czynności refaktoringowych, mających na celu uproszczenie i podwyższenie jakości kodu źródłowego z zachowaniem pierwotnej logiki rozwiązań, mogą zostać w szybki, niemal w pełni automatyczny sposób przetestowane. Następnym znaczącym argumentem przemawiającym za stosowaniem automatyzacji jest wyraźna niezależność etapów testowania i implementacji.

Rodzaje testów

Testy oprogramowania dzielimy na dwie główne grupy. Pierwsza z nich to testy funkcjonalne (program traktowany, jako „czarna skrzynka"). Drugą grupę stanowią testy obejmujące testy jednostkowe, testy integracyjne i testy funkcjonalne.

Testy z pierwszej grupy zazwyczaj obejmują zagadnienia takie jak:

- prawidłowy przebieg wszystkich przypadków użycia
- prawidłowe działanie interfejsu użytkownika
- prawidłowe działanie walidatorów

Nazwę „czarno-skrzynkowe" zawdzięczają temu, iż testerzy nie wiedzą, jak system działa i co jest w środku. Zadaniem testerów jest sprawdzenie, czy program działa w prawidłowy sposób. Osoby testujące zazwyczaj nie uczestniczą w procesie wytwarzania oprogramowania.

Testy drugiej grupy obejmują sobą:

- sprawdzenie prawidłowego współdziałania z platformą
- sprawdzenie prawidłowego współdziałania poszczególnych modułów
- sprawdzenie czy moduły dostarczają odpowiednich funkcjonalności w zamierzony

sposób

- sprawdzenie prawidłowego działania poszczególnych klas i metod.

Poniżej opisane zostaną dokładniej testy „białoskrzynkowe”

Testy modułowe (jednostkowe)

Testy takie muszą spełniać kilka założeń.

- Niezależność

Testy powinny być niezależne. Oznacza to, że dwa dowolne testy nie wpływają na siebie nawzajem. Nie istnieje konieczność utrzymania sztywnej kolejności wywołania testów. Nie zawsze można uzyskać taki stan, ale warto zadbać o jak największą niezależność testów.

- Powtarzalność

Testy powinny być powtarzalne, łatwe do uruchomienia w dowolnym momencie i gwarantujące stabilność. Oznacza to, że pisząc test jednostkowy możemy w dowolnym momencie go uruchomić bez konieczności uruchamiania dodatkowych elementów. Powtarzalność oznacza też, że test daje taki sam wynik za każdym razem dla danego zestawu danych.

- Jednoznaczność

Test jednostkowy powinien być jednoznaczny, czyli jasno odpowiadać na pytanie o poprawność działania testowanej funkcjonalności.

- Jednostkowość

Test jednostkowy to test, który testuje jedną rzecz na raz. Nie wolno pisać testu, w którym staramy się sprawdzić dwie funkcjonalności. Nie należy też sprawdzać w jednym teście kilku zestawów danych. Taki test w przypadku niepowodzenia nie zwraca jednoznacznych wyników.

Najczęście za pomoca testów jednostkowych sprawdzane są:

• Czy wyniki są poprawne?
• Czy warunki brzegowe zostały prawidłowo określone?
• Czy można sprawdzić relacje zachodzące w odwrotnym kierunku?
• Czy można sprawdzić wyniki w alternatywny sposób?
• Czy można wymusić błędy?
• Czy efektywność jest zadowalająca?

Testy modułowe służą do sprawdzenia, czy metody testowanej klasy, funkcje lub procedury działają w zamierzony sposób. W środowisku Delphi najpopularniejszym mechanizmem prowadzenia testów modu-

łowych jest wykorzystanie asercji – za pomocą procedury **Assert**. Procedura ta generuje wyjątek, gdy podana wartość logiczna (sprawdzany warunek) – nie jest prawdziwa. O ile, podczas pisania małych programów, wygodnym może okazać się umieszczenie wywołania tej procedury wewnątrz sprawdzanej funkcji, o tyle w dużych projektach zaleca się oddzielenie modułów testowania od samego programu. Przykład sprawdzenia poprawności funkcji:

```
uses SysUtils;
function Kwadrat(x:double):double;
begin
result:=sqr(x);
end;

const Liczba = -1.234;
begin
Assert(Liczba*Liczba <> Kwadrat(Liczba));
end.
```

Biblioteka **SysUtils** odpowiada za prezentację wyjątku, który może wystąpić w aplikacji, zamiast generowania błędu środowiska uruchomieniowego. Jest ona niezbędna do organizacji testów modułowych.

