**Nowy model kanałów danych - DataChannel**

Ponieważ w aplikacji występuje mnóstwo różnych danych o charakterze czasowym ( w formie argument – wartość, gdzie argument to najczęściej czas) na potrzeby ich przechowywania, zarządzania nimi oraz dostępem do tych danych przeprojektowano poprzedni model.

**Bazowy interfejs do surowych danych kanałów – dostęp indexowy do danych**

Nowy model proponuje udostępnienie interfejsów z dostępem do danych surowych – po indeksach danych – zakładając, iż kolejne próbki mają argumenty w porządku ściśle rosnącym. Interfejsy te są całkowicie niezależne od sposobu przechowywania danych – nie narzucają żadnego z kontenerów, jednak nie udostępniają akcesoriów specyficznych dla kontenerów – iteratorów.

**Kanały ogólne i z argumentami równooddalonymi**

Dodatkowo w nowym modelu zostały silnie rozdzielone dane od ich interpretacji, dostępu i opisu. Jedyny podział jaki jest wprowadzony to podział na dane ogólne (wymaganie by kolejne argumenty były większe od poprzedniego) oraz na dane o równomiernym rozkładzie argumentów – dla tych danych realizowane są dużo efektywniej dostępy do danych wg argumentu a nie indeksów. Jest to możliwe poprzez wprowadzenie pomocniczych metod zwracających indeksy najbliższych próbek odpowiadających zadanej wartości argumentu.

**Interfejsy**:

* [IRawGeneralDataChannelReader](https://www.assembla.com/wiki/show/edytor/IRawGeneralDataChannelReader)
* [IRawUniformDataChannelReader](https://www.assembla.com/wiki/show/edytor/IRawUniformDataChannelReader)

**Akcesory do danych**

Wszystkie dane przechowywane są w sposób dyskretny. Dzięki interfejsom uniezależniamy się od faktycznego sposobu przechowywania danych (tamblic, wektorów). W dostarczonych implementacjach dane są przechowywane w kontenerach typu wektor. Dane są przechowywane zawsze parami: argument - wartość. Do każdej ze składowych takiej pary mamy dostęp po indeksie próbek. Aby nadać danym charakter ciągły można posłużyć się akcesorami – klasami zapewniającymi dostęp do danych wg wartości argumentów a nie indeksów. Akcesor może wewnętrznie realizować różne metody interpolacji zwracając wartości próbek dla argumentów, które nie występują w kanale jawnie, jednak które teoretycznie mieszczą się w kanale. Akcesor może realizować też inne funkcje podczas dostępu do danych – może wprowadzać offset i skalę przy operacjach na argumentach. Akcesory pracują z najogólniejszym interfejsem danych opartym na dostępie indeksowym do danych.

**Interfejsy**:

* [IDataChannelTimeAccessor](https://www.assembla.com/wiki/show/edytor/IDataChannelTimeAccessor)

**Implementacje**:

* [SimpleDataChannelTimeAccessor](https://www.assembla.com/wiki/show/edytor/SimpleDataChannelTimeAccessor)
* [GeneralDataChannelTimeAccessor](https://www.assembla.com/wiki/show/edytor/GeneralDataChannelTimeAccessor)

**Kanały obserwujące, automodyfikujące**

Kolejną nowością są kanały obserwujące inne kanały i w razie ich modyfikacji modyfikujące same siebie na bazie obserwowanych kanałów. To pozwala wprowadzać np. różnego rodzaju metody wygładzające dane w kanałach, filtrujące dane w kanałach. Do swojego działania takie kanały potrzebują jedynie prostych obiektów funkcyjnych realizujących modyfikację, którą chcemy realizować na obserwowanym kanale. Można też tworzyć obiekty modyfikujące obiektów modyfikujących tworząc w ten sposób ciąg modyfikacji danego kanału, mając kopię każdej modyfikacji z osobna.

**Interfejsy**:

* [IChannelAutoModifier](https://www.assembla.com/wiki/show/edytor/IChannelAutoModifier)

**Implementacje**:

* [ChannelAutoModifier](https://www.assembla.com/wiki/show/edytor/ChannelAutoModifier)

**Timery**

Specjalnie dla danych czasowych wprowadzono też pojęcie Timerów – prostych klas obsługujących odczyt i zapis czasu. Można tu wyszczególnić interfejsy dla odczytu czasu, dla odczytu i zapisu czasu. Dodatkowo dostępna jest  implementacja najprostszego Timera. Timery pozwalają nam mając jedną kopie danych prowadzić dla nich niezależne zegary. Jest/może to być wykorzystywane w wizualizatorach.

**Interfejsy**:

* ITimer

**Implementacje**:

* Timer

**Klasy narzędziowe**

Poza wymienionymi elementami wprowadzono również kilka klas narzędziowych. Jedną do pobierania aktualnej wartości kanału, która przyjmuje jako argumenty akcesor oraz timer i udostępnia metodę o sygnaturze:

value getCurrentValue(argument)

na bazie czasu z timera i sposobu interpretacji danych akcesoria.

**Implementacja**:

* [CurrentValueExtractor](https://www.assembla.com/wiki/show/edytor/CurrentValueExtractor)

Drugą klasą pomocniczą jest [ChannelStats](https://www.assembla.com/wiki/show/edytor/ChannelStats), która dla zadanego kanału zwraca jego podstawowe statystyki – wartość minimalną/maksymalną, średnią, medianę, variancję. Typ danych, które są przechowywane w kanale musi mieć jednak zdefiniowane podstawowe operacje arytmetyczne pozwalające wyznaczyć wartości wymienionych statystyk.

**Implementacja**:

* [ChannelStats](https://www.assembla.com/wiki/show/edytor/ChannelStats)

**Implementacje kanałów**

Dostepne są dwie implementacje kanałów – jedna dla kanałów ogólnych i druga dla kanałów z argumentami równooddalonymi. Implementacje te różnią się od interfejsów o dostępie indeksowym tym, iż mają już charakter kontenerów, ponieważ sposób przechowywania w nich danych jest już ściśle zdefiniowany - mamy więc tutaj dostęp do iteratorów. Dodatkowo implementacje te pozwalają dodawać kolejne próbki do kanałów, czego nie ma w przypadku interfejsów bazowych – dostęp do danych jest tam tylko do odczytu.

**Implementacje**:

* [RawGeneralDataChannel](https://www.assembla.com/wiki/show/edytor/RawGeneralDataChannel)
* [RawUniformDataChannel](https://www.assembla.com/wiki/show/edytor/RawUniformDataChannel)

Dekoratory kanałów – opis argumentów i wartości

Każdy kanał można dodatkowo rozszerzyć o opis jego argumentów (czasu) i wartości. Można podawać jednostki i skalę dla argumentów i wartości.

**Implementacje**:

* [ChannelWithDescriptor](https://www.assembla.com/wiki/show/edytor/ChannelWithDescriptor)

**Deklaracje klas i interfejsów**

Poniżej znajdują się podstawowe interfejsy i typy danych z ich krótką charakterystyką i specyfikacją.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType)> struct [ConstReferenceType](https://www.assembla.com/wiki/show/edytor/ConstReferenceType);

Wzorzec ten generuje typ danych stałej referencji dla typu [PointType](https://www.assembla.com/wiki/show/edytor/PointType) – używany przy przekazywaniu parametrów w celu uniknięcia ich kopiowania.

* [IChannelDescriptorReader](https://www.assembla.com/wiki/show/edytor/IChannelDescriptorReader)

Interfejs dostarczający opisu danych w kanale:

* Jednostek argumentu (argument – zazwyczaj czas)
* Skali argumentu (względem jego jednostki podstawowej – np. dla czasu będzie to wielokrotność 1 sekundy. Wartość skali równa 60 oznacza że czas jest w minutach, 3600 w godzinach, …)
* Jednostek wartości (value – różne wielkości)
* Skali wartości – patrz skala argumentu

* [IChannelDescriptorWriter](https://www.assembla.com/wiki/show/edytor/IChannelDescriptorWriter)

Interfejs pozwalający modyfikować opis kanału dostępny przez interfejs [IChannelDescriptorReader](https://www.assembla.com/wiki/show/edytor/IChannelDescriptorReader)

* [IChannelDescriptor](https://www.assembla.com/wiki/show/edytor/IChannelDescriptor)

Interfejs pozwalający czytać i modyfikować opis kanału

* [ChannelDescriptor](https://www.assembla.com/wiki/show/edytor/ChannelDescriptor)

Implementacja interfejsu [IChannelDescriptor](https://www.assembla.com/wiki/show/edytor/IChannelDescriptor)

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [ChannelAutoModifier](https://www.assembla.com/wiki/show/edytor/ChannelAutoModifier);

Wzorzec klasy umożliwiającej śledzenie zmian obserwowanego kanału i dokonywania modyfikacji własnych danych. Można łączyć takie obiekty tworząc bardziej skomplikowane łańcuchy modyfikacji danych

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [IDataChannelTimeAccessor](https://www.assembla.com/wiki/show/edytor/IDataChannelTimeAccessor);

Wzorzec interfejsu zapewniającego dostęp do danych po argumentach (czasie)

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [IRawDataChannelBasicWriter](https://www.assembla.com/wiki/show/edytor/IRawDataChannelBasicWriter);

Wzorzec interfejsu umożliwiającego dodawanie kolejnych próbek do kanału. Metody mogą zgłaszać wyjątek jeśli nie są obsługiwane w kanale.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [IRawDataChannelExtendedWriter](https://www.assembla.com/wiki/show/edytor/IRawDataChannelExtendedWriter);

Wzorzec interfejsu pozwalającego dodawać kolejne próbki do kanału oraz modyfikować wartości kanału dla zadanych indeksów (nie pozwala modyfikować argumentów)

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [IRawGeneralDataChannelReader](https://www.assembla.com/wiki/show/edytor/IRawGeneralDataChannelReader);

Wzorzec interfejsu umożliwiającego dostęp do danych po indeksach – podstawowy interfejs wszystkich kanałów

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [IRawUniformDataChannelReader](https://www.assembla.com/wiki/show/edytor/IRawUniformDataChannelReader);

Wzorzec interfejsu dla kanałów o równomiernym rozłożeniu (stałym odstępie) argumentów

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [RawGeneralDataChannel](https://www.assembla.com/wiki/show/edytor/RawGeneralDataChannel);

Wzorzec klasy implementującej podstawowy kanał danych o charakterze kontenera – udostepniający iteratory.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [RawUniformDataChannel](https://www.assembla.com/wiki/show/edytor/RawUniformDataChannel);

Wzorzec klasy implementującej kanał danych o rozkładzie równomiernym argumentów o charakterze kontenera – udostepniający iteratory.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [SimpleDataChannelTimeAccessor](https://www.assembla.com/wiki/show/edytor/SimpleDataChannelTimeAccessor);

Wzorzec klasy umożliwiającej dostęp do danych po argumencie stwarzającej wrażenie ciągłości danych. Zapytanie o wartość z argumentem mniejszą od anjkmnijszego argumentu w kanale lub wiuększą od największego (czyli spoza zakresu kanału argumentów kanału) skutkuje zwróceniem najbliższej mu próbki – pierwszej lub ostatniej.

