

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, 1 grudnia 2016

Spis treści

Spis listingów	5
1. Cel projektu	6
2. Obrazowanie wielospektralne	7
2.1. Format danych	7
2.1.1. Konsekwencje formatu danych	8
2.2. Dane w systemie Gerbil	8
2.2.1. Wpływ hierarchii danych na proces wykonania	9
3. Technologie wykorzystane w systemie Gerbil	10
3.1. C++	10
3.1.1. STL	10
3.2. Qt	11
3.2.1. Sygnały i sloty	11
3.2.2. Wątek GUI oraz wątki robocze	13
3.3. Boost	14
3.3.1. Boost.Any	14
3.4. Intel Threading Building Blocks	14
3.5. OpenCV	14
4. Aktualny stan projektu Gerbil	16
4.1. Wzorzec MVC	16
4.2. Architektura aplikacji Gerbil	17
4.2.1. Wady architektury	17
5. Projekt nowego systemu	19
5.1. Zarys projektu	19
5.1.1. Streszczenie przepływu danych w systemie	19
5.2. Model danych współdzielonych	20
5.2.1. Przechowywanie danych	20
5.2.2. Synchronizacja dostępu do danych	20
5.3. Model	22

5.4. Subskrypcja (Subscription) oraz Zamek (Lock)	24
5.5. Zadanie (Task)	25
5.6. Task Scheduler	28
5.7. Menedżer Subskrypcji (Subscription Manager)	28
Indeks rzeczowy	31
Literatura	31

Spis rysunków

2.1. Schemat kostki wielospektralnego	7
2.2. Graf zależności danych w systemie Gerbil	9
4.1. Podział ról we wzorcu architektonicznym MVC	16

Spis listingów

3.1. Klasa Sender	12
3.2. Klasa Receiver	12
3.3. Składnia tworzenia połączeń między obiektami	13
5.1. Aliasy używane w kodzie aplikacji	20
5.2. Składowe klasy DataEntry zapewniające bezpieczne użycie w środowisku wielowątkowym	21
5.3. Metody klasy DataEntry zapewniające bezpieczny odczyt danych współdzielonych w środowisku wielowątkowym	21
5.4. Metody klasy DataEntry zapewniające bezpieczny zapis danych współdzielonych w środowisku wielowątkowym	22
5.5. Interfejs klasy Model	23
5.6. Przykłady rejestrowania danych	23
5.7. Przykład tworzenia obiektu subskrypcji z odroczonym dostępem do danych . . .	25
5.8. Przykład tworzenia obiektu subskrypcji z bezpośrednim dostępem do danych . .	25
5.9. Interfejs klasy Task	27
5.10. Deklaracja klasy TaskScheduler	28

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.

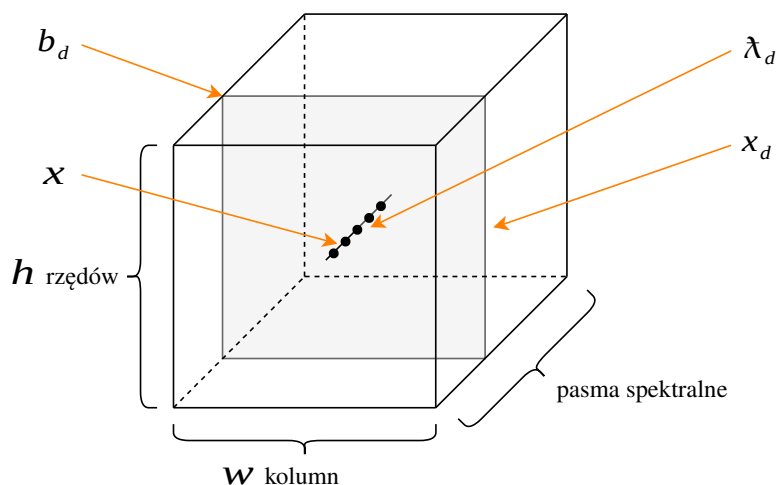
Rozdział 2

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 (h rzędów na w kolumn) 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 serie obrazów wielospektralnych mogą bezproblemowo osiągać rozmiary setek megabajtów, lub nawet gigabajtów. Zazwyczaj jednak obrazy są rejestrowane aparatem o matrycy ok. 2 Mpix. 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 kopii danych,
- dokonywać obliczeń danych wyłącznie na żądanie,
- zwalniać z pamięci dane, które nie są już wykorzystywane przez aplikację.