Procedury testuje się trudniej, gdyż zamiast zwracanej wartości należy testować stan systemu (zawartości pamięci czy urządzeń) lub wartości parametrów podawanych inaczej niż przez wartość, a więc przez nazwę, przez wskaźnik czy przez zmienną globalną.

Testowanie z raportowaniem

Ponieważ procedura **Assert** generuje wyjątek w razie niezgodności, można wykorzystać blok **try catch** do jego przechwycenia i prezentacji (plik, ekran) informacji o niezgodności. Wygodniejszym może jednak okazać się mechanizm, udostępniony przez sam język Delphi, a mianowicie zmienna globalna **AssertErrorProc**, przechowująca adres do procedury obsługi wyjątku asercji.

Przykład wykorzystania procedury obsługi niezgodności:

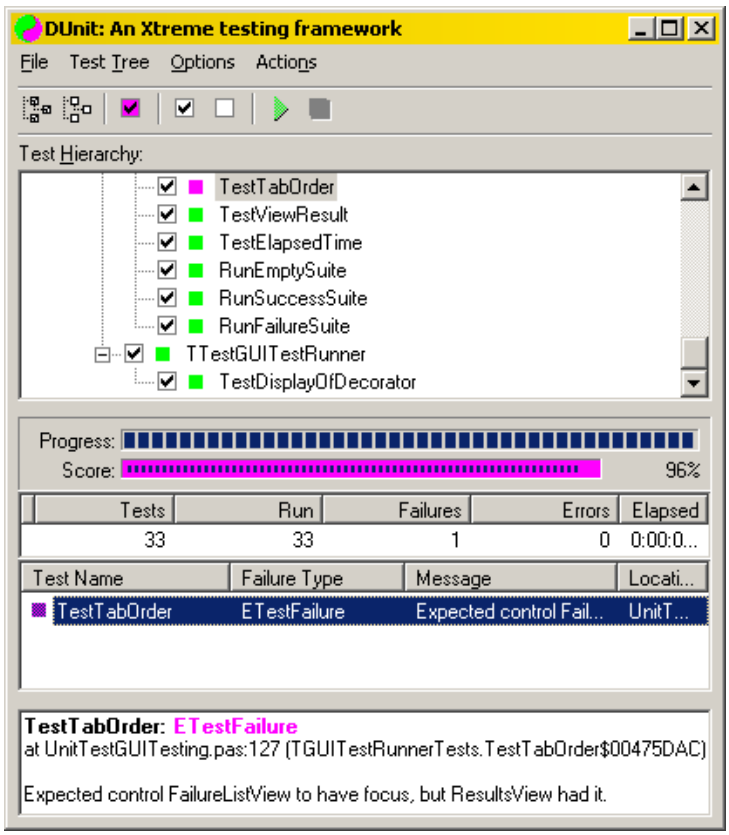
```
uses SysUtils;
function Kwadrat(x:double):double;
begin
result:=sqr(x);
end;
//Procedura obsługi niezgodności
procedure AssertErrorHandler(const Message, Filename: string; LineNumber: Inte-
ger; ErrorAddress: Pointer);
var TextFile:text;
begin
AssignFile(TextFile, 'c:\\raport.log');
```

```
Rewrite(TextFile);
      WriteLn(TextFile, Format('(Linia: %d, Adres: $%d) "%s": %s', [LineNumber,
Integer(ErrorAddress), ExtractFileName(FileName), Message]));
CloseFile(TextFile);
end;
//Test
const Liczba = -1.234;
begin
AssertErrorProc:=AssertErrorHandler;
      Assert(Liczba*Liczba <> Kwadrat(Liczba), ,Błąd funkcji Kwadrat(x);');
end.
```

Turbo Delphi Explorer i DUnit

Pracę z modułem **Duinit** rozpocząć należy od instalacji narzędzia Turbo Delphi Explorer. Narzędzie DUnit możemy wybrać opcjonalnie z listy dostępnych składników pakietu. Producent udostępnił programiście dwa kreatory przypadków testowych (Test Case). Skorzystajmy najpierw z pierwszego kreatora. Przystępując do testów należy najpierw samemu skonfigurować **menu: File->New** dodając za pomocą funkcji **File->Configure** ww. elementy. Następnie wybierając funkcję **File->Test Project Delphi** wyświetli okno **Test Project Wizard**.