* template <class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType), class [PointRefType](https://www.assembla.com/wiki/show/edytor/PointRefType) = typename boost::add\_reference< typename [PointType](https://www.assembla.com/wiki/show/edytor/PointType)>::type > class [ExceptionTimeOverflowResolver](https://www.assembla.com/wiki/show/edytor/ExceptionTimeOverflowResolver);

Wzorzec klasy obsługującej zapytania o wartości dla argumentów spoza zakresu argumentów kanału. Rzuca wyjątkiem w każdym wypadku.

* template <class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> struct [DataOperatorsManipulator](https://www.assembla.com/wiki/show/edytor/DataOperatorsManipulator);

Wzorzec klasy umożliwiającej uzyskanie wartości dla argumentów znajdujących się w zakresie argumentów kanału, ale nie zapisanych bezpośrednio w kanale – podstawa interpolatorów i ekstrapolatorów.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType), class [TimeOverflowResolver](https://www.assembla.com/wiki/show/edytor/TimeOverflowResolver) = [ExceptionTimeOverflowResolver](https://www.assembla.com/wiki/show/edytor/ExceptionTimeOverflowResolver)<[PointType](https://www.assembla.com/wiki/show/edytor/PointType" \o "PointType), [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)>, class Manipulator = [DataOperatorsManipulator](https://www.assembla.com/wiki/show/edytor/DataOperatorsManipulator)<[PointType](https://www.assembla.com/wiki/show/edytor/PointType" \o "PointType), [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)>> class [GeneralDataChannelTimeAccessor](https://www.assembla.com/wiki/show/edytor/GeneralDataChannelTimeAccessor);

Wzorzec klasy umożliwiającej dostęp do danych po argumencie, stwarzającej wrażenie ciągłości danych. Zapytania o argument spoza zakresu argumentów kanału skutkuje użyciem metod klasy do tego przewidzianej – [TimeOverflowResolver](https://www.assembla.com/wiki/show/edytor/TimeOverflowResolver) – w celu wygenerowania wartości. W przypadku argumentów wewnątrz kanału, które jednak nie są w nim zapisane wykorzystana jest klasa Manipulator, będąca najczęściej interpolatorem.

* template<class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [ITimerReader](https://www.assembla.com/wiki/show/edytor/ITimerReader);

Wzorzec interfejsu pozwalającego odpytać się o czas.

* template<class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class ITimer;

Wzorzec interfejsu pozwalającego odpytać się i ustawiać czas

* template<class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class Timer;

Wzorzec klasy implementującej interfejs ITimer – faktycznie najprostszy timer.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [CurrentValueExtractor](https://www.assembla.com/wiki/show/edytor/CurrentValueExtractor);

Wzorzec klasy pomocniczej umożliwiającej dostęp do  aktualnej wartości kanału za pomoca dostarczonego akcesora na bazie czasu dostarczonego timera.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [ChannelStats](https://www.assembla.com/wiki/show/edytor/ChannelStats);

Wzorzec klasy użytkowej oferujący proste statystyki kanałów, pod warunkiem że typ danych dla wartości implementuje podstawowe operacje arytmetyczne.

* template<class [PointType](https://www.assembla.com/wiki/show/edytor/PointType), class [TimeType](https://www.assembla.com/wiki/show/edytor/TimeType)> class [IChannelAutoModifier](https://www.assembla.com/wiki/show/edytor/IChannelAutoModifier);

Wzorzec interfejsu dla kanałów śledzących inne kanały I modyfikujących swoje wartości.

* template<class Channel> class [ChannelWithDescriptor](https://www.assembla.com/wiki/show/edytor/ChannelWithDescriptor);

Wzorzec klasy pomocniczej ubierający kanały w dekoracje w formie opisu przechowywanych wartości i argumentów.

**Zgodność bibliotek**

Projekt EDR bazuje na wielu zewnętrznych bibliotekach udostępnianych częściowo klientom poprzez interfejs pluginów:

* boost,
* Open Scene Graph,
* Qt,
* stl.

Aby plugin został poprawnie załadowany i uruchomiony w aplikacji musi być zbudowany z tymi samymi wersjami tych bibliotek jak cała aplikacja EDR. Każda niezgodność skutkuje komunikatem o błędzie i niezaładowaniu pluginu do aplikacji.

Co więcej, w aplikacji nie jesteśmy w stanie kontrolować wersji bibliotek zewnętrznych na bazie których tworzone są obiekty domenowe w pluginach. Może zdażyć się tak, że dwa niezależne pluginy tworzą całkowicie odrębne typy domenowe o tej samej bazie pochodzącej z różnych wersji tej samej biblioteki. W takiej sytuacji problem objawia się tym, że obiekty bazowe teoretycznie rozpoznawane są jako ten sam typ, jednak ich wymiana i obsługa w teoretycznie kompatybilnych pluginach może skutkować katastrofalnymi błędami wynikającymi z niegodności interfejsów czy z niezgodności binarnej.

Aby temu częsciowo zapobiec, rejestrując typ domenowy pochodzący z zewnętrznej biblioteki należy podać numer wersji tej biblioteki. Zasada ta jest niezwykle istotna i powinna być skrupulatnie przestrzegana przez dostawców pluginów. Jeśli się zaniecha tego mochanizmu znalezienie przedziwnych błedów w aplikacji związanych z bezpośrednim bądź pośrednim operaowaniem na danym pluginie (wprowadzonym typie) może być niezwykle trudne i czasochłonne!!

**Należy pamiętać by budując pluginy aktualizować wersję bibliotek zewnętrznych z których się korzysta w przypadku wprowadzania nowych obiektów domenowych!!**

Wymienione wersje podaje się w definicji pluginu, w momencie wprowadzania/rejestracji obiektów domenowych w postaci Fabryk obiektów, gdzie poza typem obiektu domenowego powinno się podać wersję biblioteki na bazie której ten typ został stworzony, Więcej o rejestracji typów domenowych znajdziemy czytając dokumentacje klas [ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper) i [ObjectWrapperT](https://www.assembla.com/wiki/show/edytor/ObjectWrapperT) oraz fabryk tych obiektów.

## Konwencje kodowania

# Informacje ogólne

Konwencja stosowana w ramach projektu jest portem konwencji Javy dostosowanym do wymagań C++. W związku z tym przy ustalaniu zasad dotyczących sposobu kodowania konkretnej kwestii należy posługiwać się algorytmem.

1. Czy w tym artykule znalazłeś odpowiedź? Jeżeli tak, przejdź do 4.
2. Czy w <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf> znalazłeś odpowiedź? Jeżeli tak, przejdź do 4.
3. Dodaj informację o niejasności do sekcji Niejasności.
4. Zastosuj się do instrukcji.

# Pliki

* nagłówki: rozszerzenie .h
* źródła: rozszerzenie .cpp
* preferowany to model jeden typ (plus jego typy zagnieżdżone) na jeden nagłówek i jeden plik źródłowy (jeżeli potrzebny); odstępstwem od tej reguły może być typowo biblioteczny kod, dostarczający funkcji, makr bądź dużej liczby prostych typów

# Kwestia #include

* publiczne nagłówki biblioteczne powinny załączać dyrektywą #include minimalny zestaw nagłówków, tak aby w kodzie użytkowym nie zachodziła potrzeba załączanie ściślej niesprecyzowanych dodatkowych nagłówków
* publiczna nagłówki biblioteczne powinny używać dyrektywy #incude wyłącznie w wariancie <>
* dyrektywy include powinny być w następującej kolejności:
  1. Nagłówki standardowe C++
  2. Nagłówki bibliotek zależnych
  3. Nagłówki naszych bibliotek
  4. Nagłówki publiczne projektu
  5. Nagłówki prywatne projektu

# Przestrzenie nazw

* kod biblioteczny powinien być umieszczony w specyficznej mu przestrzeni nazw
* przestrzeń nazw powinna być odwzorowaniem struktury katalogów, np ::vm::core odnosiłoby się do plików z katalogu /vm/core

# Nazewnictwo typów

* nie stosuje się przedrostków dla klas, struktur, unii;
* wyjątkiem są czysto wirtualne interfesjy (przedrostek I)

# Klamry

* w tej samej linii po instrukcjach warunkowych, pętlach, try, catch
* w następnej linii przy definicji typu lub funkcji/metody

# Dokumentacja

* dokumentacja powinna być zgodna z którymś z formatów obsługiwanych przez doxygen
* wszystkie metody powinny być, choćby w minimalnym stopniu, udokumentowane (np. opisany tag return lub krótki opis tekstowy)

# Header guardy

* nie stosujemy #pragma once
* format header guarda: HEADER\_GUARD\_<MODUŁ>\_\_<NAZWA\_NAGŁÓWKA\_BEZ\_H>\_H\_\_

# Konstruktory

* Listy inicjalizacyjne powinny rozpoczynać się w nowej lini; znak rozpoczęcia listy ":" powinien być ostatnim znakiem w linii z sygnaturą konstruktora

Foo::Foo(Bar\* bar) :   
FooBase(bar), zmienna1(0), zmienna2(0)  
{  
 ...  
}

# Wskaźniki

* Wskaźniki nie powinny być porówywane do zera ani być inicjowane zerem. Do inicjacji powinno się używać słowa kluczowego nullptr. W prorównaniach powinno się używać albo nullptr, albo używać domyślnej konwersji do boola.

int \* ptr;  
ptr = 0; // źle  
ptr = NULL; // źle  
ptr = nullptr; // dobrze

if ( ptr ) {} // dobrze  
if ( ptr != nullptr ) {} // dobrze  
if ( ptr != 0 ) {} // źle  
if ( ptr != NULL ) {} // źle

void foo(int\* bar) {}  
...  
foo(NULL); // źle  
foo(0); // źle  
foo(nullptr); // dobrze

## FAQ Kodowanie

### Q: Jaka jest konwencja kodowania?

A: Konwencja została przedstawiona w artykule [Konwencje kodowania](https://www.assembla.com/wiki/show/edytor/Konwencje_kodowania). Zaleca się stosowanie do sformułowanych tam reguł. Jeżeli masz wątpliwości lub pomysł na ulepszenie konwencji, to zamiast stosować zmiany na własną rękę rozpocznij dyskusję punkcie niejasności lub sugestie.

### Q: Jaka znajomość C++ jest wymagana od programistów?

**A**: Bardzo dobra. I to nie znajomość typu "programowałem wcześniej w C#/Java, C++ pamiętam". Podobnie jeżeli uważasz, że "C++ to C z klasami", to czeka Cię dużo nauki. Projekt tworzymy w idiomatycznym C++ (szablony, cechy, polimorfizm), z silnym naciskiem na bibliotekę STL (kolekcje, iteratory, algorytmy) i Boost (wyrażenia regularne, inteligentne wskaźniki, polimorficzne obiekty funkcyjne, kod użytkowy dla praktycznie każdej dziedziny), wykorzystując przy tym możliwości nowego standardu C++0X (auto, decltype, lambda). Dobrym miejscem nauki o C++ jest C++ FAQ (http://www.parashift.com/c++-faq-lite/index.html), jednak zawarte są tam informacje raczej o samym języku, nie o STLu i Boost'cie.

