POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH

PRACA INŻYNIERSKA

Zarządzanie zadaniami w systemie obrazowania wielospektralnego

Task management for hyperspectral imaging system

AUTOR:
Aleksander Cieślak

PROWADZĄCY PRACĘ:
dr inż. Tadeusz Tomczak

OCENA PRACY:

WROCŁAW, 18 listopada 2016

Spis treści

Spis rysunków

Spis tabel

Rozdział 1

Cel projektu

Celem niniejszej pracy jest projekt i implementacja modułu zarządzania zadaniami dla systemu Gerbil (`http://gerbilvis.org/`). Jest to system do analizy i wizualizacji danych wielospektralnych. Gerbil posiada zestaw wielu algorytmów przetwarzania obrazów oraz uczenia maszynowego, które przekładają się na szerokie spektrum funkcjonalności. Jednak jego słabym punktem jest warstwa zarządzania danymi oraz potok przetwarzania danych. To z kolei powoduje niestabilność całej aplikacji. W ramach pracy dyplomowej został zaproponowany system, który rozwiązuje wyżej wspomniane problemy. System ten pozwala na bezpieczny dostęp do danych w całej aplikacji oraz gwarantuje zachowanie właściwego potoku przetwarzania danych.

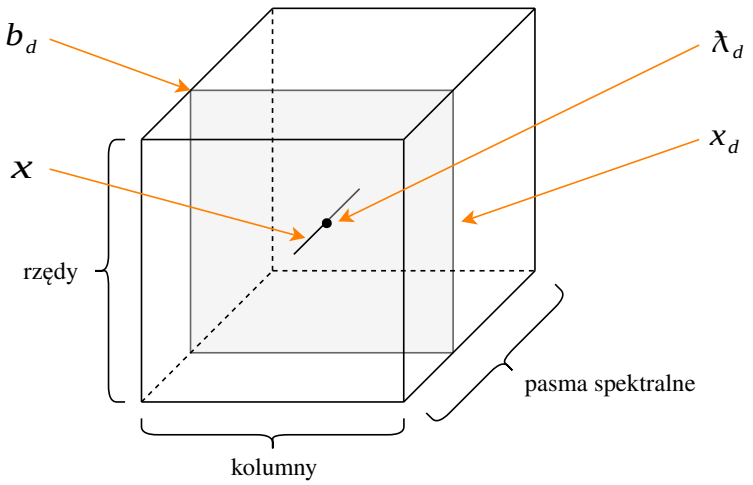
Rozdział 2

Obrazowanie wielospektralne

Obrazowanie wielospektralne jest techniką rejestracji obrazu za pomocą fal elektromagnetycznych o wybranej częstotliwości spośród widma spektroskopowego. Podczas gdy ludzkie oko widzi w głównie w trzech zakresach spektralnych (czerwonym, niebieskim oraz żółtym), obraz wielospektralny jest rejestrowany w znacznie większej liczbie zakresów (przykładowo 31).

2.1. Format danych

Dane wielospektralne są często nazywane kostką wielospektralną.



Rys. 2.1: Schemat kostki wielospektralnego

Na rysunku 2.1 zilustrowano układ danych w kostce wielospektralnej. Kostka taka składa się z n_x pikseli x. Każdy piksel jest wektorem współczynników spektralnych o długości n_D, gdzie n_D jest liczbą obrazów spektralnych, na które składa się dana wielospektralna. Każdy współczynnik x_d jest wartością reakcji sensorycznej dla odpowiadającego pasma spektralnego b_d skoncentrowanego wokół fali λ_d. W skrócie obraz wielospektralny jest zbiorem obrazów rejestrowanych przy użyciu fal elektromagnetycznych o zadanych długościach.

2.1.1. Konsekwencje formatu danych

Ze względu na swoją charakterystykę obrazy wielospektralne mogą bezproblemowo osiągać rozmiary setek megabajtów, lub nawet gigabajtów. Większość danych pochodnych, które są efektem analizy tego obrazu posiadają podobne rozmiary. Informacja ta jest kluczowa podczas projektowania mechanizmu zarządzania danymi w takim systemie. Biorąc pod uwagę rozmiar danych mechanizm taki powinien:

- unikać tworzenia zbędnych kopi danych,
- dokonywać obliczeń danych wyłącznie na żądanie,
- zwalniać z pamięci dane, które nie są już wykorzystywane przez aplikację.

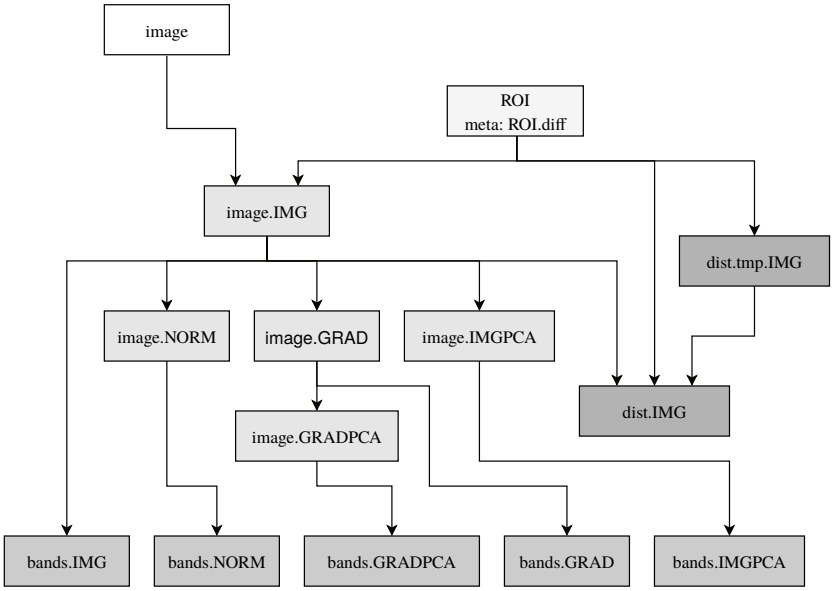
2.2. Dane w systemie Gerbil

Oryginalny obraz wielospektralny jest traktowany jako dana wejściowa w systemie. Na jego podstawie powstają dane pochodne. Są to głównie kolejne obrazy oraz histogramy wielospektralne. Do stworzenia prototypu mechanizmu zarządzania danymi oraz procesem przetworzenia użyte zostały poniższe dane:

- **image** – oryginalny obraz wielospektralny. Dana ta jest obliczana podczas inicjalizacji aplikacji. Użytkownik może wejść w interakcję z systemem dopiero gdy image zostanie przetworzone.
- **ROI (Region of Interest)** – wyselekcjonowany podzbiór danych, w tym przypadku wybrane prostokątne zaznaczenie obrazu. Jest przechowywany jako współrzędne lewego górnego wierzchołka zaznaczenia, jego wysokość oraz szerokość,
- **image.IMG** – fragment obrazu oryginalnego zdeterminowany przez ROI,
- **image.NORM** – image.IMG po normalizacji wektorów składających się z pikseli o jednakowych współrzędnych na przestrzeni pasm spektralnych,
- **image.GRAD** – gradient obrazu image.IMG,
- **image.PCA** – image.IMG po zastosowaniu metody PCA (analizy głównych składowych),
- **image.GRADPCA** – image.GRAD po zastosowaniu metody PCA,
- **bands.*.N** – pojedynczy N-ty obraz spektralny danej reprezentacji image.* (przykładowo bands.NORM.6),
- **dist.IMG** - histogram wielospektralny obrazu image.IMG,
- **dist.tmp.IMG** - dana pomocnicza używana do uzyskania danej dist.IMG.

Z racji, że jedne dane produkują inne, łatwo jest zdefiniować hierarchię danych w tym systemie.

Na rysunku 2.2 przedstawiono diagram zależności danych. Dane jednego koloru są do siebie semantycznie zbliżone. Przykładowo, image.NORM, image.GRAD, image.GRADPCA itp. są reprezentacjami obrazu oryginalnego. Dane posiadają również swoje metadane. Przykładowo metadaną ROI jest ROI.diff, które określa różnicę pomiędzy aktualnym a poprzednim ROI.



Rys. 2.2: Graf zależności danych w systemie Gerbil

2.2.1. Wpływ hierarchii danych na proces wykonania

Analizując rysunek 2.2 można dojść do wniosku, że proces przetworzenia danych jest dyktowany poprzez ich hierarchię. Przykładowo, do obliczenia image.GRADPCA wymagane jest aby dane image, ROI, image.IMG oraz image.GRAD były już przetworzone. Dodatkowo można określić porządek, w którym te dane powinny zostać obliczone:

1. image (podczas inicjalizacji systemu),
2. ROI,
3. image.IMG,
4. image.GRAD,
5. image.GRADPCA.

Scenariusz ten zakłada obliczenie każdej danej w hierarchii, co jest przypadkiem skrajnym. Często zdarza się, że pewna część danych jest aktualna. Wówczas przetwarzanie powinno rozpocząć się od pierwszej nieaktualnej danej znajdującej się najwyżej w hierarchii.

Dodatkowo należy rozpatrzeć scenariusz równoległego wykonywania zadań. Zakładając, że aplikacja wyświetla jednocześnie dane image.NORM oraz image.GRAD, natomiast image.IMG zostało odświeżone, można dojść do wniosku, że system powinien w następnym kroku dokonać obliczeń obu danych (image.NORM i image.GRAD). Obliczenia te można wykonać szeregowo bądź równolegle, wobec tego można zdefiniować opcjonalne wymaganie dla systemu zarządzania zadaniami jako możliwość równoległego przetwarzania zadań.

Rozdział 3

Technologie wykorzystane w systemie Gerbil

3.1. C++

System Gerbil jest rozwijany w języku C++. Jest to język programowania ogólnego przeznaczenia, ze szczególnym zastosowaniem w tworzeniu systemów. C++ to język:

- wieloparadygmatowy – pozwala na programowanie proceduralne, obiektowe, funkcyjne oraz ogólne,
- statycznie typowany – zgodność typów jest sprawdzana w trakcie kompilacji,
- pozwalający na bezpośrednie zarządzanie pamięcią,
- tworzony według zasady zerowego narzutu - elementy tego języka oraz proste abstrakcje muszą być optymalne (nie marnować bajtów pamięci ani cyklów procesora),
- umożliwiający tworzenie lekkich i wydajnych abstrakcji [1] [2].

Język o takiej charakterystyce jest dobrym wyborem do implementacji systemu analizy i wizualizacji skomplikowanych danych.

3.1.1. STL

STL (ang. Standard Template Library) jest biblioteką standardową języka C++. Oferuje ona szereg kontenerów, klas, obiektów funkcyjnych oraz algorytmów. Składniki te opisane są w standardzie ISO języka C++, oraz gwarantują identyczne zachowanie w każdej implementacji [3]. Ułatwia to tworzenie aplikacji wieloplatformowych.Dzięki gotowym rozwiązaniom zawartym w bibliotece standardowej, proces wytwarzania oprogramowania zyskuje na prostocie i efektywności.