W pierwszym kroku procedury należy wybrać nazwę projektu oraz miejsce, gdzie będą przechowywane pliki wynikowe. Standardowo jest to folder Test. W oknie Personality należy wskazać Delphi. W kolejnym kroku można wybrać typ aplikacji DUnit. Do wyboru mamy aplikację uruchamianą w oknie konsoli oraz interfejs graficzny. Wskazujemy GUI oraz naciskamy przycisk Finish.



Rys. 15 Ekran aplikacji DUnit

Do grupy projektów został dodany nowy projekt testów. Z **menu: File** wybieramy funkcje **TestCase**. W kroku pierwszym wskazujemy klasę oraz metody, które chcemy przetestować. W następnym kroku należy wybrać należy projekt, do którego ma zostać dodany przypadek testowy oraz nazwę pliku, w którym zostanie zapamiętany. Klasą generalizacji dla wybranej klasy testującej jest klasa **TTestCase**. Analizując kod źródłowy widzimy, że zostały wygenerowane metody **SetUp, TearDown, TestWybranaMetoda**. Metody te mogą być znane użytkownikom **JUnit**, jednak w celu przypomnienia przytoczę krótko ich semantykę. Metoda **SetUp** uruchamiana jest przez szkielet tylko raz, podczas inicjalizacji procesu testowania funkcji. **TearDown** jest również redefinicją oraz uruchamiana jest po zakończeniu testów kolejnej funkcji np. w celu zwolnienia wykorzystywanych zasobów. Naszą uwagę skupmy na metodzie **TestWybranaMetoda**, ponieważ w jej ciele zostanie zaimplementowany kod źródłowy przypadku testowego. Na uwagę zasługuje wywołanie metody **Check**. Metoda ta umożliwia weryfikację otrzymanych wyników oraz dodatkowo daje programiście możliwość zdefiniowania komunikatu informacyjnego w przypadku wystąpienia błędu. Innymi przydatnymi metodami są: **CheckTrue, CheckFalse, CheckEquals, CheckNotEquals**. W celu zapoznania się ze sposobem ich działania odsyłam czytelnika do udostępnionej dokumentacji i kodu źródłowego **DUnita**. Tak przygotowany projekt uruchamiamy. Na pulpicie pojawi się okno **DUnit**: An Xtreme testing framework (patrz rysunek wyżej). W tym momencie możemy przystąpić do testowania. W oknie **Test Hierarchy** wybieramy wcześniej zaimplementowaną metodę **TestWybranaMetoda** oraz uruchamiamy test.

Wykrywanie wycieków pamięci a DUnit

DUnit posiada opcję wykrywania przecieków pamięci podczas wykonywania testów tzn. sprawdza całkowitą ilość zaalokowanej pamięci przed uruchomieniem testu i po jego zakończeniu, a gdy druga wartość jest większa od pierwszej zgłasza błąd wykonania testu.

Aby włączyć tę funkcjonalność w **DUnit**, należy skorzystać z pozycji **Fail TestCase if memory leaked** w **menu: Options**. Program może zgłaszać błędy również w sytuacji prawidłowej np.: gdy podczas testu alokowana jest pamięć, która jest zwalniania nie w samym teście, ale później (np. przy zamykaniu programu). Dzieje się tak często z wywołaniami systemowymi, bibliotekami zewnętrznymi czy wielomodułowym kodem własnym. Często spotykanym rozwiązaniem stosowanym w programowaniu jest alokacja pamięci w sposób dynamiczny (w przeciwieństwie do explicite) – przy pierwszym wywołaniu procedury/funkcji czy przy pierwszym odwołaniu do obiektu/klasy, które wymagają dodatkowej pamięci. A pamięć w ten sposób przydzielona często jest zwalniana dopiero wtedy, gdy klasa/moduł usuwane są z pamięci systemu podczas zamykania aplikacji (np. w sekcji finalization modułu). Załóżmy na przykład, że testujemy kod bazodanowy (połaczenie z MySQL poprzez DBExpress). W teście tworzymy połączenie, wykonujemy operacje na danych i zwalniamy połączenie. Niestety, ponieważ DBExpress alokuje pamięć przy tworzeniu połączenia po raz pierwszy, i nie zwalnia jej dopóki aplikacja nie zakończy działania, otrzymujemy błąd przeciekania pamięci:

```
procedure TYACDBTests.TestMemory;
var LConnection: TSQLConnection;
begin
    LConnection := CreateSQLConnection;
    Check(TRUE);
    FreeAndNIL(LConnection);
end;
```

Kod jest w pełni poprawny (gdzie **CreateSQLConnection** tworzy obiekt, ustawia odpowiednia parametry i nawiązuje połączenie), ale nadal otrzymujemy informację o przeciekającej pamięci.