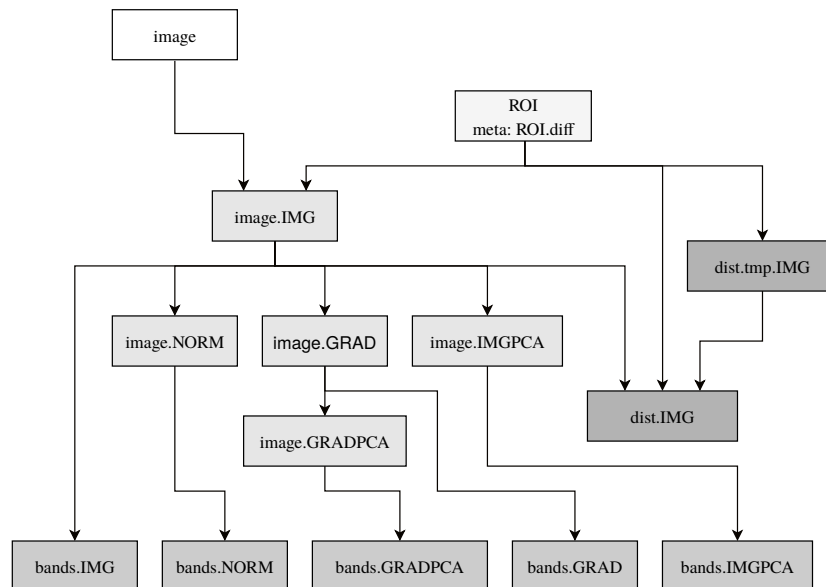
2.2. Dane w systemie Gerbil

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 (ang. Principal Component Analysis - Analiza głównych składowych)[1],
- **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 jedno 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 rozpatrzyć 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 równoległe, wobec tego można zdefiniować opcjonalne wymaganie dla systemu zarządzania zadaniami jako możliwość równoległego przetwarzania zadań.

Rozdział 3

Technologie wykorzystane w systemie Gerbil

Projekt jest rozwijany pod systemem operacyjnym Linux, dystrybucją Ubuntu 16.04.

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 cykli procesora),
- umożliwiający tworzenie lekkich i wydajnych abstrakcji [2][3].

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

W projekcie używany jest standard ISO/IEC 14882:2011 języka C++ oraz kompilator GCC w wersji 5.4.0 [4].

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 [2]. Ułatwia to tworzenie aplikacji wieloplatformowych. Dzięki gotowym rozwiązaniom zawartym w bibliotece standardowej, proces wytwarzania oprogramowania zyskuje na prostocie i efektywności.

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 [2].

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 danych 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 [2].

condition_variable

3.2. Qt

Qt jest platformą programistyczną wyposażoną w narzędzia pozwalające usprawnić proces wytwarzania oprogramowania oraz interfejsów użytkownika dla aplikacji desktopowych, wbudowanych bądź mobilnych [5].

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 rozszerzeń 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 zgodne. Mechanizm ten jest luźno powiązany (ang. loosely coupled). Oznacza to, że klasa emitująca sygnał nie musi być świadoma 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` [5].

Składnia

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

```

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 3.1: Klasa `Sender`

```

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 };

```

Listing 3.2: Klasa `Receiver`

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.

