ZPR

Dokumentacja końcowa projektu:

„Problem komiwojażera”

Skład zespołu:
Bartosz Stalewski
Bartłomiej Stefański
Rafał Witowski

1. Wynik pracy.

1.1. Zrealizowana funkcjonalność:

Zespołowi udało się zrealizować część założonych funkcjonalności, zrealizowano:

- możliwość tworzenia nowych zbiorów miast do odwiedzenia i usuwania istniejących
- zapamiętanie utworzonego miasta w bazie i późniejsze ewentualne modyfikowanie go
- dodawanie i usuwanie miast ze zbioru
- wyznaczenie rozwiązania bliskiego optymalnemu problemu komiwojażera

1.2. Niezrealizowana funkcjonalność:

- Możliwość wyboru parametru określającego ilość wykonanych obliczeń w celu wybrania pożądanej dokładności
- Udostępnienie użytkownikowi możliwości obserwacji postępu obliczeń
- Anulowanie trwającego zadania

Rezygnacja z tych funkcjonalności była spowodowana zmianą sposobu obliczania wyniku(zmiana wykorzystywanej biblioteki w module kalkującym z faif na Boost.Graph).

2. Problemy napotkane podczas tworzenia aplikacji.

Zespołowi nie udało się uniknąć, tak powszechnego, zjawiska niedoszacowania wymaganego czasu pracy nad projektem. W założeniu stopień zaangażowania poszczególnych członków miał być jak najbardziej równomierny w czasie pracy nad programem, jednak okazało się, że wobec nowych komplikacji i zmian w modułach, prognozowany czas zakończenia prac był nierealny. Na szczęście przyjęty na początku

zapas czasu wystarczył na ukończenie prac. Pomógł również rozsądny wybór funkcjonalności realizowanych przez aplikację(część niezrealizowana przez zmianę sposobu obliczeń).
Niedogodnością okazał się również przyjęty model prowadzenia projektu, w wyniku czego terminy zakończenia kolejnych etapów pracy przesuwały się w czasie. Zastosowanie jednej z metodyk zwinnych(np. scrum) usprawniłoby współpracę między członkami grupy oraz przyspieszyło realizację zadania, w szczególności w projekcie, w którym założenia były często zmieniane.
Kolejnym wyzwaniem było to, że grupa podjęła się wykonania aplikacji w technologiach, z jakimi do tej pory miała niewielką lub żadną styczność: Adobe Flex, python, django. Spowodowało to dołożenie dodatkowego obciążenia czasowego.

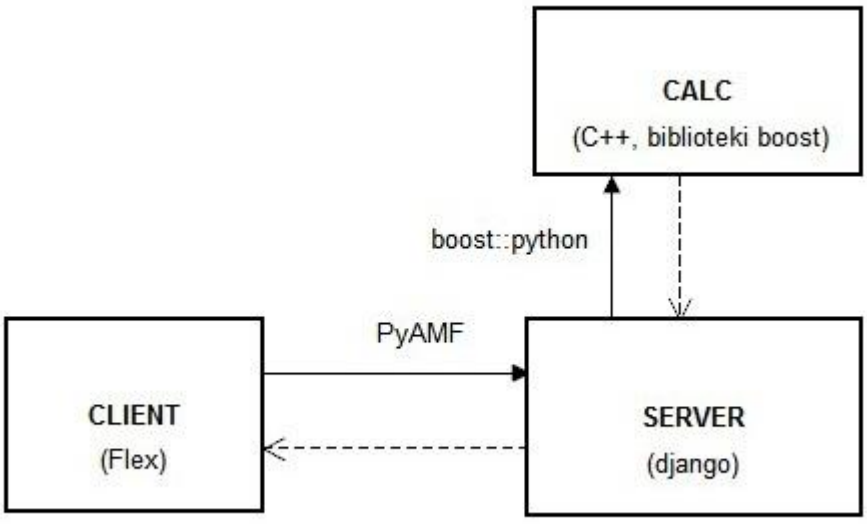
3. Struktura aplikacji.

3.1. Schemat modułów aplikacji.

Aplikacja została podzielona na kilka modułów, aby ułatwić zaprojektowanie jej i zapewnić jak największą niezależność kodu znajdującego się w poszczególnych modułach od reszty. Zwiększa to także elastyczność, gdyż wymieniając część z nich można zmienić / dodać funkcjonalność dostarczaną przez aplikację.
Projektowane oprogramowanie zostało podzielone na 3 moduły:

- Client – odpowiedzialny za interfejs graficzny oraz odbieranie poleceń od użytkownika
- Server – zajmujący się odbieraniem zadań od Klienta, kolejkowaniem ich oraz przekazywaniem żądań do części obliczeniowej
- Calc – moduł obliczający wyniki dla otrzymanych od Klienta(za pośrednictwem Servera) żądań

Schemat ten jest przedstawiony na poniższym rysunku.