### Q: Czy używamy inteligentnych wskaźników?

**A:** Tak. Używamy ich wszędzie tam, gdzie od czystej wydajności ważniejsza jest przejrzystość i niezawodność kodu. W zależności od kontekstu można używać któregoś z typów:

* boost::shared\_ptr (zamapowany na core::shared\_ptr)
* boost::weak\_ptr (zamapowany na core::weak\_ptr)
* boost::scoped\_ptr (zapapowany na core::scoped\_ptr)
* osg::ref\_ptr
* osg::observer\_ptr

W dalszych pytaniach omówione będą ww. typy wsaźników. Najpierw jednak porada: ponieważ wprowadzanie inteligentych wskaźników wiąże się z zastępowaniem miłych dla oka linii typu

core::Data\* data = new core::Data();

na

core::shared\_ptr<core::Data> data(new Data());

bardzo silny nacisk kładziony jest na **tworzenie mnemonik typów wskaźników na typy zaraz po ich definicji.** Przykład:

namespace core {  
 class Data { ... };  
 typedef shared\_ptr<Data> DataPtr;  
}

Dzięki temu poprzednia linia zmienia się w:

core::DataPtr data(new Data());

Można też użyć funkcji make\_shared:

core::DataPtr data = core::make\_shared(new DataPtr());

### Q: Kiedy powinienem używać shared\_ptr i weak\_ptr?

**A:** shared\_ptr w większości przypadków. shared\_ptr należy traktować jako obiekt, który współdzieli z innym instancjami licznik referencji dla konkretnej instancji obiektu. Szczegóły użycia tych wskaźników znajdują się w dokumentacji Boosta. Uwaga: Trzeba pamiętać o tym, że obiekt opakowywany tym wskaźnikiem **nie może być wyłuskany i przypisany do innej instancji wskaźnika**. Wiąże to się z tym, że licznik referencji trzymany jest przez same shared\_ptr, nie przez opakowywany obiekt. Gdy wyłuska się "surowy" wskaźnik i przypisze wartość do innej instancji shared\_ptr, będą występować dwa niezsynchronizowane liczniki referencji tego samego obiektu, co musi prowadzić do katastrofy.

core::DataPtr data(new Data());  
core::Data\* raw = data.get(); // można też: core::Data\* raw = &\*data;  
core::DataPtr data2(raw); // będzie katastrofa!

Uzupełnieniem shared\_ptr jest weak\_ptr. Stosując shared\_ptr należy wystrzegać się **cyklicznych zależności**. Gdy obiekt A odwołuje się do obiektu B, a obiekt B do obiektu A, to oba **nigdy** nie zostaną usunięte, ponieważ licznik obu shared\_ptr zawsze będzie większy od zera. Gdy zachodzi konieczność cykliczności wskaźników, jednym z nich **musi** być weak\_ptr, który nie zwiększa licznika referencji.

### Q: Kiedy powinienem używać scoped\_ptr?

**A:** Do nieudostępnianych zmiennych prywatnych oraz zmiennych lokalnych. Syntaktycznie używa się go jak shared\_ptr, ale semantycznie znacząco się różni - nie można go kopiować. Dzięki temu jest o wiele szybszy.

### Q: Kiedy powinienem używać osg::ref\_ptr i osg::observer\_ptr?

**A:** Są to w pewnym sensie odpowiedniki shared\_ptr i weak\_ptr dla obiektów z biblioteki OSG (konkretnie: rozszerzających osg::Referenced). Różnica polega na tym, że licznik referencji przechowywany jest w samych obiektach, a nie w wskaźnikach. Dzięki temu **możliwe jest wyłuskiwanie surowych wskaźników i przypisywanie ich** do nowych instancji ref\_ptr (właściwie to ref\_ptr posiada operator niejawnej konwersji do i z surowych wskaźników). Wiele typów samego OSG tak robi, trzymając prywatne zmienne jako ref\_ptr, a w akcesorach zwracając wyłuskane surowe wskaźniki. To samo postępowanie w przypadku shared\_ptr jest wyjątkowo naganne i niewłaściwe.

osg::ref\_ptr<osg::Geometry> geom = new osg::Geometry(); // niejawna konwersja z surowego wskaźnika  
osg::Geometry\* geomRaw = geom; // niejawna konwersja do surowego wskaźnika  
osg::ref\_ptr<osg::Geometry> geom2 = geomRaw; // wszystko ok

Pozostałe uwagi, w tym ta dotycząca cykliczności, są aktualne. Dla znających Boosta: osg::ref\_ptr przypomina boost::intrusive\_ptr.

### Q: Jak rzutować inteligentne wskaźniki?

**A: Absolutnie** nie należy wyłuskiwać surowego wskaźnika, rzutować go static/dynamic\_castem i przypisywać do innej instancji (w przypadku osg możliwe, ale nieidiomatyczne). Zamiast tego należy używać funkcji {core|osg}::{static|dynamic}\_pointer\_cast, w zależności od tego, jakie wskaźniki będziemy rzutować i w jaki sposób.

osg::ref\_ptr<osg::Drawable> drawable;  
...  
osg::ref\_ptr<osg::Geometry> geom = osg::dynamic\_pointer\_cast<osg::Geometry>(drawable);

### Q: Czy używamy wyjątków?

**A:** Tak. Trzeba jednak pamiętać, że C++ nie działa jak C#/Java i nie można rzucić wyjątku oraz liczyć na to, że runtime sam posprząta. Przykład:

void foo()  
{  
 double \* hugeLeak = new double[1000000];  
 std::string \* smallLeak = new std::string();  
 throw std::runtime\_error("bar");  
}

Zmienne hugeLeak i smallLeak nigdy nie zostanie uprzątnięta, przez co wyciek pamięci. Zamiast kombinowania z wewnętrznymi blokami try/catch oraz umieszczania w nich delete w celu usunięcia takich wycieków powinno używać się idiomu [Resource Acquisition Is Initialization](http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Resource_Acquisition_Is_Initialization):

void foo()  
{  
 std::vector<double> hugeLeak(1000000);  
 core::scoped\_ptr<std::string> smallLeak(new std::string());  
 throw std::runtime\_error("bar");  
}

Destruktory std::vector i core::scoped\_ptr zostaną wywołane przy rzuceniu wyjątku, przez co nie powstaną wycieki pamięci.

Podsumowując: jeżeli nie jesteś pewny, czy wyjątki nie spowodują wycieków pamięci, to znaczy, że albo używasz jakiejś biblioteki napisanej w C i czeka Cię trochę modyfikacji kodu, albo po prostu piszesz w niediomatycznym C++.

### Q: O co chodzi z podziałem nagłówków na publiczne i prywatne?

**A:** Architektura modułowa wymaga ustalenia pewnej konwencji udostępniania nagłówów. U nas przyjęliśmy strategię stosowaną w OSG, po części również w Qt. Zakłada ona wprowadzenie podziału plików na:

* część publiczną, u nas w katalogu {edr|edrutils}/include/{podprojekt}
* część prywatną, u nas w katalogu {edr|edrutils}/src/{podprojekt}

W części publicznej z definicji mogą znajdować się wyłącznie nagłówki. Stosujemy tutaj zasadę, że **aby użyć typu X zdefiniowanego w nagłówku X.h wystarczy includować tylko i wyłączenie ten nagłówek.** Innymi słowy, do nagłówków powinno dodawać się includy dzięki którym wszystkie typy w nim użyte będą znane. Gdy używamy wskaźników czasami można wymigać się tzw. [forward declarations](http://www-subatech.in2p3.fr/%7Ephotons/subatech/soft/carnac/CPP-INC-1.shtml). Dzięki temu nie trzeba tworzyć zbiorczych nagłówków lub piętrowych dyrektyw #include gdy chcemy użyć jakiegoś konkretnego typu z jakiegoś innego modułu. Ponieważ i tak używamy prekompilowanych nagłówków nie ma to dużego wpływu na prędkość kompilacji, a przejrzystość i uniwersalność znacznie zyskują.

Jak zostało wspomniane w konwencji kodowania, powinniśmy w części publicznej używać dyrektywy #include w postaci z nawiasami ostrymi (<>), nie cudzysłowami (""). Różnice między obiema wersjami opisane są [tu](http://msdn.microsoft.com/en-us/library/36k2cdd4%28v=VS.100%29.aspx). Przykładowo plik [VideoStream](https://www.assembla.com/wiki/show/edytor/VideoStream).h z vidlib:

#include <string> // część biblioteki standardowej  
#include <vidlib/PixelFormat.h> // plik znajduje się w <edrutils>/include/vidlib  
...  
class VideoStream   
{  
 ...  
 PixelFormat format; // dzięki nagłówkowi ten typ jest znany  
 ...  
}

W części prytatnej oczywiście znajdują się źródła oraz nagłówki prywatne, jeżeli takowe istnieją. Przy includowaniu nagłówków publicznych dobrze jest używać również wersji z nawiasami - wtedy mamy pewność, że wszystko znajduje się w dobrym miejscu. Dla prywatnych nagłówków powinno się używać jednak postaci z cudzysłowami. Przykładowo plik [VideoStream](https://www.assembla.com/wiki/show/edytor/VideoStream).cpp z vidlib:

#include <math.h> // nagłówek biblioteczny  
#include <vidlib/VideoStream.h> // nagłówek publiczny  
#include "VidLibPrivate.h" // nagłówek prywatny

## FAQ Budowa projektu

### Q: Od czego w ogóle zacząć?

**A:** Projekty (edr i edrutils) organizowane są przy użyciu CMake'a. Ponieważ staramy się używać tam jak najnowszych rozwiązań najlepiej zawsze mieć aktualną wersję. Po ściągnięciu projektu należy urchomić cmake-gui, wskazać katalog ze źródłami (czyli edr/src, edrutils/src), wskazać katalog builda (NIE należy wskazywać katalogu poddawanego kontroli wersji; w ogóle najlepiej builda trzymać w osobnej gałęzi, np. w katalogu edr/../build). Gdy ściągnęliśmy pakiet bibliotek z CI (pytanie niżej) wystarczy kliknąć Configure (dwa razy, aż do ustalenia stanu wszystkich zmiennych), potem Generate i w katalogu builda powinnna być gotowa solucja.

Tutaj uwaga: nie powinno się ściągać całego trunka, ani tym bardziej całego repo. Utrudnia to potem zarządzanie i poruszanie się w ramach projektu. Zamiast tego polecam po prostu ściągać /trunk/edr lub /trunk/edrutils.

### Q: Jak powinienem zorganizować biblioteki zależne?

**A:** W obu projektach wykorzystujemy wiele zewnętrznych bibliotek. Zwłaszcza na Windowsie zarządzanie bibliotekami jest uciążliwe, gdyż często na bibliotekę składa się plik .lib (używany do linkowania) oraz plik .dll (musi znajdować się pod %PATH% albo w katalogu wykonywalnym). Dzięki CMake'owi udało się uprościć cały proces. Oczekiwana struktura katalogu z bibliotekami wygląda następująco:

<lib>/<platforma>/{ debug | release }/<nazwa\_biblioteki>/<biblioteka\_składowa>.{ so | a | lib | dll | ...  }

<platforma> ustawiana jest automatycznie (zmienna PROJECT\_LIBRARIES\_PLATFORM), natomiast <lib> (zmienna PROJECT\_LIBRARIES\_ROOT) domyślnie ustawiany jest na <src>/../../lib. Wobec tego, gdy źródła znajdują się w katalogu <EDR>/trunk/src, to domyślnie biblioteki będą wyszukiwane w <EDR>/lib. Podobnie wygląda sprawa z plikami nagłówkowymi bibliotek:

<include>/<nazwa\_biblioteki>/<nagłówki biblioteki>

<include> (zmienna PROJECT\_LIBRARIES\_INCLUDE\_ROOT) to domyślnie <src>../../include. Czasem biblioteki mają nagłówki konfiguracyjne dostosowane do platformy. Wówczas powinny się one znajdywać w:

<include>/<platforma>/<nazwa\_biblioteki>/<nagłówki biblioteki>

Najlepiej więc **ściągnąć** z CI pakiet bibliotek, rozpakować go do katalogu **poziom wyżej** niż kopia robocza i pozwolić całości konfigurować się automatycznie.