[1] Stroustrup B., Język C++. Kompendium wiedzy, Wydawnictwo Helion, Gliwice, 2014.
[2] Krótki opis języka C++ `http://www.cplusplus.com/info/description` (dostęp 31.10.2016)
[3] Stroustrup B., Język C++. Kompendium wiedzy, Wydawnictwo Helion, Gliwice, 2014.

shared_ptr

shared_ptr jest typem umożliwiającym reprezentację własności wspólnej. Wykorzystywany jest w sytuacjach, gdy dwa (lub więcej) fragmenty kodu wymagają dostępu do danych, podczas gdy żaden nie jest odpowiedzialny za usunięcie tych danych. Obiekt shared_ptr jest rodzajem wskaźnika z licznikiem wystąpień. Jeśli liczba obiektów wskazujących na konkretną daną spadnie do zera, dana ta jest usuwana [4].

mutex

Muteks jest obiektem typu mutex, służącym do reprezentowania wyłącznych praw dostępu do konkretnego zasobu. Wykorzystuje się go do ochrony przed wyścigami do danych oraz synchronizacji dostępu do dancyh współdzielonych między wątkami.

Muteks może być w posiadaniu tylko jednego wątku na raz. Zajęcie muteksu jest równoznaczne z nabyciem wyłącznych praw własności do niego. Operacja zajmowania muteksu jest blokująca. Zwolnienie muteksu oznacza zrzeczenie się z prawa własności do niego. Daje to możliwość zajęcia muteksu przez inne oczekujące wątki [5].

condition_variable

3.2. Qt

Qt jest platformą deweloperską wyposażoną w narzędzia pozwalające usprawnić proces wytwarzania oprogramowania oraz interfejsów użytkownika dla aplikacji desktopowych, wbudowanych bądź mobilnych [6].

Platforma Qt posiada szerokie spektrum funkcjonalności. Między innymi są to:

- system meta-obiektów,
- mechanizm sygnałów i slotów służący do komunikacji pomiędzy obiektami,
- wbudowany system przynależności obiektów,
- wieloplatformowe wsparcie modułu wielowątkowości.

W systemie Gerbil jest wykorzystywane Qt w wersji 5.7.

3.2.1. Sygnały i sloty

Spośród rozrzerzeń języka C++, jakie oferuje Qt, na szczególną uwagę zasługuje mechanizm sygnałów i slotów. Dzięki niemu możliwe jest skomunikowanie dwóch dowolnych obiektów w sposób alternatywny do użycia wywołań zwrotnych.

Sygnał jest wysyłany, gdy nastąpi jakieś zdarzenie (np. naciśnięcie przycisku przez użytkownika), natomiast slot jest odpowiedzią na ten sygnał. Sygnatury sygnału i slotu muszą być

[4] Stroustrup B., Język C++. Kompendium wiedzy, Wydawnictwo Helion, Gliwice, 2014.
[5] Stroustrup B., Język C++. Kompendium wiedzy, Wydawnictwo Helion, Gliwice, 2014.
[6] Dokumentacja Qt 5.7 http://doc.qt.io/qt-5/index.html (dostęp 30.10.2016).

zgodne. Mechanizm ten jest luźno powiązany (ang. loosely coupled). Oznacza to, że klasa emitująca sygnał nie musi być swiadoma klasy odbierającej. Sygnały i sloty pozwalają na przekazanie dowolnej liczby argumentów dowolnego typu. Sygnały muszą zostać zadeklarowane po słowie kluczowym `signals`. Z jednym slotem można połączyć dowolną ilość sygnałów, i odwrotnie – z jednym sygnałem można skojarzyć dowolną ilość slotów.

Wszystkie klasy korzystające z tego mechanizmu muszą w swojej deklaracji zawierać makro `Q_OBJECT` oraz dziedziczyć (bezpośrednio bądź pośrednio) po klasie `QObject` [7].

Składnia

Sposób tworzenia połączeń zostanie zilustrowany na przykładzie. Za punkt wyjścia posłużą dwie klasy: `Sender` (Listing 3.1) oraz `Receiver` (Listing 3.2).

Listing 3.1: Klasa Sender

```
1  class Sender : public QObject
2  {
3      Q_OBJECT
4  public:
5      explicit Sender(QObject *parent = 0) : QObject(parent) {}
6
7  signals:
8      void sendMessage(QString msg);
9
10 };
```

Listing 3.2: Klasa Receiver

```
1  class Receiver : public QObject
2  {
3      Q_OBJECT
4  public:
5      explicit Receiver(QObject *parent = 0) : QObject(parent) {}
6
7      void receiveMessageMethod(QString msg) {
8          std::cout << "got message in method: " << msg;
9      }
10
11 public slots:
12     void receiveMessageSlot(QString msg) {
13         std::cout << "got message: " << msg;
14     }
15 };
```

[7] Dokumentacja Qt 5.7. Sygnały i Sloty `http://doc.qt.io/qt-5/signalsandslots.html` (dostęp 30.10.2016).

Z analizy listingów 3.1 oraz 3.2 wynika, że klasa `Sender` zawiera sygnał `sendMessage`, natomiast klasa `Receiver` zawiera publiczną metodę `receiveMessageMethod` oraz publiczny slot `receiveMessageSlot`.

W Qt występują dwa rodzaje składni pozwalające na ustanowienie połączenia. Jedna z nich (starsza) pozwala na ustanowienie połączenia jedynie pomiędzy sygnałem a sygnałem, bądź sygnałem a slotem. Drugi rodzaj składni, wprowadzony w Qt5, pozwala dodatkowo na nawiązanie połączenia pomiędzy sygnałem a metodą klasy. Na listingu 3.3 przedstawiony jest zarówno stary jak i nowy zapis.