```

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);

```

Listing 3.3: Składnia tworzenia połączeń między obiektami

Mechanizm sygnałów i slotów jest powszechnie wykorzystywany w systemie Gerbil do ustanowienia 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 komponenty graficznego interfejsu użytkownika (ang. widget) 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 [5]. Gdyby te obliczenia były wykonywane w głównym wątku, aplikacja przestałaby być responsywna podczas obliczeń, co jest efektem niepożądanym.

3.3. Boost

Boost jest kolekcją bibliotek do języka C++. Biblioteki te poszerzają funkcjonalności tego języka [6]. 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 [2].

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 posiadać możliwość inicjalizacji na bazie innego obiektu tego typu – ang. copy-constructible). 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** [6].

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 [7].

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

3.5. OpenCV

OpenCV (Open Source Computer Vision Library) to biblioteka przeznaczona do rozpoznawania obrazów oraz uczenia maszynowego [8].

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

Rozdział 4

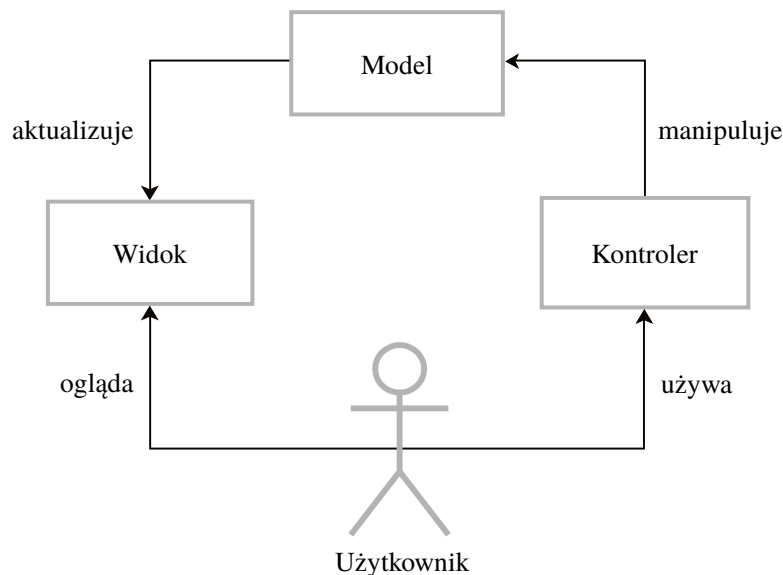
Aktualny stan projektu Gerbil

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 [5].

4.2. Architektura aplikacji Gerbil

W aplikacji Gerbil wzorec 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 (obiekt klasy 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, który pozostaje przez cały czas responsywny.

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 przypadku gdy użytkownik poprzez interakcję z systemem zleci wykonanie na raz kilku zadań dokonujących obliczeń na tych samych danych, system może zachować się w sposób nieoczekiwany. Prowadzi to do zakończenia aplikacji z powodu naruszenia ochrony 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 patologii. Przykład stanowi używanie współdzielonych 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 SubscriptionManager (SM). Jest on właścicielem wszystkich współdzielonych danych. Pozostałe komponenty uzyskują dostęp do danych przez obiekty klasy Subscription. Prawa dostępu są przyznawane oraz kontrolowane przez SM. Dane tworzą graf zależności, który zapewnia automatyczne obliczanie potrzebnych danych. Komponenty typu Model (zbliżone do modeli we wzorcu MVC) kontrolują proces powstawania danej poprzez tworzenie odpowiednio sparametryzowanych zadań. Utworzone zadania trafiają do komponentu TaskScheduler, który zarządza ich wykonaniem.

5.1.1. Streszczenie przepływu 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 widoku tworzony jest obiekt klasy Subscription, który stanowi definicję takiego żądania. Obiekty tego typu stanowią interfejs pomiędzy komponentami żądającymi dostępu do danych a obiektem klasy SubscriptionManager. Jeżeli dane są aktualne, panel otrzymuje stosowną informację. W ciele metody obsługującej tę informację 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 Model, który zadeklarował umiejętność wytworzenia tej danej. Komponent w odpowiedzi na informację tworzy obiekt klasy Task oraz przekazuje go do obiektu klasy TaskScheduler. 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). Obiekt klasy TaskScheduler tworzy odpowiednie obiekty subskrypcji dla danego zadania, na podstawie jego sygnatury zależności. Akcja ta może prowadzić do powstania kolejnych obiektów zadań (jeżeli zadanie żąda dostępu

do danej, która również nie jest aktualna). Obiekt klasy TaskScheduler uruchamia powstałe zadania w osobnych wątkach. W momencie gdy dana zażądana 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.

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

Listing 5.1: Aliasy używane w kodzie aplikacji

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 wykorzystywana 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.

```

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

```

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

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

```

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 }

```

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

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 uśpiony do momentu powiadomienia przez zmienną warunkową. Aby uniknąć fałszywych wybudzeń przekazywana jest dodatkowo wyrażenie lambda, która służy za predykat. Jeśli wartość zwrócona przez lambda 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.

```
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 }
```

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

Analizując implementację metod pozwalających na bezpieczny zapis danych przedstawionych na listingu 5.4 można dostrzec dużą analogię 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. Model

Model jest zmodyfikowaną wersją Modelu z wzorca MVC. Jego zadaniem jest kontrola procesu wytworzenia danej. Interfejs zdefiniowany dla modelu został przedstawiony na listingu 5.5.