### Q: Chciałbym użyć swoich bibliotek do testów / bo mam nowsze wersje / bo CI nie działa, jak to zrobić?

**A:** Można oczywiście wrzucić binarki/nagłówki do zaproponowanej struktury katalogów. Istnieje lepszy sposób - ręczne wskazanie katalogów wyszukiwania dla bibliotek. Zmiany można dokonać dla całego projektu, dla całej biblioteki albo dla jej części,

* lokalizacja domyślnej struktury bibliotek - PROJECT\_LIBRARIES\_ROOT.
* lokalizacja podgałęzi domyślnej struktury dla wersji debug/release - zmienne PROJECT\_LIBRARIES\_ROOT\_{DEBUG|RELEASE} (to one składają się na człon <lib>/<platforma>/{debug|release} z poprzedniego pytania).
* lokalizacja katalogu wyszukiwania dla biblioteki - <biblioteka>\_LIBRARY\_DIR\_{DEBUG|RELEASE}
* lokalizacja konkretnej składowej biblioteki - <biblioteka>\_<składowa>\_LIBRARY\_{DEBUG|RELEASE} (na Windows dla bibliotek współdzielonych dochodzą <biblioteka>\_<składowa>\_LIBRARY\_{DEBUG|RELEASE}\_DLL)
* lokalizacja nagłówków - <biblioteka>\_INCLUDE\_DIR
* lokalizacja nagłówków konfiguracyjnych (dla niektórych bibliotek) - <biblioteka>\_INCLUDE\_CONFIG\_DIR

### Q: Ratunku! Zmieniłem PROJECT\_LIBRARIES\_ROOT / PROJECT\_LIBRARIES\_ROOT\_{DEBUG|RELEASE} / <biblioteka>\_LIBRARY\_DIR\_{DEBUG|RELEASE}, a ciągle mam wskazane stare biblioteki! Co zrobić?

**A:** Gdy zmieni się zmienna, na podstawie której stworzono zmienną pochodną, ta druga się nie zaktualizuje automatycznie. Zależności między zmiennymi są następujące:

PROJECT\_LIBRARIES\_ROOT = <src>/../../lib  
PROJECT\_LIBRARIES\_INCLUDE\_ROOT = <src>/../../include  
PROJECT\_LIBRARIES\_ROOT\_{DEBUG|RELEASE} = <PROJECT\_LIBRARIES\_ROOT>/<PROJECT\_LIBRARIES\_PLATFORM>/{debug|release}  
<biblioteka>\_LIBRARY\_DIR = PROJECT\_LIBRARIES\_ROOT\_{DEBUG|RELEASE}/<biblioteka>  
<biblioteka>\_<składowa>\_LIBRARY\_{DEBUG|RELEASE}[\_DLL] = <biblioteka>\_LIBRARY\_DIR/<składowa\*>  
<biblioteka>\_INCLUDE\_DIR = PROJECT\_LIBRARIES\_INCLUDE\_ROOT/<biblioteka>  
<biblioteka>\_INCLUDE\_CONFIG\_DIR = PROJECT\_LIBRARIES\_INCLUDE\_ROOT/<PROJECT\_LIBRARIES\_PLATFORM>/<biblioteka>

Gdy zmienimy "rodzica" zmiennej, to aby zaktualizować wartość dziecka należy go usunąć i skonfigurać na nowo.  
\* - uproszczenie, określanie nazwy plików jest bardziej skomplikowane

### Q: Czemu przy uruchamianiu wyskakuje mi, że brakuje jakiś plików dll?

**A:** Bo nie zaznaczyłeś opjci PROJECT\_COPY\_MODULES. Dzięki niej przy każdej konfiguracji pliki .dll będą kopiowane do katalogów z plikiem exe. Proces ten trochę trwa i jest uciążliwy, dlatego jest domyślnie wyłączony.

### Q: Mam jakieś dziwne błędy z plikami dll, o co chodzi?

**A:** W tej sytuacji najlepiej usunąć katalog <build>/bin/{Debug|Relaese|...}, zaznaczyć opcję PROJECT\_COPY\_MODULES, skonfigurować i zbudować projekt od nowa. Jeżeli coś dalej jest nie tak - wznieć alarm na assembli/Skype.

### Q: Czemu po uruchomieniu aplikacji dostaję błędy o brakujących plikach?

**A:** Bo wyłączona jest opcja PROJECT\_ENABLE\_SVN\_RESOURCES oraz PROJET\_ENABLE\_SVN\_RESOURCES\_COPYING. Koncepcja zasobów omówina została w wątku https://www.assembla.com/spaces/edytor/messages/631875 (zmieniły się nieco nazwy zmiennych). W każdym razie zaznaczenie tych opcji spowoduje dodanie do solucji projektu resources, którego zbudowanie spowoduje skopiowanie zasobów (o ile takie są w projekcie) do katalogów z plikiem exe.

### Q: Zasoby się kopiują, a wciąż mam brakujące pliki. Co jest nie tak?

**A:** W Twoim środowisku trzeba ustawić working directory (nie da się tego zrobić z poziomu CMake). W VS2010 wystarczy w opcjach projektu "Properties -> Configuration Properties -> Debugging -> Working Directory" wpisać $([OutDir](https://www.assembla.com/wiki/show/edytor/OutDir" \o "OutDir)).

### Q: Do czego służy zmienna X?

**A:** Zmienne zazwyczaj wyposażone są w komentarze. Wystarczy najechać myszką.

### Q: Wciąż nie wiem do czego służy zmienna X.

**A:** Skontaktuj się z autorem tego artykułu.

### Q: Chciałbym wrzucić wszystkie swoje biblioteki na ftpa, ale nie chce mi się ręcznie kopiować. Ratunku?

**A:** Zaznacz opcję PROJECT\_REBUILD\_DEPENDENCIES oraz wybierz ścieżkę zmienną PROJECT\_REBUILD\_DEPENDENCIES\_DIR. W wybrane miesjce zostaną skopiowane wszystkie biblioteki/nagłówki, zachowując przy tym "naszą" strukturę. Potem usuwasz ewentualne śmieci (katalogi .svn itp), archiwizujesz katalog, wrzucasz go na ftpa i każdy może użyć Twojego pakietu zamiast tego wygenerowanego przez CI.

### Q: Czy można zmienić podział na "Header files" i "Source files" w Visual Studio?

**A:** Tak, służą do tego ustawienia z grupy SOURCEGROUP. Ja polecam ustawić na (kolejno) "Private", "Public", "Private", "UI".

### Q: Mam biblioteki i nagłówki w odnalezionych przez CMake ścieżkach, ale nadal CMake generuje błędy typu "Invalid sign in command - # not alloved ..". Co jest nie tak?

**A:** CMake ma problemy ze znakami szczególnymi w ścieżkach. Jeśli preferujesz takie znaki w ścieżkach posłóż się Linkami symbolicznymi do takich katalogów, aby w ścieżce nie wystąpiły znaki specjalne (# ,itp). Linki symboliczne pod linuxem to raczej normalna sprawa. Dla Windows polecam zajrzeć [tu](http://www.howtogeek.com/howto/windows-vista/using-symlinks-in-windows-vista/), gdyż jest to nowość począwszy od VIsty. Tutaj dla katalogów interesuje nas przełącznik /J .

### Q: Co zrobić gdy poprzednie rady nic nie dały i nadal są problemy ze znalezieniem odpowiednich bibliotek i konfuguracją projektu w CMake?

**A:** Ostatnią nadzieją jest zaznaczenie opcji PROJECT / PROJECT\_VERBOSE\_CONFIG. Opcja ta loguje wszystkie komunikaty CMake - z ich pomocą powinno dać się zlokalizować problem.

## DataManager

# DataManager

[DataManager](https://www.assembla.com/wiki/show/edytor/DataManager) ma za zadanie zarządzać zasobami prób pomiarowych i materiałów takich jak modele, shadery czy skórki aplikacji. Zarzązdanie polega na dostarczaniu zainicjalizowanych zasobów o które proszą usługi. Całość opiera się o interfejs [IDataManager](https://www.assembla.com/wiki/show/edytor/IDataManager) oraz funkcje queryParsers pozwalające na przekazywanie odpowiednich parserów na żądanie, oczywiście jeśli szukany parser jest zarejestrowany przez DM.

# Interfejs IDataManager

Najważniejsze metody interfejsu:

|  |  |
| --- | --- |
| void findResources() | Wyszukuje zasoby aplikacji. |
| void findLocalTrials() | Wyszukuje lokalne próby pomiarowe. |
| void loadResources() | Załadowanie wszystkich zasobów aplikacji. |
| void loadTrial(const [LocalTrial](https://www.assembla.com/wiki/show/edytor/LocalTrial)& trial) | Załadowanie pojedynczej lokalnej próby pomiarowej. |
| coid clear() | Czyszczenie zasobów (również prób pomiarowych). |
| const std::string& getApplicationSkinsFilePath(int i) | Pobranie i-tej ścieżki do skórki aplikacji z listy ścieżek. |
| int getApplicationSkinsFilePathCount() | Pobranie informacji o ilości skórek aplikacji na liście. |
| bool isLoadLocalTrialData() const | Czy załadowano lokalną próbę pomiarową? |
| int getLocalTrialsCount() const | Liczba prób pomiarowych w zasobach. |
| void loadLocalTrial(int i) | Ładowanie i-tej próby pomiarowej z zasobów DM. |
| void loadLocalTrial(const boost::filesystem::path& path) | Ładowanie próby pomiarowej z podanej ścieżki. |
| void registerParser(core::[IParserPtr](https://www.assembla.com/wiki/show/edytor/IParserPtr" \o "IParserPtr) parser) | Rejestruje zadany parser. |
| core::[IParserPtr](https://www.assembla.com/wiki/show/edytor/IParserPtr" \o "IParserPtr) getParser(int idx) | Zainicjalizowany parser o zadanym indeksie. |
| core::[IParserPtr](https://www.assembla.com/wiki/show/edytor/IParserPtr" \o "IParserPtr) getParser(const std::string& filename) | Parser skojarzony z zadaną nazwą pliku. |
| int getNumParsers() const | Liczba zainicjalizowanych parserów. |

# Tworzenie parsera

Parsery pod względem tworzenia i rejestracji są podobne do plugion-ów. Parser który chcemy zarejestrować powinien dziedziczyć po interfejsie IParser, następnie do rejestracji używamy makro CORE\_PLUGIN\_ADD\_PARSER.

Interfejs IParser

|  |  |
| --- | --- |
| void parseFile(const boost::filesystem::path& path) | Tutaj odbywa się czytanie pliku, path to ścieżka do pliku który ma być odczytany. |
| IParser\* create() | Nowa instancja typu obecnie zainicjalizowanego parsera. |
| std::string getSupportedExtensions() const | Bardzo ważna metoda. Zwraca rozszerzenia jakie obsługuje parser w formacie "xxx;yyy;zzz", gdzie xxx, yyy i zzz to rozszerzenia, średnik służy jako separator. |
| [ObjectWrapperPtr](https://www.assembla.com/wiki/show/edytor/ObjectWrapperPtr) getObject() | Obiekt danych parsera przekazany przy pomocy wrappera. |

Jak widać, interfejs jest prosty, obsługa parserów odbywa się wewnątrz DM. Używanie parserów polega na kojarzeniu parserów z plikami będącymi zarządzanymi przez [DataManagera](https://www.assembla.com/wiki/show/edytor/DataManagera).