Listing 3.3: Składnia tworzenia połączeń między obiektami

```
1  Sender sender;
2  Receiver receiver;
3
4  //stara skladnia
5  //poprawne
6  QObject::connect(&sender, SIGNAL(sendMessage(QString)), &receiver,
       ↳ SLOT(receiveMessageSlot(QString)));
7  //niepoprawne
8  QObject::connect(&sender, SIGNAL(sendMessage(QString)), &receiver,
       ↳ SLOT(receiveMessageMethod(QString)));
9
10 //nowa skladnia
11 QObject::connect(&sender, &Sender::sendMessage, &receiver,
       ↳ &Receiver::receiveMessageMethod);
12 QObject::connect(&sender, &Sender::sendMessage, &receiver,
       ↳ &Receiver::receiveMessageSlot);
```

Mechanizm sygnałów i slotów jest powszechnie wykorzystywany w systemie Gerbil do ustanownienia komunikacji pomiędzy obiektami. Używana jest zarówno stara jak i nowa składnia.

3.2.2. Wątek GUI oraz wątki robocze

GUI (ang. Graphical User Interface) jest graficznym interfejsem użytkownika. Każda aplikacja jest uruchamiana w osobnym wątku. Jest on nazywany wątkiem głównym (bądź "wątkiem GUI"w aplikacjach Qt). Interfejs użytkownika rozwijany w Qt musi zostać uruchomiony w tym wątku. Wszystkie widgety oraz kilka klas pochodnych nie zadziałają w wątkach pobocznych. Wątki poboczne są często nazywane "wątkami roboczymi", ponieważ wykorzystywane są aby odciążyć główny wątek od skomplikowanych obliczeń [8]. Gdyby te obliczenia były wykonywane w głównym wątku, aplikacja przestałaby być responsywna na czas obliczeń, co jest efektem niepożądanym.

[8] Dokumentacja Qt 5.7. Wątki: podstawy `http://doc.qt.io/qt-5/thread-basics.html#gui-thread-and-worker-thread` (dostęp 30.10.2016).

3.3. Boost

Boost jest kolekcją bibliotek do języka C++. Biblioteki te poszerzają funkcjonalności tego języka [9]. Wiele z bibliotek rozwijanych przez Boost zostało włączonych do standardu C++. Z perspektywy systemu Gerbil na specjalną uwagę zasługuje Boost.Any.

3.3.1. Boost.Any

W języku C++ kwestia przechowania obiektów dowolnego typu jest problematyczna, ponieważ jest to język statycznie typowany.

`void*`

W czystym C++ można użyć `void*`. Do zmiennej typu `void*` można przypisać wskaźnik dowolnego typu poza wskaźnikiem do funkcji oraz wskaźnikiem do składowej. Aby użyć takiej zmiennej należy dokonać jawnej konwersji `static_cast`. Do zastosowania `void*` w kodzie wysokopoziomowym należy podchodzić z rezerwą. Może to wskazywać na błędy projektowe. [10]

`boost::any`

Rozwiązaniem, które z powodzeniem można stosować w kodzie wysokopoziomowym jest właśnie klasa `boost::any`. Jego zdecydowaną przewagą nad `void*` jest bezpieczny typowo interfejs. Jest to kontener opakowujący pojedynczy obiekt niemal dowolnego typu (obiekt musi być `copy-constructible` - posiadać możliwość inicjalizacji na bazie innego objektu tego typu). Aby użyć obiektu przechowywanego przez `boost::any` należy dokonać rzutowania `boost::any_cast`. Jeżeli zostanie podany typ, na który obiekt nie może zostać zrzutowany, zostanie zgłoszony wyjątek `boost::bad_any_cast` [11] [12].

3.4. Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) jest biblioteką szablonów ułatwiającą programowanie równoległe. W swojej ofercie posiada gotowe struktury danych oraz zrównoleglone algorytmy [13].

W systemie Gerbil biblioteka TBB używana jest głownie do implementacji algorytmów przetwarzania obrazów wielospektralnych.

[9] Oficjalna strona Boost `http://www.boost.org/` (dostęp 30.10.2016).

[10] Stroustrup B., Język C++. Kompendium wiedzy, Wydawnictwo Helion, Gliwice, 2014.

[11] Dokumentacja Boost 1.62.0 Boost.Any `http://www.boost.org/doc/libs/1_62_0/doc/html/any.html` (dostęp 30.10.2016).

[12] Spis bibliotek Boost `http://www.boost.org/doc/libs/` (dostęp 30.10.2016).

[13] Oficjalna strona Threading Building Blocks `https://www.threadingbuildingblocks.org/` (dostęp 31.10.2016).

3.5. OpenCV

OpenCV (Open Source Computer Vision Library) to biblioteka przeznaczona do rozpoznawania obrazów oraz uczenia maszynowego [14].

Biblioteka ta znajduje wykorzystanie w systemie Gerbil jako bogata baza struktur danych wykorzystywanych do przetwarzania obrazów oraz zaawansowanych algorytmów rozpoznawania obrazów.

[14] Oficjalna strona OpenCV `http://opencv.org/about.html` (dostęp 31.10.2016).