Jednym z możliwych rozwiązań jest dodanie kodu, który wymusiłby tę pierwszą alokację jeszcze przed uruchomieniem testów, aby w trakcie testów pamięć była już przydzielona (a więc i uwzględniona w obliczeniach zajętości pamięci przed testem).

Kod taki można umieścić np. w części inicjalizacyjnej modułu:

```
Var LConnection: TSQLConnection;
initialization
    LConnection := CreateSQLConnection;
    FreeAndNIL(LConnection);
end.
```

Teraz, gdy tworzymy połączenie **TSQLConnection** w teście, nie alokuje już ono więcej pamięci, zatem i **DUnit** nie raportuje przecieku!

Tego typu sytuacje są w miarę łatwo wykrywalne (choć nie w 100%), gdy w pierwszym przebiegu **DUnit** zgłasza przeciek, ale przy drugim już nie (oba przebiegi wykonujemy podczas jednego uruchomienia programu **DUnit**). Wtedy mamy dużą szansę na to, że pamięć jest alokowana w pierwszym przebiegu, a w drugim jest już tylko wykorzystywana.

Uruchamianie testów DUnit z lini poleceń

Taki scenariusz pracy z automatyzacją testów jest preferowany przez większość developerów. Pozwala to – bez wbudowanych bezpośrednio w Delphi narzędzi – przeprowadzanie serii testów.

Plik **Dunit.jar** daje możliwości uruchamiania środowiska graficznego jak i poleceń z linii komend. Tryb graficzny jest preferowany w przypadku scenariuszy testowych o charakterze interakcyjnym. Wywoływanie testów z poziomu linii komend daje możliwość integracji testów z procesem build.

Składnia uruchominiea testów dla przykładowego projektu (**DUnitTests.dpr**) z linii poleceń:

```
dcc32 -ud:\src\srcpas\dunit\src -b -cc DUnitTests.dpr
```

Uruchamianie testów w trybie tekstowym

W celu uruchomienia testów w trybie tekstowym należy skorzystać ze składni:

```
dcc32 -ud:\src\srcpas\dunit\src -b -cc DUnitTests.dpr
```

Poniżej przykład wywołania tekstowego **DUnit**:

```
[g:\srcjava\jbbook\debug\unittesting]runTextTests.bat
java -cp .;g:/utils/classes/jaxp.jar;g:/utils/classes/codebox.jar;g:/jb7/
lib/dx.
jar;g:/jb7/lib/dbswing.jar;g:/jb7/lib/jbcl.jar;g:/jb7/lib/DUnit.jar;g:/
utils/cla
sses/mm.mysql-2.0.11-bin.jar;g:/Compilers/interbase/InterClient/intercli-
ent.jar;
classes DUnit.textui.TestRunner unittesting.AllTests
...GetFirstWord: OneWord
.GetFirstWord:
.GetFirstWord: Two
.GetFirstWord: A
...Data
Call: 01
Call: 001
IntToStrPad0: 001
.Data
......
Time: 0.5

OK (16 tests)
```

Pierwsza linijka zawiera wywołanie pliku bat z komendami widocznymi później w tekście. Widać tam całą ścieżkę **classpath**. Rozpoznać można również nazwę klasy przechowująca testy do uruchomienia **unittesting.AllTests**.

Referencje do **GetFirstWord** i **IntToStrPad0**, to wyniki kolejnych wywołań **System.out.println** umieszczonych w kodzie testów. Następna linia prezentuje czas trwania testów, a ostatnia linka informuje, ze z sukcesem uruchomiono 16 testów.