# Otrzymywanie parserów

Serwisy posiadają metodę loadData([IServiceManager](https://www.assembla.com/wiki/show/edytor/IServiceManager" \o "IServiceManager)\* serviceManager, [IDataManager](https://www.assembla.com/wiki/show/edytor/IDataManager)\* dataManager). Metoda wykonuje się w momencie ładowania próby pomiarowej przez edytor. Rządany parser wyłuskujemy przy pomocy szablonowych funkcji core::queryParsers(), które jako pierwszy argument zawsze przyjmują instancję [IDataManagera](https://www.assembla.com/wiki/show/edytor/IDataManagera). Funkcje zwracają instancję parsera konkretnego typu lub core::[IParserPtr](https://www.assembla.com/wiki/show/edytor/IParserPtr" \o "IParserPtr)() w przypadku gdy nie znajdzie się szukany parser. Zwracane parsery są inicjalizowane wewnątrz [DataManagera](https://www.assembla.com/wiki/show/edytor/DataManagera), nie należy podownie wywoływać metody parseFile().

Istnieją trzy przeciążone funkcje queryParsers. Najprostsza wyszukuje pojedynczy parser konkretnego typu, kolejna wersja wyłuskuje parser dla konkretnego typu i odpowiedniego pliku, ostatnia wersja służy to wyłuskiwania list parserów konkretnego typu.

## Filesystem

# Opis

Filesystem jest pojedyńczą klasą z metodami statycznymi pozwalającymi na podstawowe operacje na katalogach i plikach. Metody zapewniają poprawne działanie na uniksach i systemach windows bez konieczności używania dyrektyw preprocesora.

# Lista metod

|  |  |
| --- | --- |
| static void createDirectory(const std::string& path) | Tworzy folder o podanej ścieżce. W przypadku, gdyby w środku ścieżki nie istniał folder też zostanie utworzony, np. podając ścieżkę first/second/third gdzie istnieje tylko first, zostanie utworzony second i third. |
| static void deleteDirectory(const std::string& path) | Usuwa folder z podanej ścieżki wraz z zawartością. |
| static void deleteFile(const std::string& path) | Usuwa pojedynczy plik. |
| static void move(const std::string& pathOld, const std::string& pathNew) | Przesuwa plik lub folder wraz z zawartością o podanej ścieżce. W przypadku istnienia w miejscu docelowym pliku o tej samej nazwie plik się nadpisze. |
| static std::vector<std::string> listFiles(const std::string& path, bool recursive, const std::string& mask) | Listuje pliki które spełniają wymagania maski (rozszerzenia pliku, np. ".avi"). Aby wylistować pliki także podfolderów ustawiamy recursive na true. |
| static std::vector<std::string> listFiles(const std::string& path, bool recursive, const std::vector<std::string>& masks) | Przeciążona metoda, pozwala podać więcej niż jedną maskę. |
| static std::vector<std::string> listSubdirectories(const std::string& path) | Listuje wszystkie podfoldery danego folderu. |

## Łączenie Qt i OSG (QWidgety OSG)

Ponieważ aplikacja oparta jest na Qt (UI wysokiego poziomu) i OSG (API graficzne) konieczne jest umożliwienie łączenia obu środowisk. Można to zrobić na kilka sposobów, należy jednak zawsze używać typu core::[QOsgDefaultWidget](https://www.assembla.com/wiki/show/edytor/QOsgDefaultWidget" \o "QOsgDefaultWidget). Typy mają identyczne interfejsy, więc w ramach testów można się między nimi przełączać.

## Wizualizatory i Timeline

Aby rozdzielić silną zależność serii danych dostepnych w wizualizatorach i kanałami w Timeline wyprowadzono w wizualizatorach zamiast jednej klasy bazowej serri danych dwie takie klasy. Obie wspomniane klasy zdefiniowane są w interfejsie IVisualizer.

Pierwsza z nich - [SerieBase](https://www.assembla.com/wiki/show/edytor/SerieBase) jest klasa podstawową wszystkich serii danych. Powinna służyć jedynie dla danych o charaterze statycznym, gdzie zmiana czasu nie odgrywa żadnej roli. Dla danych o charakterze czasowym wyprowadzono drugi typ bazowy - [TimeSerieBase](https://www.assembla.com/wiki/show/edytor/TimeSerieBase). Wspiara on ustawianie z zewnątrz czasu dla danej serii, zmiany jej skali i offsetu. To pozwala nam w prosty sposób opakować takie serie danych w kanały związane z Timeline.

Aby opakowywanie kanałów było prostsze przygotowano 2 klasy usprawniające przełanczanie pomiędzy seriami danych wizualizatoró a kanałami timeline. Obie klasy mają wspólny interfejs bazowy [IVisualizerChannel](https://www.assembla.com/wiki/show/edytor/IVisualizerChannel). Interfejs ten dziedziczy po interfejsie kanałów z Timeline i dodatkowo rozszerza go o czystowirtualną metodę releaseChannel().

Pierwsza z klas - VisualizerChannel - służy do opakowania w kanał pojedynczej serii danych wizualizatora i umieszczenia jej w timeline. Druga z nich - VisualizerMultiChannel - pozwala w formie jednego kanału Timeline zawrzeć kilka serii danych o charakterze czasowym. Drugą klase wykorzystyje się między innymi przy rozkładaniu [VectorChannel](https://www.assembla.com/wiki/show/edytor/VectorChannel) na 3 niezależne składowe skalarne i dodawaniu ich do wizualizatora typu Chart. W ten sposób mamy ze sobą ściśle powiązane czasowo 3 składowe wektora, na wizualizatorze występuja 3 niezależne serie danych a w Timeline mamy jeden zbiorczy kanał dla nich. Usunięcie tego kanału z Timeline skutkuje usunięciem wszystkich 3 serii z nim związanych z wizualizatora/ów.

Nalezy tutaj omówić idee metody releaseChannel implementowanej w wymienionych klasach. Jej potrzeba wynika z faktu iż najogólniejszy wizualizator, gdzie uzytkownik sam zarządza typem wizualizatora i jego widocznych serii danych na bazie dostepnych danych sam zarządza seriami i kanałami. Z drugiej strony mamy mechanizm zarządzania tymi elementami w wyniku operacji użytkownika na drzewie reprezentującym dane. Działanie metody relaseChannel() ma na celu poinformować dany kanał wizualizatorów iż serie danych z nim związane zostały już usunięte z wizualizatorów i nie ma potrzeby ich ponownego usuwania, co skutkowałoby błedami w momencie niszczenia takiego kanału. Nalezy zwrócić uwage że kanały takie są niszczone dopiero w momencie niszczenia serwisu Timeline, a więc w momencie wyładowywania pluginów. Operacje te następują juz po zniszczeniu aplikacji Qt, a więc wizualizatory i ich widgety nie istnieją. Dlatego usuwając kanał z Timeline w trakcie działania aplikacji mamy automatycznie zapewnione usunięcie związanych z nim serii danych z wizualizatorów a niszczenie kanału w momencie zamknięcia aplikacji jest bezpieczne bo wczesniej zwalniamy kanały wizualizatorów metodą releaseChannel().

PODSUMOWANIE:

Zarządzając z zewnątrz seriami danych i kanałami w TImeline:

1) Tworzymy serię danych:

auto serie = vis->createSerie(data, name);

2) Próbujemy uzyskać serię o charakterze czasowym:

auto timeSerie = boost::dynamic\_pointer\_cast<IVisualizer::TimeBaseSerie>(serie);

3) Jeżeli seria ma charakter czasowy zamykamy ją w kanale dla wizualizatorów:

VisualizeChannelPtr channel(new VisuzalizerChannel(serie,visu)); //visu to VisualizerWidget w którym osadzony jest nasz wizualizator, dla którego wygenerowalismy serię

4) Dodajemy taki kanał do timeline w standardowy sposób:

a) przeszukujemy serwisy w poszukiwaniu interfejsu dla Timeline:

TimelinePtr timeline = core::queryServices<ITimelineService>(core::getServiceManager());

b) jeżeli usługa istnieje próbujemy dodać kanał i zapamietac że taki kanał z serii wizualizatorów utowrzyliśmy:

try{

   timeline->addChannel(path.toStdString(), channel);

   visualizerChannels.insert(channel); //istotne dla poprawnego zwalniania kanałów wizualizatorów w momencie niszczenia aplikacji, zmienna visualizerChannels wykorzystywana w destruktorze klasy HmmMainWindow

}catch(...){

   LOG\_ERROR("Could not add visualizer channel to timeline!");

  }

## Uruchamianie przykładowego pluginu

**Wymagania:**

Język wykorzystywany przy tworzeniu edytora i pluginów to C++0x, więc dla systemu Windows są one tworzenie w środowisku Visual Studio 2010. Projekt pluginu najlepiej utworzyć za pomocą narzędzia CMake; przykład został utworzony w wersji 2.8.4 tego narzędzia, ale najlepiej korzystać zawsze z najnowszej wersji.

**Kroki potrzebne do skompilowania przykładowego pluginu (przy użyciu CMake)**:

**1. Ściągnąc plugin z repozytorium svn**

**2. Ściągnąc paczkę z bibliotekami z serwera ftp (dev/plugin\_libs)**

**3. Rozpakować biblioteki do głównego folderu (gdzie istnieje już folder plugins).**

**4. Uruchomić program CMake.**

**5. W polu "Where is the source code" ustawić folder  "plugins\src"**

**6. W polu "Where to build the binaries" ustawić jakiś własny folder np. [główny folder]\bin**

**7. Nacisnąc "Configure"**

  Proces konfiguracji powinien przebiegać bez błędów (nie należy się przejmować, jeśli jakaś wersja biblioteki istnieje tylko w wersji release).

**8. Nacisnąć "Generate"**

  Jeśli wszystko przebiegło bezbłędnie, to w tym momencie, powinna istnieć poprawna solucja w folderze ustawionym w punkcie 6.

**Kroki potrzebne aby umożliwić debuggowanie pluginu (CMake)**

  Niestety nie wszystko da się ustawić z poziomu CMake. Co prawda powinna istnieć już możliwość kompilowania pluginu, debuggowanie jednak wymaga kilku dalszych kroków.

**0. Ustawić aktywną konfigurację na** [**RelWithDebInfo**](https://www.assembla.com/wiki/show/edytor/RelWithDebInfo)

Ustawia się to w Solution Configurations na pasku narzędziowym (wersja debug nie będzie chciała działać z wersją instalacyjną aplikacji, wersji release nie będzie się dało debuggować.