Rozdział 4

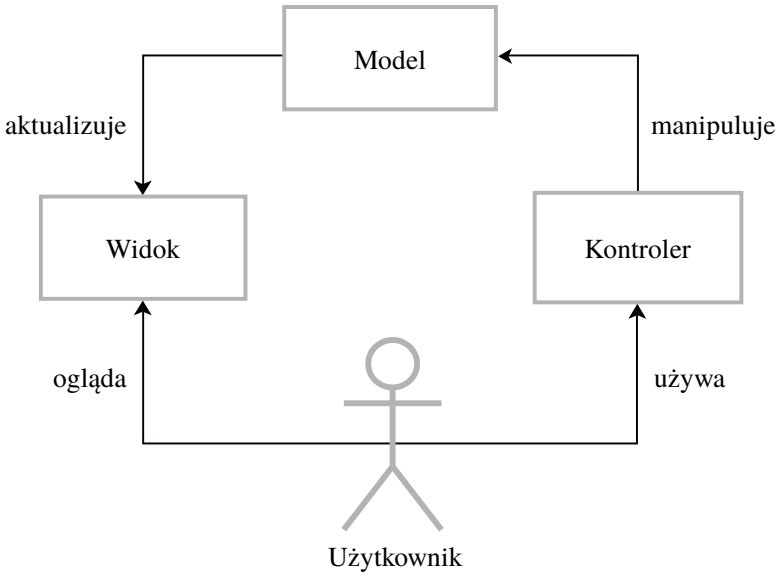
Aktualny stan projektu Gerbil

4.1. Wzorzec MVC

Aplikacja Gerbil jest zaprojektowana według wzorca MVC (Model-View-Controller) z wykorzystaniem platformy Qt. MVC jest wzorcem architektonicznym używanym często do tworzenia interfejsów użytkownika. Podstawą MVC są trzy obiekty:

- model - komponent odpowiedzialny za serwowanie danych,
- widok - komponent odpowiedzialny za wizualizację danych,
- kontroler - komponent definiujący logikę, za pomocą której interfejs użytkownika odpowiada na żądania.

Podział tych ról można zaobserwować na rysunku 4.1.



Rys. 4.1: Podział ról we wzorcu architektonicznym MVC

Dzięki wykorzystaniu tego wzorca sposób przechowywania danych nie ma wpływu na to jak są one przedstawione użytkownikowi [1].

4.2. Architektura aplikacji Gerbil

W aplikacji Gerbil wzorzec MVC zastosowano w sposób klasyczny:

- modele są odpowiedzialne za obliczenia danych oraz sygnalizowanie pojawienia się ich nowej wersji,
- widoki wyświetlają dane,
- kontrolery zajmują się kojarzeniem akcji użytkownika z konkretną funkcjonalnością modelu.

Dodatkowo w aplikacji występuje wątek roboczy. W nim uruchomiona jest kolejka zadań. Zadanie (Task) jest komponentem realizującym wykonanie czasochłonnego algorytmu analizy danych. Modele tworzą zadania i przekazują je do kolejki. Kolejka przyjmuje zadania i wykonuje je po kolei. W ten sposób skomplikowane obliczenia nie blokują wątku GUI, które pozostaje przez cały czas responsywne.

W takiej architekturze pojawia się problem dostępu do danych, ponieważ dwa wątki (wątek GUI, w którym znajdują się komponenty MVC, oraz wątek roboczy, w którym wykonywane są zadania) próbują uzyskać dostęp do tych samych danych. Zadania wykonywane w tle powinny w bezpieczny sposób dokonywać zapisu danych. Widoki zaś powinny być w stanie bezawaryjnie wizualizować dane oraz dbać o aktualność prezentowanych danych.

4.2.1. Wady architektury

System ten jest mocno zdecentralizowany. Na barkach kontrolerów spoczywa odpowiedzialność odpowiedniej propagacji sygnałów informujących o nowej wersji danych, inwalidacji danych, jak również zapytań o dokonanie nowych obliczeń. Prowadzi to do:

- zaciemnienia kodu źródłowego zbędnymi instrukcjami warunkowymi,
- zignorowania pewnych sygnałów,
- podjęcia niewłaściwej decyzji.

Zarządzanie zadaniami również jest wadliwe. W razie gdy użytkownik poprzez interakcję z systemem zleci wykonanie kilku zadań na raz, które dokonują obliczeń na tych samych danych, system może zachować się w sposób nieoczekiwany. Prowadzi to do zakończenia aplikacji z powodu naruszenia pamięci. Nowa wersja systemu powinna w najgorszym wypadku zakolejkować te zadania i wykonać jedno po drugim.

Wiele komponentów interfejsu użytkownika przechowuje własne uchwyty do danych oraz ewentualnie muteks. Wobec tego same dokonują synchronizacji lub nie robią tego wcale. Nieprzemyślany model doprowadził do wielu patologi. Przykład stanowi używanie współdzielonych

[1] Dokumentacja Qt 5.7. Programowanie Model/Widok `http://doc.qt.io/qt-5/model-view-programming.html` (dostęp 30.10.2016).

wskaźników do przekazywania danych, podczas gdy dane te z założenia powinny być współdzielone.

Konkluzja

Aktualny model współdzielonych danych, w powiązaniu z modelem zarządzania nimi nie gwarantuje bezpiecznego dostępu do danych ani prawidłowego przebiegu procesu wykonania zadań. Biorąc pod uwagę wyżej wymienione problemy nowy mechanizm zarządzania danymi oraz procesem wykonania powinien:

- posiadać wewnętrzny mechanizm synchronizacji dostępu do danych,
- gwarantować bezpieczny dostęp do współdzielonych danych,
- gwarantować bezpieczne wykonanie zadań w tle,
- gwarantować prawidłową kolejność procesu przetwarzania danych,
- posiadać scentralizowany mechanizm propagacji sygnałów,
- prawidłowo propagować informację o dostępności nowej wersji danych,
- prawidłowo propagować informację o żądaniu obliczeń nowych danych.

Rozdział 5

Projekt nowego systemu

5.1. Zarys projektu

Nowy system opiera się na komponencie zwanym Subscription Manager (SM). Jest on właścicielem wszystkich współdzielonych danych. Pozostałe komponenty uzyskują dostęp do danych przez obiekty nazywane Subscription (Subskrypcja). Prawa dostępu są przyznawane oraz kontrolowane przez SM. Dane tworzą graf zależności, który zapewnia automatyczne obliczanie potrzebnych danych. Komponenty typu Creator (odpowiedniki modeli we wzorcu MVC) kontrolują proces powstawania danej poprzez tworzenie odpowiednio sparametryzowanych zadań. Utworzone zadania trafiają do komponentu Task Scheduler, który zarządza ich wykonaniem.