Rys. 1 Podział aplikacji na moduły.

3.2. Hierarchia klas modułu Calc.

4. Wykorzystane biblioteki z kolekcji boost.

- Boost.Graph – opis połączeń pomiędzy miastami, obliczanie najkrótszej ścieżki w zadanym grafie
- Boost.Test – testy jednostkowe i modułowe w modułach Server i Calc
- Boost.Python – współpraca między C++ a Pythonem w module Calc
- Smart Pointers – użycie bezpieczniejszych wskaźników od wskaźników wbudowanych
- Boost.Thread – wykorzystanie wątków do zrównoleglenia obliczeń, zapewnienia współpracy z wieloma klientami na raz
- boost::bind – funktory przekazywane wątkom

Oprócz bibliotek z kolekcji boost wykorzystane zostały także te z kolekcji standardowej, np. kontenery(queue, vector), iteratory

5. Użyte wzorce projektowe.

Architektura projektowanego systemu okazała się na tyle prosta, że założenie polegające na zastosowaniu jak największej ilości wzorców projektowych okazało się zbyteczne, ponieważ postanowiono nie używać ich „na siłę, gdzie się tylko da". W większości wypadków wystarczyły zwykłe rozwiązania.

Lista wykorzystanych wzorców projektowych:

- proxy – sprytne wskaźniki
- singleton – klasa zarządzająca przydziałem wątków do zadań: TspManager oraz jej odpowiednik w Pythonie: TspManagerPy
- komenda – funktory przekazywane wątkom przez boost::bind

6. Testy.

6.1. Ręczne uruchamianie testów.

W modułach Server oraz Calc znajdują się pliki z testami odpowiednio: server/traveler/tests.py oraz calc/tests/test.cpp. Można uruchomić je ręcznie lub z poziomu SConsa(o tym w następnym punkcie). Ręczne uruchomienie testów wymaga wcześniejszego zbudowania modułu Calc(o tym w następnym punkcie).

Uruchomienie testów modułu Server wymaga wpisania w konsoli z poziomu katalogu Server polecenia: „python manager.py test", natomiast testów modułowych modułu Calc przez wpisanie w konsoli z poziomu katalogu głównego projektu polecenia: „./testCpp" w przypadku Linuksa i „testCpp.exe" dla Windowsa.

6.2. Sposób tworzenia testów.

Testy były pisane po pojawieniu się stabilnej partii kodu(takiej, która miała się zmienić w niewielkim stopniu). Automatyzacja procesu testowania(wykorzystanie biblioteki Boost.Test) pozwoliła zaoszczędzić dużo czasu. Ułatwiło to kontrolowanie poprawności

aplikacji podczas dodawania nowych funkcjonalności(testy regresywne)czy zmianom w istniejących i przetestowanych kodach źródłowych.

7. Kompilacja i uruchamianie aplikacji.

Do budowy aplikacji zostało użyte narzędzie SCons(plik SConstruct ze skryptem użytym do budowy aplikacji znajduje się w głównym katalogu projektu).

Proces budowy aplikacji:

1) Przejść w konsoli do głównego katalogu projektu.
2) Wpisać polecenie „scons" – budowa modułu Calc i utworzenie testów
3) Wpisać polecenie „scons test=1" – uruchomienie testów(oczywiście ten krok nie jest niezbędny).
4) Wpisać polecenie „scons run=1" – uruchomienie serwera

8. Wymagania.

Aplikacja może być uruchomiona pod systemem Windows lub Linux (pozostałe platformy nie są wspierane i próba kompilacji zakończy się komunikatem odmownym).

Pozostałe wymagania:

a) środowisko:
- środowisko Visual Studio C++ 2010(Windows) lub gcc 4.4 lub nowszy(Linux)
- python 2.6
- AdobeFlex SDK 4.0

b) biblioteki:
- boost 1.44 lub nowsza (C++)
- django 1.2.4 (Python)
- PyAMF 0.6 (Python)
- SCons (kompilacja)

9. Podsumowanie.

Zespół uważa, że wybrany temat pozwolił jego członkom poszerzyć wiedzę w obszarach, w których mieli niewielki doświadczenie. Wewnętrznie narzucone wymogi na jakość kodu pozwoliły utrwalić dobre praktyki programistyczne, zastosowanie wzorców projektowych nauczyło podstaw projektowania architektury tworzonych programów. Zespół jest przekonany, że wiedza przyswojona podczas pracy z nowo poznanymi narzędziami(np. Scons, Mercurial) technologiami(Python, Flex, django) będzie w przyszłości ponownie wykorzystana w następnych projektach.