Obiekty imitacji (Mock Objects)

W najprościej mówiąc, obiekt imitacji jest po prostu alternatywną implementacją klasy udostępniającej „fałszywe” odpowiedzi na wywołania metod. Przykładowo: klasa **TCustomer** posiadająca metodę **GetCustomerName**, normalnie wywołuje ona połączenie do produkcyjnej bazy danych i pobiera z niej określoną wartość. Aby uniknąć połączenia do bazy danych należy utworzyć obiekt **TMockCustomer** i zamplemetować wywołanie metody **GetCustomerName** zwracającej predefiniowane dane np. „Optymalizacja kodu”. Dzięki temu można przetestować klasę, która używa **TCustomer** bez konieczności łączenia się z bazą danych.

Najczęściej decydujemy się na zastosowanie mock object gdy:

- nie chcemy, aby obiekt rzeczywisty brał udział w teście
- obiekt rzeczywisty zachowuje się niedeterministycznie
- obiekt rzeczywisty jest trudny do skonfigurowania
- trudno jest wywołać interesujące nas zachowanie obiektu (np. błąd sieci)
- obiekt rzeczywisty działa powoli
- obiekt rzeczywisty ma interfejs użytkownika
- obiekt rzeczywisty jeszcze nie istnieje

Jednak to zbyt trywialne podejście. W przypadku, kiedy konieczne jest wprowadzenie dodatkowej logiki związanej z badaniem danych zwracanych przez funkcję – obiekt imitacji należy rozbudować.

Co jeżeli możemy mieć framework, który pozwoliłoby na realizację dowolnego interfejsu i zdefiniowania w prosty i dokładny sposób rodzaju danych wejściowych i wyjściowych? Pozwoliłoby to na łatwe tworzenie obiektu imitacji, który może odpowiedzieć na wywołania metod w określony, łatwy do skonfigurowania sposobów. Taką funkcjonalność daje nam Mocking Framework.

To potężne narzędzie jest w stanie dynamicznie rozpoznać wywołania metod i odpowiednio na nie reagować. Na szczęście, Delphi XE2 jest w stanie sprostać temu zadaniu. Delphi XE2 wprowadza klasę **TVirtualInterface** który pozwala dynamicznie realizować każdy interfejs.

Jedną z najbardziej popularnych bibliotek z mock object jest stworzony przez **Vince Parrett** z VSoft Technologies – **FinalBuilder** (Delphi Mocks). Autor napisał rzetelny przewodnik po tej bibliotece, ja postaram się w dalszej częsci zbudować prosty przykład dość użycia obiektów imitacji.

Typowym scenariuszem użycia obiektów mock jest zastapienie usługi (metody/procedury), która konsumuje sporo zasobów systemowych. Mogą to być cykle procesora, połączenia z bazą danych lub dowolne operacje wpływające na zewnętrzne zasoby, które uniemożliwiają testowanie danego modułu/funkcji w izolacji. Ponieważ testy jednostkowe powinny zawsze testować coś atomowo nie tworząc niepotrzebnych „pozostałości" (wpisy bazy danych, pliki, itp). Za każdym razem, kiedy będzie się tak działo podczas uruchamiania testów jednostkowych, należy rozważyć zastosowanie obiektu mock .

Typowym przykładem jest klasa logowania. Jeśli przeprowadzamy testy jednostkowe, nie jest pożądane każdorazowo zapełniać rejestr wpisów, dlatego tworzy się klasę pośrednika (stub) **Logger**, który ma budowę jak prawdziwy system logowania, ale faktycznie nic nie robi.

Równie ciekawymi zasobami w sieci, traktującymi o tematyce obiektów imitacji są serwisy:

- http://sourceforge.net/projects/mockobjects/
- http://www.mockobjects.com/
- http://easymock.org

Testy akceptacyjne

Testy akceptacyjne, to testy funkcjonalne, których celem jest wykazanie, że wyspecyfikowane wymagania zostały poprawnie zaimplementowane. W metodykach lekkich (np. eXtreeme Programming) często stanowią integralną część specyfikacji i są automatyzowane przy pomocy jednego z wielu dostępnych narzędzi (**Fitnesse**, **Fit**, **Selenium**, **BadBoy**, **Proven**, **Abbot**, **jfcUnit**, **AutoIt**). Kiedy wszystkie testy akceptacyjne przypisane do historii użytkownika (przypadku użycia) zostaną poprawnie przeprowadzone historia jest uważana za poprawnie zaimplementowaną

Najczęściej wykorzystywane narzędzia – ułatwiające testy akceptacyjne:

- testowanie przez GUI
  - Capture & Replay
- aplikacje desktopowe
  - Abbot (http://abbot.sourceforge.net)
  - JfcUnit (http://jfcunit.sourceforge.net/)
- aplikacje internetowe
  - Selenium (http://seleniumhq.org/)
  - Kompleksowe rozszerzalne frameworki
  - Fit (http://fit.c2.com/)
  - Fitnesse (http://fitnesse.org/)
  - Proven

Podsumowanie

Optymalizacja jest jedną z najważniejszych gałęzi inżynierii oprogramowania. Często proces optymalizacji stoi w opozycji do pozostałych procesów i celów inżynierii oprogramowania. Jako twórca musimy wybierać między stabilnością, małym rozmiarem, przenaszalnością, a szybkością pracy programu. Jakkolwiek na poziomie kodu optymalizacja jest pożądana i zawsze przynosi korzystne skutki. Optymalizacja pozwala poprawić wydajność, często jednak taki kod jest trudniejszy do debugowania, ponieważ utracona zostaje pełna odpowiedniość pomiędzy kodem źródłowym a wykonywanym.

Aplikacje i narzędzia IT są kluczowym elementem działania każdej firmy i instytucji. Nadają one tempo działalności i określają konkurencyjność firmy. Dlatego przyspieszenie i optymalizacja oprogramowania decyduje o sukcesie komercyjnym danego narzędzia. Mam nadzieję, że powyższa książka ułatwi w przyszłości tworzenie optymalnego kodu.

Techniki użytkowe prezentowane w książce, są na tyle uniwersalne, że mogą znaleźć zastosowanie większości środowisk programistycznych. Materiał opisuje również narzędzia niezbędne do prac optymalizacyjnych. Oprócz technik użytkowych stosowanych w samych algorytmach, należy zwrócić uwagę na narzędzia ułatwiające sprawdzanie wydajności kodu oraz jego stabilność. Przybliżone zostało też zagadnienie testowania oprogramowania – bezpośrednio wiążące się fazami optymalizacji produkcji oprogramowania.

Bibliografia

1. Jaszkiewicz Andrzej, *Inżynieria oprogramowania*, Helion, Gliwice 1997.
2. Chrapko Mariusz*, CMMI – doskonalenie procesów organizacji*, Wyd. PWN, Warszawa 2010.
3. Górski Janusz (red.) , *Inżynieria oprogramowania w projekcie informatycznym*, Mikom, Warszawa 2000.
4. Falk Heiko, Marwedel Peter, *Source code optimization techniques for data flow dominated embedded software*, Springer Science+Business Media, New York 2004.
5. Kaspersky Kris, *Optymalizacja kodu. Efektywne wykorzystanie pamięci*, Oficyna Wydawnicza READ ME, Warszawa 2005.
6. Marianini Rico, Bohling Brandon, Connie U. Smith, Scott Barber, *Improving .NET Application Performance and Scalability. Patterns & Practices*, , Microsoft Press, Redmond 2004.
7. Wright Charles, *C# porady & metody*, Wydawnictwo Nakom, Poznań 2002.
8. Jones Allen, *C#: Księga przykładów*, APN Promise, Warszawa 2006.

Netografia

1. Lechert Łukasz, *Delphi i testy jednostkowe*, „Software Developer's Journal" 2010, dostępny w Internecie: http://sdjournal.pl/article/13159-dunit-delphi-i-testy-jednostkowe dostęp [04.08.2013].
2. Podstawa Roman, *Optymalizacja kodu na platformie .NET*, dostępny w Internecie: http://codeguru.geekclub.pl/baza-wiedzy/optymalizacja-kodu-na-platformie-net,2110 [dostęp: 04.08.2013].
3. Jureczko Marian, *Testowanie oprogramowania*. *Część 1*, 2010. dostępne w Internecie: http://staff.iiar.pwr.wroc.pl/marian.jureczko/testy1.pdf oraz [dostęp: 04.08.2013].
4. Jureczko Marian, *Testowanie oprogramowania. Część 2*, 2010. dostępne w Internecie: oraz http://staff.iiar.pwr.wroc.pl/marian.jureczko/testy2.pdf [dostęp: 04.08.2013].
5. www.fastcode.sourceforge.net
6. www.drbob42.com
7. www.delphi.about.com
8. www.docwiki.embracadero.com
9. www.4programmers.net
10. www.bbproject.net
11. www.dyszla.aplus.pl

www.e-naukowiec.eu

ISBN 978-83-936418-8-8