5.1.1. Przepływ danych w systemie

Obliczenia danych są wywoływane poprzez interakcję użytkownika z interfejsem graficznym. Dane są wizualizowane w konkretnych panelach. W celu prezentacji danych użytkownik musi aktywować dany panel. Z punktu widzenia systemu panel jest komponentem żądającym dostępu do danej w celu odczytu. W kodzie panela tworzony jest obiekt subskrypcji, który stanowi definicję takiego żądania. Obiekty tego typu stanowią interfejs pomiędzy komponentami żądającymi dostępu do danych a Subscription Managerem. Jeżeli dane są aktualne, panel otrzymuje stosowną informację. W ciele obsługi tej informacji dokonywany jest faktyczny dostęp do danych, dzięki czemu dana zostaje zaprezentowana w GUI. Możliwy jest również scenariusz, w którym żądana dana nie jest aktualna, bądź nie została jeszcze zainicjalizowana. Wówczas system wysyła informację do odpowiedniego komponentu typu Creator, który zadeklarował umiejętność wytworzenia tej danej. Komponent w odpowiedzi na informację tworzy obiekt Zadania oraz przekazuje go do Task Scheduler'a. Każdy obiekt zadania posiada pewne zależności wobec danych. Minimum stanowi żądanie dostępu do danej, która ma zostać obliczona, w celu zapisu. Zazwyczaj zależności dopełniają żądania dostępu do odczytu danych, od których obliczana dana zależy (o ile zależy). Task Scheduler tworzy odpowiednie obiekty subskrypcji dla danego zadania, na podstawie jego sygnatury zależnosci. Akcja ta może prowadzić do powstania kolejnych obiektów zadań (jeżeli zadanie żąda dostępu do danej, która również nie jest aktualna).

Task Scheduler uruchamia powstałe zadania w osobnych wątkach. W momencie gdy dana zarządana przez użytkownika zostanie finalnie obliczona, panel GUI otrzymuje stosowną informację i może dokonać jej prezentacji.

5.2. Model danych współdzielonych

Ważne jest aby do roli danej współdzielonej można było promować każdą daną w systemie. Dlatego model danej współdzielonej nie może opierać się na interfejsie, który inne klasy by implementowały, lecz na opakowaniu, w które można każdą daną włożyć. Klasa danych współdzielonych nazwana została `DataEntry`.

5.2.1. Przechowywanie danych

Kwestię przechowania dowolnego typu można rozwiązać poprzez wykorzystanie `boost::any`. Natomiast problem uchwytu do danych, którego można użyć w różnych fragmentach kodu rozwiązuje `std::shared_ptr`. Z tego powodu `std::shared_ptr<boost::any>` stanowi trzon modelu danych współdzielonych. Zarówno dane jak i towarzyszące im metadane są przechowywane w ten sposób.

Listing 5.1: Aliasy używane w kodzie aplikacji

```
1 using handle = std::shared_ptr<boost::any>;
2 using handle_pair = std::tuple<handle, handle>;
```

Aby kod był bardziej zwięzły stosowane są w nim aliasy (Listing 5.1). Pierwsza linia jest skróceniem zapisu typu uchwytu do danych, natomiast druga skraca zapis pary takich uchwytów (para uchwytów często jest wykorzysytywana do reprezentacji danych zagregowanych z metadanymi).

5.2.2. Synchronizacja dostępu do danych

Rozwiązanie to jednak nie likwiduje problemu synchronizacji dostępu do danych. Wobec tego model został wzbogacony o muteks oraz dwie zmienne warunkowe (Listing 5.2). Pierwsza zmienna warunkowa - `not_reading` służy do obsłużenia wątków oczekujących na dostęp do danych w celu zapisu, natomiast druga `not_writing` do obsługi wątków oczekujących na dostęp do danych w celu odczytu.

Listing 5.2: Składowe klasy `DataEntry` zapewniające bezpieczne użycie w środowisku wielowątkowym

```
1 std::mutex mu;
2 std::condition_variable not_reading;
3 std::condition_variable not_writing;
```

Z pomocą tych narzędzi model udostępnia metody pozwalające na bezpieczny dostęp do danych w aplikacji wielowątkowej.

Listing 5.3: Metody klasy `DataEntry` zapewniające bezpieczny odczyt danych współdzielonych w środowisku wielowątkowym

```
1  handle_pair DataEntry::read()
2  {
3      std::unique_lock<std::mutex> lock(mu);
4      not_writing.wait(lock, [this]() {
5          return !doWrite && initialized;
6      });
7      return handle_pair(data_handle, meta_handle);
8  }
9
10 void DataEntry::endRead()
11 {
12     if (doReads == 0) not_reading.notify_one();
13 }
```

Na listingu 5.3 widoczna jest implementacja metod realizujących dostęp do danych w celu odczytu.

W metodzie `read` tworzona jest blokada, która zajmuje mutex. Następnie wątek wywołujący metodę zostaje uśpione do momentu powiadomienia przez zmienną warunkową. Aby uniknąć fałszywych wybudzeń przekazywana jest dodatkowo lambda, która służy za predykat. Jeśli wartość zwrócona przez lambdę jest prawdziwa (zawarte jest w niej sprawdzenie czy wewnętrzny stan danej jest prawidłowy), wówczas wybudzenie jest słuszne. Po wybudzeniu zostaje zwrócona para uchwytów – do danej oraz metadanej.