```

1  class Model : public QObject
2  {
3      Q_OBJECT
4  public:
5      explicit Model(SubscriptionManager& sm, TaskScheduler* scheduler,
6                      QObject *parent = 0);
7      virtual ~Model();
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 };

```

Listing 5.5: Interfejs klasy Model

Metoda `registerData` służy do rejestrowania danych. Za jej pomocą model zgłasza współdzielone dane, za które bierze odpowiedzialność. Każda dana współdzielona, aby istnieć w systemie, musi zostać zarejestrowana przez któryś z modeli. 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 linii 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.

```

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

```

Listing 5.6: Przykłady rejestrowania danych

Z analizy listingu 5.5 wynika, że klasa implementująca interfejs `Model` musi zdefiniować metodę `delegateTask` aby nie być klasą abstrakcyjną. Metoda ta jest kluczowa dla tego interfejsu, oraz bardzo ważna dla całego systemu. 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 modelu jest utworzenie odpowiedniego obiektu klasy `Task` i przekazanie go do obiektu klasy `TaskScheduler`. Teoretycznie możliwe jest, aby model sam dokonał obliczenia danej, zamiast tworzyć obiekt klasy `Task`. 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 modelu.

5.4. Subskrypcja (Subscription) oraz Zamek (Lock)

Komponent `Subscription` pełni rolę pośrednika między jednostką zarządzającą danymi a jednostką, która o dostęp do danych prosi. Poprzez utworzenie subskrypcji komponent deklaruje chęć uzyskania dostępu do danych. Do stworzenia takiego obiektu potrzebne jest ustalenie jego przeznaczenia (odczyt bądź zapis) oraz nazwa pożądanego danej. Aby dokonać faktycznego dostępu do danej tworzy się obiekt klasy `Lock` (zamek). Podczas inicjalizacji przekazuje mu się obiekt subskrypcji - można stwierdzić, że jest ona wówczas zajęta. `Lock` jest szablonem klasy. Podczas tworzenia tego obiektu należy skonkretyzować go dwoma typami: pierwszy jest typem danej, natomiast drugi jest typem metadanej. Jeżeli jednak nie ma się zamiaru korzystać z metadanych, nie ma potrzeby podawania jej typu. Za pomocą obiektu typu `Lock` można otrzymać bezpośredni uchwyt do danej oraz do metadanej. Podczas pozyskiwania uchwytów wewnętrzna implementacja klasy `Lock` dokonuje rzutowania z typu `boost::any` na podany przy tworzeniu obiektu zamku typ, stąd potrzeba ich specyfikowania. Dostęp do danych może być blokujący - wykonanie w wątku dokonującym dostępu do danych może zostać wstrzymane. Wynika to z mechanizmów odczytu/zapisu modelu danych współdzielonych oraz logiki jednostki zarządzającej danymi.

Klasa subskrypcji zapewnia jeszcze jedną bardzo ważną funkcjonalność. Jest nią sygnalizowanie aktualizacji danych. Jest to bardzo przydatne dla komponentów interfejsu graficznego, których zadaniem jest prezentowanie żądanych danych oraz dbanie o aktualność tych danych. Aby ułatwić to zadanie, komponent GUI może podczas tworzenia obiektu subskrypcji przekazać metodę, której realizowana będzie obsługa sygnału aktualizacji danej. Dzięki temu komponenty takie jak widok nie muszą posiadać logiki sprawdzania czy dane zostały zaktualizowane. Taki sposób dostępu do danych został nazwany odroczonym (ang. *deferred*). Konstrukcja obiektu subskrypcji w ten sposób została przedstawiona na listingu 5.7. W opozycji do dostępu odroczonego stoi dostęp bezpośredni (ang. *direct*), który jest bardziej uniwersalny. Jest to nic innego jak utworzenie obiektu subskrypcji oraz obiektu zamka. Deklarując dostęp odroczony przy

tworzeniu obiektu subskrypcji można używać również dostępu bezpośredniego (co zazwyczaj ma miejsce w ciele metody, która jest obsługą sygnału aktualizacji danej). Nie można jednak używać dostępu odroczonego jeśli subskrypcja została utworzona z zadeklarowanym dostępem bezpośrednim. Konstrukcja dostępu bezpośredniego została przedstawiona na listingu 5.8.