**1. Ustawić Output Directory**

W Properties - > Configuration Properties -> General -> Output Directory należy ustawić folder Application Data\plugins zainstalowanej aplikacji (najlepiej zobrazuje to przykład z życia wzięty : "C:\Users\Wojtek\[AppData](https://www.assembla.com/wiki/show/edytor/AppData)\Local\PJWSTK\EDR\plugins"

**2. Ustawić Command i  Working Directory**

W Properties - > Configuration Properties -> Debugging  należy ustawić :

dla Command ścieżkę do pliku core.exe zainstalowanej aplikacji (np. : "C:\Program Files (x86)\EDR\bin\core.exe"):

dla Working Directory folder gdzie znajduje się plik core.exe (np. : "C:\Program Files (x86)\EDR\bin")

**3. Ustawić plugin\_example1 jako startup project**

Należy nacisnąć prawy przycisk myszy na projekcie a następnie "Set as Startup Project"

// Zmieniły się niektóre założenia i poniższe opisy nie są aktualne!

**Kroki potrzebne do skompilowania przykładowego pluginu (bez użycia CMake):**

**1. Ściagnąć plugin z repozytorium svn**

**2. Ustawić konfigurację na tryb "Release"**

**3. Dodać potrzebne nagłówki**

Należy usunąć wcześniej ustawione w projekcie (properties\configuration properties\c/c++\general\additional include directories). Następnie ustawić poniższe nagłówki:  
    - z paczki ściągniętej z ftp :  
                include\qt  
                include\boost  
                include\utils  
    - z edr/include  
    - z bin (potrzebne core\Config.h)

**4. Dodać potrzebne biblioteki** (properties\configuration properties\linker\input\additional dependencies):  
    - z paczki  
    - bardzo ważne jest, aby biblioteki z paczki zgadzały się z bibliotekami z którymi została zlinkowana aplikacja.

   Aby skompilować plugin potrzebne są następujące: libboost\_system-mt.lib; libboost\_filesystem-mt.lib; [QtGui4](https://www.assembla.com/wiki/show/edytor/QtGui4).lib; [QtCore4](https://www.assembla.com/wiki/show/edytor/QtCore4).lib

**5. Ustawić dyrektywy preprocesora** (properties\configuration properties\C/C++\Preprocessor)  
   \_\_WIN32\_\_; WIN32; NDEBUG ;\_WINDOWS; \_WINDLL; \_USRDLL; BOOST\_ALL\_NO\_LIB; EDRTEST\_EXPORTS;

Po tych krokach projekt powinien się poprawnie skompilować.

**Kroki potrzebne aby umożliwić debuggowanie pluginu.**

Możliwość debuggowania istnieje, jeśli skopiuje się utworzoną bibliotekę dll do folderu bin w zainstalowanej aplikacji, a następnie w odpowiedni sposób uruchomi aplikację z poziomu Visual Studio. Można zrealizować to na kilka sposobów, poniższy jest  jednym z nich.

**1. Utworzyć Post-[BuildEvent](https://www.assembla.com/wiki/show/edytor/BuildEvent" \o "BuildEvent)** (  properties\configuration properties\Build Events\Post-Build Event )

W polu Command Line wpisać (uwzględniając właściwe ścieżki)

copy $([SolutionDir](https://www.assembla.com/wiki/show/edytor/SolutionDir" \o "SolutionDir))$(Configuration)\$([ProjectName](https://www.assembla.com/wiki/show/edytor/ProjectName)).dll "C:\Program Files (x86)\EDR\bin\plugin\_test.dll"

Uwaga 1 - Aplikacja załaduje plugin jedynie jeśli jego nazwa zgadza się z przyjętą konwencją (plugin\_\*.dll).

Uwaga 2 - W przypadku systemów Vista / Windows 7 potrzebne są prawa własności aby możliwe było skopiowanie tego pliku. Można objeść ten problem na wiele sposobów np. można uruchomić Visual Studio w trybie administratora, można też stworzyć plik np. postBuild.bat z powyższą komendą, skrót do tego pliku postBuild.lnk, w zaawansowanych właściwościach tego skrótu ustawić "uruchom jako administrator", a Command Line naszego Eventa wpisac scieżke do utworzonego skrótu (np. $([SolutionDir](https://www.assembla.com/wiki/show/edytor/SolutionDir" \o "SolutionDir))postBuild.lnk)  
      
**2. Ustawić ścieżki** (properties\configuration properties\debug):  
    'Command' na plik core.exe w folderze aplikacji (np. C:\Program Files (x86)\EDR\bin\core.exe)  
    'working directory' na folder gdzie jest aplikacja (konkretniej core.exe; np. C:\Program Files (x86)\EDR\bin)

**3. Uaktywnić debuggowanie w trybie Release**

W tym celu należy ustawić odpowiednie opcje :

W properties\configuration properties\c/c++\general - Debug Info powinno być ustawione na "Program Database".

W properties\configuration properties\linker\debugging - Generate Debug Info powinno być ustawione na YES.

Po tych czynnościach powinna być możliwość uruchamiania i debuggowania pluginu.

**Struktura katalogów**

Edytor, w aktualnej wersji, zakłada istnienie trzech drzew katalogów. Są nimi:

1. [**UserData**](https://www.assembla.com/wiki/show/edytor/UserData)- Katalog z którego wczytywane są próby pomiarowe i przykłady.  Domyślnie jest on powiązany z dokumentami użytkownika. Dla systemu windows będzie to folder "Moje dokumenty" z sufiksem EDR. Przykład - "C:\Users\Wojtek\Documents\EDR"
2. [**ApplicationData**](https://www.assembla.com/wiki/show/edytor/ApplicationData) - Katalog, do którego działająca aplikacja może zapisywać dane w razie potrzeby. Dla windowsa jest to folder [AppData](https://www.assembla.com/wiki/show/edytor/AppData)\Local\ z odpowiednim sufiksem - PJWSTK\EDR. Przykład - "C:\Users\Wojtek\[AppData](https://www.assembla.com/wiki/show/edytor/AppData)\Local\PJWSTK\EDR"
3. **Resources** - Wskazuje na katalog "resources" w foldrze, gdzie zainstalowana została aplikacja. Zawiera jej lokalne zasoby (np. ikony). Przykład - "C:\Program Files (x86)\EDR\bin\resources" .

Interfejs [IDataManager](https://www.assembla.com/wiki/show/edytor/IDataManager) udostępnia 3 metody, które umożliwiają dostęp do omówionych folderów.

    //! zwraca pelna sciezke do folderu "[MyDocuments](https://www.assembla.com/wiki/show/edytor/MyDocuments" \o "MyDocuments)\EDR"  
    virtual const Path& getUserDataPath() const = 0;  
    //! zwraca pelna sciezke do folderu "[ApplicationData](https://www.assembla.com/wiki/show/edytor/ApplicationData" \o "ApplicationData)\EDR"  
    virtual const Path& getApplicationDataPath() const = 0;  
    //! zwraca pelna sciezke do folderu z zasobami aplikacji  
    virtual const Path& getResourcesPath() const = 0;

Do interfejsu [IDataManager](https://www.assembla.com/wiki/show/edytor/IDataManager) można dostać się w bardzo prosty sposób. Należy skorzystać z funkcji core::getDataManager().

Pobranie ścieżki dla ikony znajdującej się w którymś z podkatalogów drzewa **Resources** będzie wyglądało następująco:

    boost::filesystem::path p = core::getDataManager()->getResourcesPath() / "icons/video.png"

## Łączenie pluginów z zewnętrznymi bibliotekami

Aby CMake rozpoznał i skonfigurował zależności pluginów od zewnętrznych bibliotek (np. bibliotek z edrutils) należy w katalogu edr\src\[CMakeModules](https://www.assembla.com/wiki/show/edytor/CMakeModules" \o "CMakeModules) umieścić konfigurację CMake dla poszukiwania konretnej biblioteki, z której ma kożystać nasz plugin. Tak jak w tym pliku zdefiniujemy nazwę tej biblioteki do takiej zależności będziemy musieli się odwoływać pisząc skrypt CMake dla projektu naszego plugina. Przykładowy i najprostszy plik konfigurcaji CMake do poszukiwania bibliotek zewnętrznych można podejrzeć w gotowych już skryptach w wymienionym katalogu (np. [FindKinematicLib](https://www.assembla.com/wiki/show/edytor/FindKinematicLib) czy [FindVdf](https://www.assembla.com/wiki/show/edytor/FindVdf)). Są to zwykłe pliki tekstowe o nazwach zaczynających się słowem Find i kończące się nazwą modułu którego mają poszukiwać.

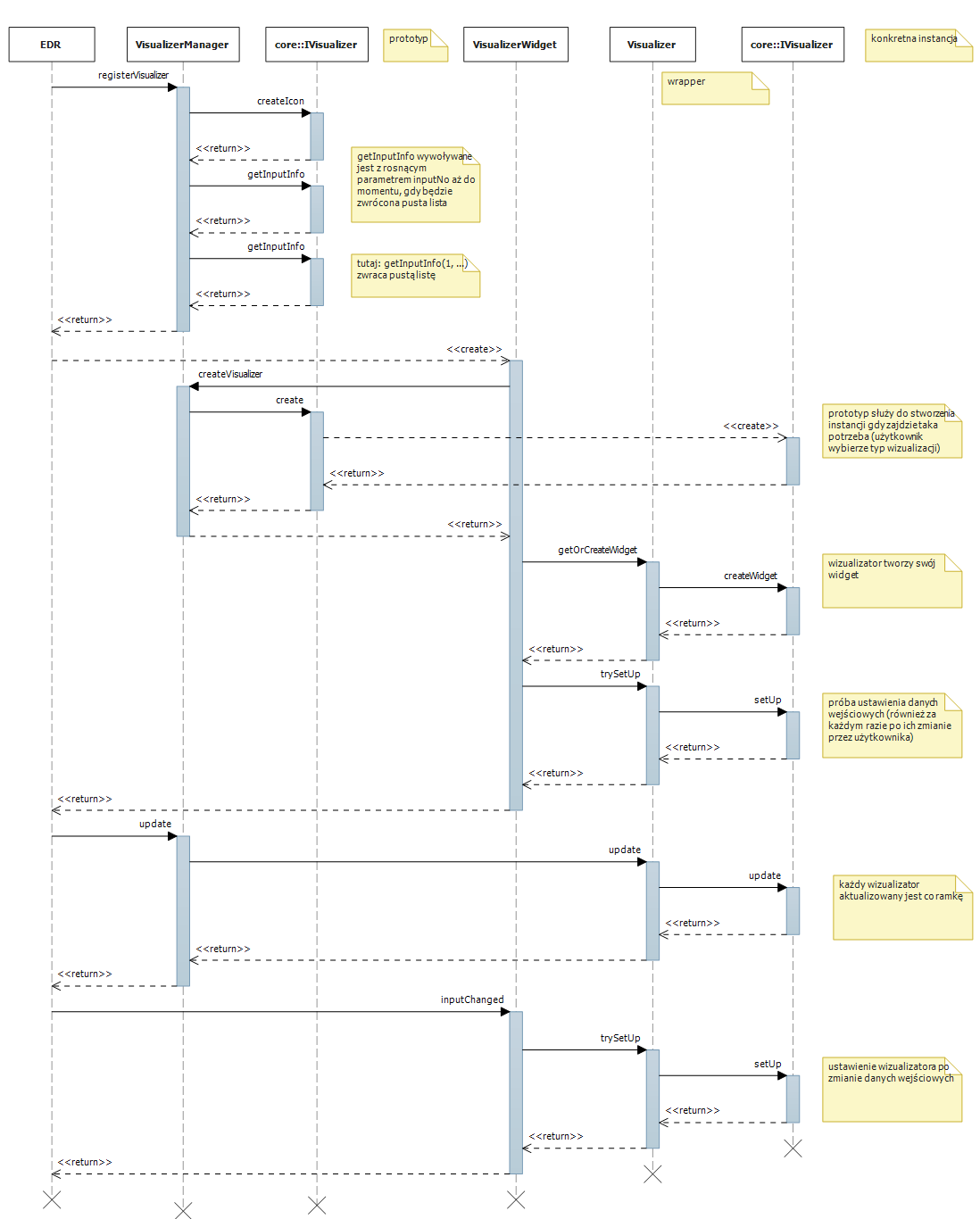
## Obiekty domenowe a polimorfizm przy użyciu RTTI

Mechanizm RTTI (http://en.wikibooks.org/wiki/C++\_Programming/RTTI) jest bardzo ograniczony. Typ std::type\_info, zwracany przez typeid, zawiera znacznie mniej informacji niż np. klasa Type z języka C#. Co istotne, nie jest dostępna informacja na temat klasy bazowej danego typu, nie jest możliwe też uzyskanie typów klas pochodnych.   
  