Metoda `endRead` służy do sygnalizacji zakończenia odczytu. W jej ciele wykonywane jest sprawdzenie czy wewnętrzny stan danej jest prawidłowy. Jeśli jest, wówczas zmienna warunkowa `not_reading` dokonuje przebudzenia jednego z wątków oczekujących na dostęp do danych w celu zapisu.

Listing 5.4: Metody klasy `DataEntry` zapewniające bezpieczny zapis danych współdzielonych w środowisku wielowątkowym

```
1  handle_pair DataEntry::write()
2  {
3      std::unique_lock<std::mutex> lock(mu);
4      not_reading.wait(lock, [this]() {
5          return doReads == 0 && !doWrite;
6      });
7
8      return handle_pair(data_handle, meta_handle);
9  }
10
11 void DataEntry::endWrite()
12 {
13     if (!doWrite) not_writing.notify_all();
14 }
```

Analizując implementację metod pozwalających na bezpieczny zapis danych przedstawionych na listingu 5.4 można dostrzec dużą analogie do metod z listingu 5.3. Jedyną zasadniczą różnicą jest fakt, że po zakończeniu zapisu zmienna warunkowa `not_writing` budzi wszystkie wątki oczekujące na dostęp w celu odczytu.

5.3. Subscription oraz Subscription Manager

Subscription Manager pełni rolę jednostki głównej w systemie. Jest on abstrakcją o poziom wyższą niż dane współdzielone. Komponent ten agreguje wszystkie takie dane oraz nimi zarządza. Do jego odpowiedzialności należą:

- przyznawanie dostępu do danych,
- propagacja informacji o dostępności nowej wersji danych,
- propagacja informacji o żądaniu obliczeń nowych danych,
- zarządzanie cyklem życia danych,
- kontrola wewnętrznego stanu danych.

5.4. Kreator (Creator)

Kreator jest klasą semantycznie zbliżoną do Modelu z wzorca MVC. Zadaniem kreatora jest kontrola procesu wytworzenia danej. Interfejs zdefiniowany dla kreatora został przedstawiony na listingu 5.5.

Listing 5.5: Interfejs klasy `Creator`

```
1  class Creator : public QObject
2  {
3      Q_OBJECT
4  public:
5      explicit Creator(SubscriptionManager& sm, TaskScheduler* scheduler,
6                       QObject *parent = 0);
7      virtual ~Creator();
8
9  public slots:
10     virtual void delegateTask(QString requestedId,
11                               QString parentId = "") = 0;
12     void taskFinished(QString id, bool success);
13
14 protected:
15     void registerData(QString dataId,
16                       std::vector<QString> dependencies);
17     bool isTaskCurrent(QString id);
18
19 private:
20     SubscriptionManager& sm;
21     TaskScheduler* scheduler;
22
23     std::map<QString, std::shared_ptr<Task>> tasks;
24
25 };
```

Metoda `registerData` służy do rejestrowania danych. Za jej pomocą kreator zgłasza współdzielone dane, za które bierze odpowiedzialność. Każda dana współdzielona, aby istnieć w systemie, musi zostać zarejestrowana przez któryś z kreatorów. Do zarejestrowania danej wymagane jest jej ID oraz lista ID danych, od której owa dana jest zależna. Na listingu 5.6 został przedstawiony przykład rejestrowania współdzielonych. W pierwszych dwóch liniach zarejestrowane zostały dane, które są niezależne (lista ich zależności jest pusta). W trzeciej lini jest wyrażona rejestracja danej `image.IMG` oraz jej zależności od danych `image` oraz `ROI`. Dzięki takiemu formatowi rejestrowania danych system jest w stanie stworzyć graf zależności danych, wykorzystywany do prawidłowej propagacji informacji.

Listing 5.6: Przykłady rejestrowania danych

```
1  registerData("image", {});
2  registerData("ROI", {});
3  registerData("image.IMG", {"image", "ROI"});
```

Z analizy listingu 5.5 wynika, że klasa implementująca interfejs Kreatora musi zdefiniować metodę `delegateTask` aby nie być klasą abstrakcyjną. Metoda ta jest kluczowa dla tego interfejsu, oraz bardzo ważna dla całego systemu. Ciało tej metody powinna stanowić obsługa żądania W

definicji tej metody powinna znaleźć się obsługa żądania dokonania obliczeń danych. Żądanie takie może płynąć bezpośrednio od użytkownika, bądź w sposób pośredni, na skutek wewnętrznego mechanizmu systemu. Standardowym zachowaniem kreatora jest utworzenie odpowiedniego obiektu Zadania i przekazanie go do Task Scheduler'a. Teoretycznie możliwe jest, aby kreator sam dokonał obliczenia danej, zamiast tworzyć obiekt Zadania. Ta metoda jednak nie jest zalecana. Dopuszcza się jej stosowanie jedynie w przypadku nieskomplikowanych obliczeń na małych strukturach danych.

Dodatkowo na uwagę zasługuje metoda `isTaskCurrent`, dzięki której można sprawdzić, czy istnieje aktualnie zadanie o danym id, oczekujące na wykonanie, bądź aktualnie wykonywane. Informacja ta jest pomocna w tworzeniu rozbudowanej logiki kreatora.

5.5. Zadanie (Task)

Zadanie to komponent odpowiadający za bezpośrednie obliczenia danych. Służy zatem do zaktualizowania danej bądź jej inicjalizacji. Podstawową zasadą zadania jest to, że zawsze dokonuje on modyfikacji tylko jednej danej, natomiast może bazować na dowolnej liczbie danych, określanych jako "źródła".

Każde zadanie posiada składową `dependencies`, która jest listą jego zależności. Złożona jest ona z danej modyfikowanej oraz źródeł. Dla każdej danej w liście zależności określony jest również cel dostępu do niej (odczyt bądź zapis).