```
1 Subscription* sub = SubscriptionFactory::create(Dependency("image.IMG",
2     SubscriptionType::READ), AccessType::DEFERRED, this,
3     std::bind(&ImgWindow::displayImg, this));
```

Listing 5.7: Przykład tworzenia obiektu subskrypcji z odroczonym dostępem do danych

```
1 Subscription* sub = SubscriptionFactory::create(Dependency("dist.tmp.IMG",
2     SubscriptionType::READ), AccessType::DIRECT);
```

Listing 5.8: Przykład tworzenia obiektu subskrypcji z bezpośrednim dostępem do danych

Aby umożliwić dostęp do danych, subskrypcja potrzebuje uchwytu do jednostki zarządzającej danymi (Subscription Manager'a). Ważne jest aby dostęp do danych współdzielonych był prosty dla każdego komponentu w systemie, również te, które nie mają dostępu do jednostki zarządzania danymi. Zatem aby tworzenie obiektów subskrypcji było możliwe z dowolnego miejsca w kodzie realizowane jest ono poprzez fabrykę SubscriptionFactory. Fabryka posiada uchwyt do Subscription Manager'a, który przekazuje obiektowi subskrypcji.

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).

Identyfikator (ID) modyfikowanej danej musi zostać przekazany zadaniu jako parametr jego konstruktora. Identyfikatory źródeł również są przekazywane w konstruktorze, 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 nazewnictwa wewnątrz zadań. Zazwyczaj zadania posiadają jedynie jedno źródło, do którego odnoszą się za pomocą identyfikatora source. ID danej modyfikowanej jest automatycznie mapowany 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 5.9.

```

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 };

```

Listing 5.9: Interfejs klasy Task

Analizując listing 5.9 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 tę 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 modelu odpowiedzialnego za zadanie.

5.6. Task Scheduler

Task Scheduler pełni zadanie komponentu zarządzającego zadaniami. Jak można wywnioskować z listingu 5.10 klasa ta nie jest skomplikowana. Poprzez jedyną publiczną metodę `pushTask` trafiają do niego zadania stworzone przez modele. 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 kwestii, czy zadanie może zostać uruchomione, czy też nie jest metoda `processDependencies` klasy `SubscriptionManager`. 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 pulę zadań w celu znalezienia kandydata do uruchomienia.