Obiekty zwracane przez funkcję core::queryDataPtr<T>(core::[IDataManager](https://www.assembla.com/wiki/show/edytor/IDataManager" \o "IDataManager)\*, bool) mogłyby zatem być poprawnie zwracane dla konkretnego typu, który został dostarczony obiektowi [ObjectWrapperPtr](https://www.assembla.com/wiki/show/edytor/ObjectWrapperPtr). Inaczej mówiąc jeśli obiekt zostanie ustawiony metodą set<T> to możliwe byłoby jedynie pobranie z niego obiektu klasy T, nie można pobrać klasy bazowej do T! Obrazuje to poniższy przykład:

Zdefiniowana jest hierarchia:  
[**ScalarChannel**](https://www.assembla.com/wiki/show/edytor/ScalarChannel) **<- C3DAnalogChannel <- EMGChannel.**  
  
Ustawienie [ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper) realizowane jest w taki sposób:  
core::[ObjectWrapperPtr](https://www.assembla.com/wiki/show/edytor/ObjectWrapperPtr" \o "ObjectWrapperPtr) emgChannel;  
...  
[EMGChannelPtr](https://www.assembla.com/wiki/show/edytor/EMGChannelPtr) ptr(new EMGChannel(\*data , i));  
emgChannel->set<EMGChannel>(ptr);   
  
Dla tak ustawionego obiektu można pobrać EMGChannel:  
std::vector<core::EMGChannel> emgChannels = core::queryDataPtr(dataManager);  
Natomiast nie ma możliwości pobrania tych danych jako C3DAnalogChannel:  
std::vector<core::C3DAnalogChannel> c3dChannels = core::queryDataPtr(dataManager);  
  
Szczęśliwe problem można rozwiązać, w tym celu należy stworzyć stworzyć specjalizację typu [ObjectWrapperKey](https://www.assembla.com/wiki/show/edytor/ObjectWrapperKey). Zamiast wdawać się w szczegóły implementacyjne wygodniej jest użyć istniejącego już makra : CORE\_DEFINE\_WRAPPER. Aby zachować naszą hierarchie klas należy utworzyć kilka tych makr jawnie wskazując, jak te typy są ułożone w hierarchii:  
  
CORE\_DEFINE\_WRAPPER([ScalarChannel](https://www.assembla.com/wiki/show/edytor/ScalarChannel" \o "ScalarChannel), utils::[PtrPolicyBoost](https://www.assembla.com/wiki/show/edytor/PtrPolicyBoost" \o "PtrPolicyBoost));  
CORE\_DEFINE\_WRAPPER(C3DAnalogChannel, [ScalarChannel](https://www.assembla.com/wiki/show/edytor/ScalarChannel));  
CORE\_DEFINE\_WRAPPER(EMGChannel, C3DAnalogChannel);  
  
Dzięki takiemu zabiegowi tworzona jest multimapa w klasie [DataManager](https://www.assembla.com/wiki/show/edytor/DataManager) i dla pojedynczego obiektu np. EMGChannel channel tworzone sa wpisy:  
<typeid([ScalarChannel](https://www.assembla.com/wiki/show/edytor/ScalarChannel" \o "ScalarChannel)), channel>  
<typeid(C3DAnalogChannel), channel>  
<typeid(EMGChannel), channel>  
  
Z std::type\_info związane są dwa dodatkowe, dość znaczące problemy. Obiekt type\_info ma prywatny konstruktor kopiujacy i prywatny operator przypisania. Aby przechowywac takie obiekty należy opakować je jakąś własną klasą lub strukturą. W EDR type\_info jest opakowane przez struct core::Type. Drugim problemem jest porównywanie. Jak się okazuje, operator porównania nie zawsze zwraca równości przy jednakowych typach. Przykładowo może zdarzyć się, że typeid(std::vector<int>) != typeid(std::vector<int>) jeśli wektory pochodzą z różnych bibliotek. Porównywanie powinno odbywać się po nazwach (std::type\_info.name()).

## Ogolna architektura aplikacji

# Wizualizatory



## Parsery i wczytywanie danych

# Założenia

* Kod kliencki (usług) powinien móc ładować dane domenowe (związane z przetwarzaniem ruchu postaci ludzkiej) w wysoce abstrakcyjny sposób
* Kod kliencki nie powinien wprost używać ścieżek i nazw plików z danymi domenowymi
* Kod kliencki w ogóle nie powinien działać w przestrzeni plików, tylko w przestrzeni danych
* System danych powinien być w łatwy sposób rozszerzalny bez rekompilacji całej aplikacji i wszystkich usług
* Powinny być ładowane tylko te dane, które są używane

# Problemy

* Niektóre typy danych pochodzą z OSG, niektóre są nasze, inne mogą pochodzić z jeszcze innych źródeł - nie da się stworzyć klasy bazowej i użyć prostego poliorfizmu
* Do obiektów OSG powinno się odwoływać przez osg::ref\_ptr, do pozostałych danych przez boost::shared\_ptr - niezgodność typów
* W edytorze nie wiemy, na jakich typach danych przyjdzie pracować - konsekwencja modułowości
* Niektóre obiekty danych są w relacji 1 do 1 w stosunku do plików, czasem pliki zawierają wiele obiektów danych (np. c3d)

# Rozwiązanie

## Dodawanie typów danych

### core::ObjectWrapper

* Ominięcie ograniczeń związanych z niejednorodną naturą danych
* Przechowuje wskaźnik do obiektu wg parametryzowanych zasad (policy)
* Wrappery definiuje się makrem CORE\_DEFINE\_WRAPPER; drugim parametrem jest albo zasada używania wskaźników (utils::[PtrPolicyBoost](https://www.assembla.com/wiki/show/edytor/PtrPolicyBoost" \o "PtrPolicyBoost) lub utils::[PtrPolicyOsg](https://www.assembla.com/wiki/show/edytor/PtrPolicyOsg" \o "PtrPolicyOsg)), albo jakiś typ, dla którego już zdefiniowano wrapper

namespace core {  
typedef DataChannel<float, float> ScalarChannel;  
class C3DAnalogChannel : public ScalarChannel {};  
class EMGChannel : public C3DAnalogChannel {};  
typedef shared\_ptr<EMGChannel> EMGChannelPtr;  
}  
CORE\_DEFINE\_WRAPPER(ScalarChannel, utils::PtrPolicyBoost);  
CORE\_DEFINE\_WRAPPER(EMGChannel, ScalarChannel);  
CORE\_DEFINE\_WRAPPER(GRFChannel, ScalarChannel);

* Typu tego nie używa się wprost nigdzie poza własną implementacją parserów

core::ObjectWrapperPtr wrapper;  
// stworzenie wrappera  
wrapper = core::ObjectWrapper::createWrapper<core::EMGChannel>();  
// ustawienie obiektu we wrapperze  
wrapper->set<core::EMGChannel>(EMGChannelPtr(new EMGChannel());  
// pobranie obiektu z parsera  
EMGChannelPtr channel = wrapper->get<core::EMGChannel>();

### core::IParser

* Bardzo prosty interfejs na parser - warstwa pośrednia pomiędzy parserem (najczęściej w formie biblioteki zależnej) a logiką edytora
* Czyli: warstwa logiki powinna być bardzo cienka i polegać na wyekstrahowaniu obiektów danych z zewnętrznych, nieuwikłanych w EDR parsów
* Implementacje dostarczane przez pluginy
* Precyzyjnie określony cykl życiowy: stworzenie->przypisanie do pliku->usunięcie
* Nie da się cofnąć do poprzedniego etapu
* Jeden plik nigdy nie jest parsowany wiele razy
* Nigdy nie jest przypisywany wielu plikom
* Implementacje to jedyny kod użytkownika mający wprost do czynienia z core::[ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper" \o "ObjectWrapper), ale ten kod powinien się wyłącznie ograniczyć do stworzenia instancji wrapperów (podczas tworzenia parsera) oraz przypisaniu im obiektów (podczas parsowania)
* **W konstruktorze trzeba stworzyć wszystkie instancje core::[ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper" \o "ObjectWrapper), do których będzie się przypisywać dane podczas parsowania**
* **Parser stanowi również fabrykę dla samego siebie (metoda create)**
* Jako przykład polecam zobaczyć C3DParser oraz [VideoParser](https://www.assembla.com/wiki/show/edytor/VideoParser) (TODO: dodać linki)

## Pobieranie danych

### core::queryData i core::queryDataPtr

* Funkcja globalna będąca liściem całej architektury
* Usługi powinny wyłuskiwać obiekty danych tylko przy ich pomocy, nie odwołując się przy tym ani do core::IParser, ani do core::[ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper" \o "ObjectWrapper)
* Uwalnia od obowiązków myślenia o plikach, ich wczytywaniu, obecności danych itp.

    std::vector<core::EMGChannelPtr> channels = core::queryData<core::EMGChannel>(dataManager);  
    BOOST\_FOREACH( core::EMGChannelPtr channel, channels ) {  
        // zrób coś z kanałem  
    }

* Gdy nie pamięta się typu inteligentego wskaźnika (tutaj [EMGChannelPtr](https://www.assembla.com/wiki/show/edytor/EMGChannelPtr)) można użyć słówka auto (kompilator sam rozpozna typy)

    auto channels = core::queryData<core::EMGChannel>(dataManager);  
    BOOST\_FOREACH( auto channel, channels ) {  
        // zrób coś z kanałem  
    }

* Istnieje też metoda queryDataPtr, której parametrem jest nie typ obiektu, a typ inteligentnego wskaźnika

    // wersja pierwsza, najbardziej przenośna ale najbardziej rozwlekła  
    std::vector<core::GRFChannelPtr> channels1;  
    core::queryDataPtr(dataManager, channels1);  
  
    // wersja jednolinijkowa, ale z dwukrotnym podaniem typu  
    std::vector<core::GRFChannelPtr> channels2 = core::queryDataPtr<core::GRFChannelPtr>(dataManager);  
  
    // wersja jednolinijkowa, pojedyncze podanie typu dzięki słowu kluczowemu auto  
    auto channels3 = core::queryDataPtr<core::GRFChannelPtr>(dataManager);  
  
    // polecana wersja jednolinijkowa, pojedyncze podanie typu dzięki idiomowi Return Type Resolver  
    std::vector<core::GRFChannelPtr> channels4 = core::queryDataPtr(dataManager);  
  