Id modyfikowanej danej musi zostać przekazane zadaniu jako parametr jego konstruktora. Identyfikatory źródeł również są przekazywane w konstrukorze, lecz w postaci mapy. W mapie tej kluczem jest identyfikator źródła, natomiast wartością identyfikator, według którego ta dana będzie rozróżniana wewnątrz zadania. Jest to zabieg zastosowany w celu ujednolicenia konwencji nazewniczej wewnątrz zadań. Zazwyczaj zadania posiadają jedynie jedno źródło, do którego odnoszą się za pomocą identyfikatora `source`. Id danej modyfikowanej jest automatycznie mapowane na nazwę `dest`.

Zadanie, aby móc realizować dostęp do danych musi posiadać odpowiednie obiekty subskrypcji. Przeznaczona na nie jest składowa `subscriptions`. Komponent nadrzędny, zarządzający zadaniem (Task Scheduler) jest zobowiązany do utworzenia dla niego właściwych obiektów subskrypcji (na podstawie jego zależności, zwracanych przez metodę `getDependencies`) oraz przekazania ich poprzez metodę `setSubscription`. Teoretycznie obiekt zadania może sam tworzyć subskrypcje, jednak nie jest to zalecane, a wręcz uznawane za błąd koncepcyjny. Przeznaczeniem zadania jest dokonywanie obliczeń na danych. Komponent tego typu nie powinien zajmować się zarządzaniem subskrypcjami.

Każde zadanie posiada własny identyfikator. Może być on przekazany jako parametr konstruktora. Jeżeli nie jest, wówczas identyfikatorem zadania staje się identyfikator modyfikowanej danej.

Zadanie posiada ściśle zdefiniowany interfejs, który został przedstawiony na listingu 5.7.

Listing 5.7: Interfejs klasy Task

```
1  class Task : public QObject
2  {
3      Q_OBJECT
4  public:
5      explicit Task(QString target, std::map<QString, QString> sources);
6      explicit Task(QString id, QString target, std::map<QString,
           ↳ QString> sources);
7
8      virtual ~Task();
9      virtual bool start() final;
10     virtual void setSubscription(QString id,
           ↳ std::shared_ptr<Subscription> sub) final;
11
12     std::vector<Dependency>& getDependencies();
13     QString getId();
14
15 signals:
16     void finished(QString id, bool success);
17
18 protected:
19     virtual bool run() = 0;
20     virtual std::shared_ptr<Subscription> sub(QString id) final;
21     virtual bool subExists(QString id) final;
22     virtual bool isCancelled();
23
24 private:
25     QString id;
26     std::vector<Dependency> dependencies;
27     std::map<QString, QString> sources;
28     std::map<QString, std::shared_ptr<Subscription>> subscriptions;
29
30 };
```

Analizując listing 5.7 można ustalić, że jest to klasa abstrakcyjna. Klasy dziedziczące muszą zdefiniować metodę `run`, aby nie być abstrakcyjne. Co więcej, wystarczające jest aby klasa zdefiniowała jedynie konstruktory oraz tą metodę, ponieważ właśnie ta ona przeznaczona jest do wykonywania obliczeń na danych. Wszelkie pozostałe metody mają charakter pomocniczy oraz są zapewnione przez klasę bazową. W ciele jej do obiektu subskrypcji można się odwołać za pomocą metody `sub`. Można również sprawdzić czy dana subskrypcja została utworzona dzięki metodzie `subExists`. Argumentem obu tych metod jest wewnętrzny identyfikator danej. Zadanie po zakończeniu metody `run` emituje sygnał `finished`. Informacja taka może być przydatna dla kreatora zadania.

5.6. Task Scheduler

Task Scheduler pełni zadanie komponentu zarządzającego zadaniami. Jak można wywnioskować z listingu 5.8 klasa ta nie jest skomplikowana. Poprzez jedyną publiczną metodę `pushTask` trafiają do niego zadania stworzone przez kreatorów. Gdy zadanie zostanie przekazane do Task Scheduler'a, tworzy on subskrypcje dla niego. Funkcjonalność tą realizuje metoda `createSubscriptions`. Zadanie, które posiada utworzone subskrypcje trafia do puli zadań (składowa `taskPool`). Po dodaniu zadania do puli, pula zostaje przeiterowana w poszukiwaniu zadań gotowych do uruchomienia. Predykatem w tej sprawie jest Subscription Manager. Scheduler przekazuje mu listę zależności zadania a z powrotem otrzymuje wartość logiczną. Jeżeli jest ona równa **`true`**, wówczas Task Scheduler uruchamia zadanie za pomocą metody `startTask`. Jeżeli otrzymana wartość wynosi **`false`**, Task Scheduler nie podejmuje żadnych akcji dla tego zadania i powraca do iteracji puli.

W metodzie `startTask` tworzony jest wątek, do którego przekazywany jest obiekt zadania. Wątek jest uruchamiany, a w nim uruchamiane zostaje zadanie. Dodatkowo ustanawiane jest połączenie, za pomocą którego po zakończeniu zadania Task Scheduler ponownie iteruje pule zadań w celu znalezienia kandydata do uruchomienia.

Listing 5.8: Deklaracja klasy `TaskScheduler`

```
1  class TaskScheduler : public QObject
2  {
3      Q_OBJECT
4  public:
5      TaskScheduler(SubscriptionManager& sm);
6      void pushTask(std::shared_ptr<Task> task);
7
8  private:
9      void checkTaskPool();
10     void startTask(std::shared_ptr<Task> task);
11     void createSubscriptions(std::shared_ptr<Task> task);
12
13     std::list<std::shared_ptr<Task>> taskPool;
14     SubscriptionManager& sm;
15 };
```