Michał Kosyl

Marta Kuzak

Prowadzący: mgr inż. Paweł Zubrycki

**Zaawansowane programowanie w C++**

**Dokumentacja końcowa**

**Implementacja wzorca aktywnego obiektu**

1. Warunki testowania:

* Platforma: Linux, g++ (4.6.3) oraz Windows, Visual C++ (2010)
* Język programowania: C++
* Używane biblioteki: boost (1.53.0)

Projekt się kompiluje i działa poprawnie.

1. Możliwości napisanej biblioteki

Podczas implementacji biblioteki staraliśmy się zapewnić użytkownikowi jak największą funkcjonalność i jednocześnie prostotę w korzystaniu z niej.

Z jednej strony chcieliśmy dać użytkownikowi możliwość tworzenia obiektów potrafiących wywoływać coś więcej niż void funkcja(), tzn. żeby nasze asynchroniczne wywołania potrafiły zwrócić wartość możliwie dowolnego typu i przyjmowały jako argumenty możliwie dużą i dowolną liczbę argumentów także dowolnego typu.

Z drugiej strony dobrze jest, gdy użytkownik takiej biblioteki nie będzie musiał pisać dodatkowych setek linii kodu, by wyspecjalizować szablon aktywnego obiektu.

Takie zachowanie udało się nam w dużej mierze osiągnąć. Użytkownik musi utworzyć jedynie dwie własne klasy:

* *Servanta* definiującego zachowanie obiektu - wystarczy, że napisze w nim metody, które *Servant* ma wykonywać w najbardziej naturalnej formie, np. double Add(double a, double b). Wewnątrz takich metod można informować o ich postępie wywołując metodę setProgress(double).
* *Proxy*, udostępniające metody o identycznych listach argumentów, ale zwracające obiekty *Future*, np. Future<double> Add(double a, double b). Ciało takiej metody składa się z zaledwie jednej linii, zależącej od liczby i typów parametrów.

Sprawia to, że specjalizacja aktywnego obiektu jest bajecznie prosta. Przykładowe specjalizacje znajdują się w plikach „Example\*.hpp”. Dokładniejszy opis tego procesu jest w pliku readme i doxygenie.

Inne możliwości to m.in.:

* *Future<void>* zachowuje się jak *Future<bool>* - getValue() zwraca true, jeśli metoda została wykonana (lub ew. czeka, aż się ona wykona).
* Można sprawdzać postęp aktywnie (na obiekcie *Future* wywołać getProgress() ), lub podpiąć do *Future* obiekt funkcyjny, który zachowuje się jak void(double), np. odpowiednią instancję boost::function. Taki **obserwator** będzie reagował na każdą zmianę postępu w zadany sposób.
* Można wykonywać na *Future* operacje przypisania („=”) i tworzyć nowe *Future* w oparciu o już istniejące (cconstructor). Czyli – na wynik jednej metody może czekać kilka obiektów Future, każdy z nich może mieć np. inną funkcję reagującą na zmianę postępu.
* Na „niedokonanych” *Future* można wywołać zatrzymanie (metoda cancelRequest() ). Jeśli użytkownik pisząc swoją metodę przewidział sprawdzanie tego warunku, to:
  + - Jeśli jest to jedyny *Future* „patrzący” na tę metodę, jej wywołanie zostanie przerwane;
    - Jeśli są inne *Future*, to metoda nie przestanie się wykonywać, ale do jej wyników nie będzie się można dostać przez „skasowany” *Future*.
* Wywołanie metody na *Proxy* bez przypisania jej wyniku do *Future* nie powoduje jej skasowania – metoda będzie czekała w kolejce i w końcu się wykona, i nie będzie możliwości przerwania jej (bo już nie ma do niej żadnego „uchwytu” w postaci *Future*). Niemniej, tuż po wykonaniu wynik jest czyszczony (tzn. *MethodRequest* „po sobie sprząta”).
* Tworząc *Proxy* możemy wybrać, ile wątków ma je obsługiwać. Poza tym można wybrać wytyczną sposobu korzystania z *Servanta*, np. każdy wątek ma swojego *Servanta*, wszystkie wątki korzystają z jednego *Servanta*, który zapewnia synchronizację itp. Inne sposoby obsługi *Servanta* można dodać, projektując odpowiednią wytyczną.
* Każdy *MethodRequest* (żądanie czekające na wykonanie) może dysponować metodą guard(), którą należy zaimplementować w *Servancie*. Jest ona bezargumentowa i zwraca bool-a, który, jeśli wynosi true, wstrzymuje wykonanie metody.
* Jeśli wszystkie żądania w kolejce są zablokowane (vide guard), to wątki wykonawcze czekają na jedno z dwóch zdarzeń:
* Dodanie nowej metody do kolejki,
* lub sygnał od specjalnego wątku *Proxy*, który może periodycznie wysyłać rozkaz sprawdzenia guardów metod (np. sprawdzanie, czy jest dostępna pamięć może nie odbywać się tylko przy dodaniu nowego żądania (co jest mało przydatne), ale np. co 5 sekund.) Interwał sprawdzania może być przekazany do *Proxy* w konstruktorze.
* Jeśli podczas wywołania metody zostanie rzucony wyjątek, to zostaje on przechwycony i zapisany „do wglądu” poprzez obiekt *Future*. Zostanie też rzucony ponownie, jeśli wywołamy na takim *Future* getValue().
* Future udostępniają operatory rzutowania na swój typ, tzn. możliwe jest napisanie np. Future<int> x; int y=12-x;. Wykonanie rzutowania zostanie ew. zablokowane do czasu wykonania metody „w tle”.