```

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 };

```

Listing 5.10: Deklaracja klasy TaskScheduler

5.7. Menedżer Subskrypcji (Subscription Manager)

Subscription Manager pełni rolę jednostki głównej w systemie. Komponent ten agreguje wszystkie takie dane oraz nimi zarządza poprzez subskrypcje. 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.

Funkcjonalności oferowane przez omówione wcześniej komponenty – Subskrypcje oraz Zamki nie są przez nie realizowane. Subskrypcja czy Zamek to jedynie interfejs udostępniający funkcjonalności Menedżera Subskrypcji.

Omówione wcześniej Subskrypcje oraz Zamki pełnią jedynie rolę interfejsów udostępniających funkcjonalności zapewniane przez Subscription Manager'a. Dla każdej danej kontrolowanej przez ten komponent przechowywane są następujące informacje:

- liczba subskrypcji utworzonych w celu odczytu danej,
- flaga wskazująca na istnienie subskrypcji utworzonej w celu zapisu danej,
- liczba aktywnych dostępów do danych w celu odczytu - pierwsza próba dostępu do danej bądź metadanej przy użyciu obiektu zamka jest rejestrowana przez SM jako aktywny dostęp do danych. Dostęp jest uznawany za zakończony gdy obiekt zamka wyjdzie poza zasięg, bądź przez jawne zakończenie dostępu poprzez metodę `release` klasy `Lock`,
- flaga wskazująca na aktywny dostęp do danych w celu zapisu - analogicznie jak w przypadku powyższym,
- flaga determinująca czy dana została zainicjalizowana,
- flaga determinująca czy dana jest aktualna,
- uchwyt do modelu, który zarejestrował daną,
- kolekcję identyfikatorów danych zależnych,
- kolekcję obiektów subskrypcji tej danej.

Gdy jakiś komponent utworzy obiekt subskrypcji do danej, obiekt ten staje się pośrednikiem pomiędzy komponentem a Menedżerem Subskrypcji.

Jeśli subskrypcja została stworzona w celu odczytu danej, SM inkrementuje liczbę takich subskrypcji. Jeżeli ze stanu danej wynika, że nie jest ona aktualna bądź zainicjalizowana SM wysyła żądanie o utworzenie zadania obliczającego do modelu odpowiedzialnego za tą daną. Jeżeli dana jest aktualna oraz nie odbywa się aktualnie zapis tej danej a subskrypcja została utworzona ze zdefiniowanym odroczonym sposobem dostępu do danej, SM wysyła sygnał do komponentu (poprzez obiekt subskrypcji), mówiący że dana jest aktualna i można przeprowadzić do niej dostęp.

Jeśli subskrypcja została stworzona w celu zapisu (modyfikacji) danej, SM przypisuje wartość logiczną prawdy do flagi wskazującej na obecność takiego typu subskrypcji oraz wartość logiczną fałszu do flagi wskazującej na aktualność danej. Następnie sygnał o modyfikacji danej jest propagowany w dół grafu zależności danych. Jeżeli któraś z danych zależnych od danej, która będzie zmodyfikowana, posiada jakieś (bezpośrednie bądź pośrednie) subskrypcje, wówczas do modelu odpowiedzialnego za nią wysyłane jest żądanie o utworzenie zadania obliczającego. Zazwyczaj model reaguje pozytywnie - tworzy zadanie, przekazuje je do Task Scheduler'a, który tworzy subskrypcje dla zadania (w tym zawsze jedną w celu zapisu), przez co wykonanie znowu

znajduje się w punkcie rejestrowania subskrypcji przez *Subscription Manager*'a, tym razem jednak dla danej pochodnej. Proces ten w sposób rekurencyjny powtarza się dla wszystkich danych zależnych od wyjściowej danej. W następnym kroku dla każdej danej zależnej od danej wyjściowej, fladze wskazującej na aktualność danej zostaje przypisana wartość logiczna fałszu.

Subscription Manager musi podjąć również jakąś akcję w przypadku, gdy obiekt subskrypcji zostanie usunięty. Kolekcja obiektów subskrypcji zostaje zaktualizowana. Jeżeli kolekcja ta jest pusta, dana oraz metadana zostają zwolnione, a flagom inicjalizacji oraz aktualności zostaje przypisany fałsz. Jeżeli kolekcja nie jest pusta, wówczas dla subskrypcji w celu odczytu dodatkowo zostaje dekrementowany licznik subskrypcji, zaś w przypadku subskrypcji w celu zapisu/modyfikacji, fladze wskazującej na istnienie takiej subskrypcji zostaje przypisany fałsz, a do wszystkich subskrybentów danej zostaje wysłany sygnał o aktualizacji danej.

Gdy ma miejsce faktyczny dostęp do danych, poprzez obiekt zamka, w celu odczytu *Subscription Manager* wywołuje metodę `read` klasy `DataEntry`, inkrementuje licznik aktywnych dostępów do danych w celu odczytu oraz zwraca dane pozyskane za pomocą metody. Podczas dostępu w celu zapisu sytuacja jest analogiczna – jedyną różnicą jest fakt przypisania wartości prawdy fladze aktywnego dostępu zamiast inkrementacji licznika.

Gdy faktyczny dostęp w celu odczytu zostaje zakończony wywołana zostaje metoda `endRead` klasy `DataEntry`, a licznik aktywnych dostępów zostaje dekrementowany. W sytuacji zakończenia zapisu wywoływana zostaje metoda `endWrite`, flaga wskazująca na aktywny zapis zostaje ustawiona na fałsz, a flagom wskazującym zainicjalizowanie oraz aktualność danej przypisana jest prawda.

Literatura

- [1] *Analiza głównych składowych* https://pl.wikipedia.org/wiki/Analiza_g%C5%82%C3%B3wnych_sk%C5%82adowych (dostęp 23.11.2016).
- [2] Bjarne Stroustrup, *Język C++. Kompendium wiedzy*. Wydawnictwo Helion, Gliwice, Wydanie IV, 2014.
- [3] *Krótki opis języka C++* <http://www.cplusplus.com/info/description> (dostęp 31.10.2016).
- [4] *ISO/IEC 14882:2011* http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372 (dostęp 23.11.2016).
- [5] *Dokumentacja Qt 5.7* <http://doc.qt.io/qt-5/index.html> (dostęp 30.10.2016).
- [6] *Oficjalna strona Boost* <http://www.boost.org/> (dostęp 30.10.2016).
- [7] *Oficjalna strona Threading Building Blocks* <https://www.threadingbuildingblocks.org/> (dostęp 31.10.2016).
- [8] *Oficjalna strona OpenCV* <http://opencv.org/about.html> (dostęp 31.10.2016).