1. Ograniczenia i sposoby radzenia sobie z nimi

* Implementacja Future korzysta z boost::any – narzuca to warunek, że typy zwracane muszą udostępniać konstruktor kopiujący. Obejście tego ograniczenia jest możliwe np. poprzez użycie sprytnych (lub zwykłych, jeśli ktoś bardzo lubi) wskaźników, tzn. metoda Servanta zwraca sprytny wskaźnik. Także przez boost::any nie udało się zaimplementować domyślnej konwersji między typami, np. Future<int> i; Future<double> d(i); Jest to możliwe do zaimplementowania (pierwsza nasza myśl to Future o dwóch parametrach szablonu: typie „widzianym” i typie „rzeczywistym”), ale zdecydowaliśmy się nie ruszać tej kwestii, jest ona raczej trzeciorzędna.
* Żądania metod, metody guard i obserwatorzy progresu są oparci o boost::function i boost::bind, co ogranicza liczbę argumentów wywołań do 9 argumentów. Ponownie, argumenty powinny udostępniać cconstructor (można to obejść albo jak wyżej, albo używając np. boost::ref). Ograniczenie na ich liczbę jest możliwe do obejścia np. używając wzorca *parameter object*.
* Metody *Servanta* mogą być albo niezsynchronizowane (wtedy każdy wątek wykonawczy powinien mieć swoją kopię Servanta, chyba że faktycznie metody są niezależne), albo zsynchronizowane w pełni (na jednym muteksie) – wtedy wszystkie wątki korzystają po kolei z jednego *Servanta* (mija się to z celem, ale jest możliwe). Jeśli część metod *Servanta* będzie synchronizowana na jednym muteksie, a część na innym, to może dojść do pomieszania (np. wywołanie cancel na pewnym *Future* może przerwać wykonywanie złej metody). Tu nie ma prostego rozwiązania – należałoby prawdopodobnie rozbić obiekt *Servanta* na dwa podobiekty (i mieć do nich oddzielne *Proxy*).

1. Konfrontacja z dokumentacją wstępną – stopień spełnienia założonej funkcjonalności

Po testach stwierdzamy, że wszystkie założone wymagania zostały spełnione. Zaimplementowana jest obsługa progressu, wielowątkowość jednego obiektu, metody guard, obiekty future. Niestety (ale – na szczęście – zgodnie z przewidywaniami) nie udało się zaimplementować kolejkowania z priorytetami – „w praniu” okazało się, że w połączeniu z metodą Guard poprawna implementacja byłaby bardzo czasochłonna: zaplanowanie sprawdzania priorytetów jednocześnie z guardami, rozwiązywanie konfliktów „wysoki priorytet, ale słaby guard” itp. skutecznie nas zniechęciło. Niemniej taka implementacja jest możliwa i, zgodnie z sugestią Prowadzącego, aż prosiłaby się o zrobienie z niej wytycznej, czyli klasy kolejki udostępniającej metody Put i Get, ale zachowującej się w różny sposób w zależności od implementacji.

1. Rozwiązania

Ze względu na wielowątkowość i zwartość projektu, a także na jego charakter typu „wiele małych trybików, które działają tylko razem i na raz” zdecydowano się wspomóc testowanie autorską klasą loggera. Wyświetlanie komunikatów zostało zaimplementowane tak, by zapewnić bezpieczeństwo ze względu na wątki. Oprócz drukowania „zwykłych” wiadomości do funkcjonalności loggera należy wypisywanie id wątku i aktualnego czasu, a także kolorowanie wyjścia w zależności od klasy, która pisze. W połączeniu z odpowiednią „gadatliwością” pozwoliło to prześledzić, co dzieje się w różnych wątkach i szybko odnajdywać problemy.

Tam, gdzie to możliwe, z czasem zautomatyzowano testy, co wymagało odpowiedniego dostosowania makr typu BOOST\_CHECK (brak synchronizacji z loggerem prowadził do wyścigu o konsolę przez logger i BOOST\_CHECK, prosta synchronizacja prowadziła do zakleszczeń). Dostosowane makro można obejrzeć w pliku testującym jako ciekawostkę.

Użycie bibliotek boost było trochę większe niż zaplanowano: oprócz function i bind skorzystano z any (co znacznie uprościło część Future-ową), signals (do implementacji informowania Future’ów o postępie); nieocenione okazały się sprytne wskaźniki (choć w niektórych miejscach zdecydowaliśmy się użyć staroszkolnych \* i samodzielnie je czyścić). Przy zapisywaniu obiektu wyjątku do wglądu przez Future w sukurs przyszła klasa boost::exception\_ptr. W loggerze przydały się też biblioteki obsługujące czas.

Oprócz tego praktycznie cały projekt jest oparty na szablonach (z tego powodu, podobnie jak w boost’cie, zdecydowano się na formę biblioteki nagłówkowej). Takie rozwiązanie umożliwiło znaczną elastyczność zastosowań.

Pozwoliliśmy sobie też na małe roszady w hierarchii klas – zdecydowaliśmy się na kolejkę zabezpieczoną pod kątem wielu czytelników i pisarzy, co pozwoliło na utworzenie nie jednego, a wielu Schedulerów (każdy ma swój wątek i swojego Servanta), którzy jednocześnie patrzą na kolejkę i wyjmują z niej kolejne żądania. Pisarzem kolejki jest z kolei samo Proxy. To pociągnęło za sobą kolejne drobne zmiany, np. przeniesienie odpowiedzialności za sprawdzanie Guardów do klasy kolejki (ale uwaga: odbywa się to co prawda w obiekcie kolejki, ale w wątku Schedulera!).

Spodziewaliśmy się, że zastosowanie boost::bind i boost::function pozwoli na całkowite pozbycie się Servanta, ale okazało się, że połączenie obu rozwiązań pozwala fantastycznie zminimalizować wysiłek przy konkretyzacji aktywnego obiektu.

1. Odczucia subiektywne, wnioski

Największym mankamentem procesu tworzenia niniejszego projektu było (oczywiście) złe rozplanowanie czasu. Poświęciliśmy go zbyt dużo na planowanie klas „na kartce”, przerywając „bo to się nie uda”, a „to jest nieeleganckie” i zaczynając od nowa. Już po rozpędzeniu się zauważyliśmy, że znacznie lepiej sprawdza się wielokrotne napisanie bzdury połączone z przebudową i refaktoryzacją – można doświadczyć, co działa, a co nie i sprawniej wymyślać nowe rozwiązania. Zwłaszcza przy tego typu, biblioteczno-wzorcowym projekcie znacznie więcej czasu zajęło wymyślenie sprawnego i dobrze zazębiającego się modus operandi niż sama jego implementacja.

Podobnym błędem było zbyt późne sprawdzenie działania na innej platformie – po miesiącu radosnego programowania w Visual Studio przesiadka na gcc ze stosunkowo dużą ilością działającego kodu była dość bolesna.

Zdajemy sobie też sprawę, że w kwestii testowania w bardzo małym stopniu skorzystaliśmy z feerii możliwości, jaką oferuje boost::test. Sądzimy jednak, że własny logger połączony z odpowiednimi funkcjami testującymi radził sobie równie dzielnie.

Ze względu na niewielki rozmiar zespołu zdecydowano się też na niekorzystanie ze specjalizowanych serwisów typu Asana. Doszliśmy do wniosku, że w dwuosobowym zespole byłoby to niepotrzebnym skomplikowaniem, zwłaszcza, że nie było problemem skorzystanie z dynamiczniejszych form komunikacji (np. hangouty na Google+).