    // wersja powodująca błąd kompilacji - bez podanego typu :)  
    // auto channels5 = core::queryDataPtr(dataManager);  
  
    // SILNIE ODRADZANE użycie queryDataPtr2 w BOOST\_FOR\_EACH - powoduje wykonanie kopii wektora  
    BOOST\_FOREACH( auto channel, core::queryDataPtr<core::GRFChannelPtr>(dataManager) ) {};  
  
    // równoważne powyższemu, ta sama uwaga  
    BOOST\_FOREACH( core::GRFChannelPtr channel, core::queryDataPtr<core::GRFChannelPtr>(dataManager) ) {};  
  
    // wersja z błędem kompilacji  
    // BOOST\_FOREACH( core::GRFChannelPtr channel, core::queryDataPtr(dataManager) ) {};  
  
    // zalecana wersja  
    std::vector<core::GRFChannelPtr> channels6 = core::queryDataPtr(dataManager);  
    BOOST\_FOREACH( core::GRFChannelPtr /\*lub auto\*/ channel, channels6 ) {}  
  
    // również zalecana  
    auto channels7 = core::queryDataPtr<core::GRFChannelPtr>(dataManager);  
    BOOST\_FOREACH( core::GRFChannelPtr /\*lub auto\*/ channel, channels6 ) {}

## Sposób działania

Makro CORE\_DEFINE\_WRAPPER tworzy specjalizację typu core::[ObjectWrapperT](https://www.assembla.com/wiki/show/edytor/ObjectWrapperT" \o "ObjectWrapperT), dziedziczącego po core::[ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper" \o "ObjectWrapper) albo po innej specjalizacji. Dzięki temu [ObjectWrappery](https://www.assembla.com/wiki/show/edytor/ObjectWrappery) mają swoją hierachię klas poddającą się polimorfizmowi, a dodatkowo dzięki przechowywaniu type\_info parametru szablonu można to przekładać na hierarchię klas faktycznych obiektów danych. W każdym razie z instancji core::[ObjectWrapper](https://www.assembla.com/wiki/show/edytor/ObjectWrapper" \o "ObjectWrapper) można pobrać listę typów (znowu type\_info), które są wspierane. Dzięki temu można zbudować multimapę indeksowaną po typie. Od tego już krótka droga do pobierania wyłuskiwania faktycznych danych z wrapperów.

### Rejestracja parserów

[DataManager](https://www.assembla.com/wiki/show/edytor/DataManager) (DM) stanowi agregat parserów. Parsery rejestrowane są podczas inicjacji aplikacji. Te instancje nigdy nie będą przypisane do żadnych plików i zawsze będą służyć tylko jako fabryki - do wywołań metody create(). Aby ułatwić sprawę parsery mapowane są pod rozszerzenia, które obsługują (multimapa, konflikty rozszerzeń między parserami trzeba jeszcze rozwiązać).

### Ładowanie próby

Ładowania próby nie należy mylić z ładowaniem zasobów. Podczas ładowania próby tworzone są instancje parserów dla każdego z plików, którego rozszerzenie znalazło się w słowniku rozszerzeń, ale **nie są one jeszcze parsowane**. Żeby ułatwić zarządzanie DM przechowuje te nowe instancje obudowane wewnętrznym typem Parser, który dostarcza pewnych informacji o stanowości (np. czy instancja została już użyta). Po przeskanowaniu katalogu próby tworzy się więc lista obiektów typu [DataManager](https://www.assembla.com/wiki/show/edytor/DataManager)::Parser, które obudowują nowe instancje core::IParser. Każdemu stworzeniu parsera towarzyszy odpytanie go o obiekty danych. Naturalnie te obiekty nie będą na nic jeszcze wskazywały (nie powinny, w końcu nie było parsowania), ale już mogą dostarczać informacji o wspieranych typach, dzięki czemu można stworzyć multimapę typów oraz obiektów do nich przypisanych.

### Ładowanie zasobów

Tak jak zostało wcześniej wspomniane, żadne zasoby nie są ładowane przed wystąpieniem zapotrzebowania. Zapotrzebowanie to zgłaszają funkcje core::queryData oraz core::queryDataPtr. Przekierowują one zapytanie do DMa, który skanuje multimapę obiektów przypisanych do typów oraz sprawdza, następującego schematu:

* Czy obiekt został już wcześniej załadowany?
  + Tak: dodaj go do listy wynikowej
  + Nie: czy parser z którego obiekt pochodzi został już użyty?
    - Tak: błąd danych albo błąd w implementacji parsera
    - Nie: dokonaj parsowania; Czy obiekt został załadowany?
      * Tak: dodaj go do listy wynikowej
      * Nie: błąd danych albo błąd w implementacji parsera

**Organizacja logiki usług**

Artykuł ten postał na bazie wiadomości https://www.assembla.com/spaces/edytor/messages/599887.

Sposób, w jaki kod dostarczany przez usługi jest wywoływany musi być ściśle określony. Przy tworzeniu modułów obliczeniowo/wizualizacyjnych trzeba wziąć pod uwagę następujące problemy:

1. Wymagające obliczenia muszą być wykonywane w osobnym wątku (wątkach).
2. Tylko wątek UI może bez przeszkód modyfikować UI (właściwość Qt).
3. Usługi nie znają i nie mają wpływu na kolejność aktualizacji, przez co jeżeli jedna usługa odwołuje się do drugiej, to stan tej drugiej może dotyczyć albo bieżącej, albo poprzedniej ramki.

Problemy te zostały rozwiązane poprzez wprowadzenie trzech przebiegów aktualizacyjnych:

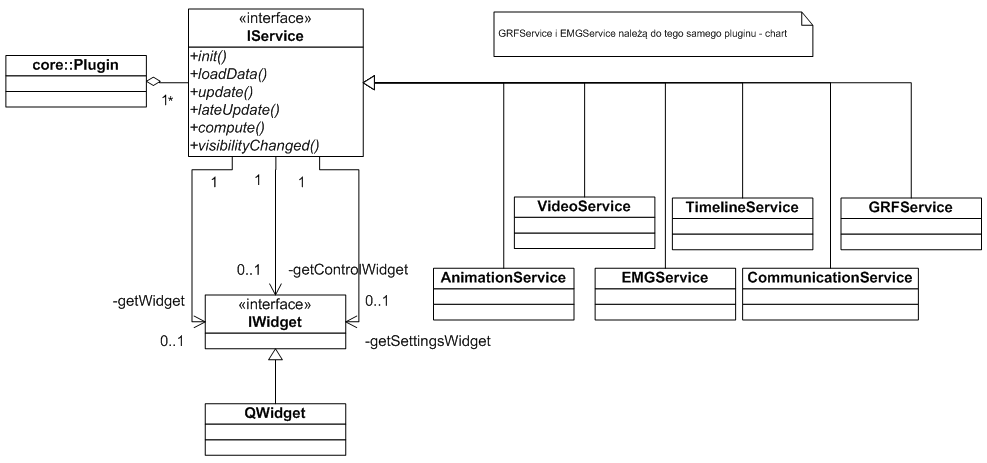
1. Proste obliczenia/modyfikacja UI następują w implementacji metody update. W jej ciele usługi nie powinny odwoływać się do innych usług.
2. Jeżeli usługa korzysta lub kontroluje inne usługi (np. timeline) powinna implementować metodę lateUpdate; jest ona wywoływana po wykonaniu update dla wszystkich usług, a więc ich stan jest ustalony. Oczywiście czasem może zachodzić sytuacja, gdy potrzebny byłby jeszcze jeden przebieg (gdy np. występuje łańcuch zależności trzech usług); problem ten zostanie rozwiązany z wprowadzeniem kontroli przepływu aktualizacji między usługami.
3. Skomplikowane obliczenia powinny być wykonywane w implementacji metody compute. Metoda ta nie powinna bezpośrednio modyfikować UI, lecz przekazywać dane tak, aby implementacja update mogła zrobić z nich pożytek przy niskim nakładzie obliczeniowym. Dzięki takiemu podziałowi można wywoływać metodę compute w osobnych wątkach, nie zamrażając w ten sposób UI nawet dla bardzo złożonych obliczeń.

**Pluginy usługi parsery**

Obecnie namespace core’a jest zaśmiecony wskutek nietrzymania się ogólnych postanowień. Docelowo core powinien być fasadą, bardzo prostą aplikacją zdolną ładować i kontrolować usługi, abstrahując przy tym od ich przeznaczenia. Usługi dostarczane są przez pluginy (moduły dynamiczne). Syntezując:

* plugin: kontener na usługi; jego definicja polega na wylistowaniu zawieranych usług
* usługa: logika biznesowa wstrzykiwana do edytora, mogąca mieć wizualną reprezentację w postaci widgetów; jej definicja polega na implementacji interfejsu IService
* parser: **obecnie niezaimplementowane**; będzie tym dla fizycznych danych czym usługi są dla logiki biznesowej

Zależności między tymi pojęciami (uwzględniając usługi rozwijane przez nas) przedstawiają się następująco:



Plugin poza metodami kontrolującymi logikę posiada metody zwracające widgety: widok, kontrola i ustawienia. Żaden z tych widgetów nie jest obowiązkowy. Widgety te są umieszczane w odpowiednich panelach z [Organizacja\_UI](https://www.assembla.com/wiki/show/edytor/Organizacja_UI).

**UWAGA**: obecnie interfejs IWidget nie jest nigdzie zdefiniowany, zamiast tego używamy forward-declaration:

class IWidget;

Skutkiem tego ta interfejsowość widgetów jest czysto umowna, bowiem po stronie usług QWidgety są z użyciem reinterpret\_cast rzutowane na IWidget, a po stronie edytora następuje proces odwrotny. Równie dobrze zamiast IWidget mógłby występować wskaźnik void\*. Proponuję porzucenie tych pozorów i jawne używanie typu QWidget.

## Logowanie i obsługa błędów

Usługi w naszej aplikacji aby być uruchamiane, muszą jawnie spełniać nałożone przez nas wymagania. Podobnie powinno być z raportowaniem błędów i logowaniem informacji diagnostycznych - zgodnie z ideą, że "feeling" i zachowanie ui będą jednorodne w ramach całego edytora. Tutaj niestety jest to już kwestia wyłącznie dyscypliny, gdyż nie ma prostych mechanizmów językowych i projektowych, które uniemożliwiłyby programiście wyświetlanie informacji przy użyciu cout, printf, ::[MessageBox](https://www.assembla.com/wiki/show/edytor/MessageBox" \o "MessageBox), osg::notify, [OutputDebugString](https://www.assembla.com/wiki/show/edytor/OutputDebugString), strumieniów plikowych itd.

Czemu właściwie to regulować? Przy odrobinie dyscypliny można kod raportujący ograniczyć wyłącznie do raportowania właściwego (tj. konstruowania wiadomości), a całą resztę, czyli sprawdzanie poziomów ważności wiadomości, wyświetlanie dialogów, równoczesne wyświetlanie wiadomości w konsoli systemowej i konsoli edytora, śledzenie źródeł, itd, pozostawić implementacji loggera.

## Logowanie

Do logowania powinniśmy używać makr:

LOG\_DEBUG(x)  
LOG\_ERROR(x)  
LOG\_INFO(x)  
LOG\_WARNING(x)

Parametr x to zawartość wpisu. W dużym uproszczeniu instrukcję

LOG\_INFO("Hello world!");

można traktować jako

std::cout<<"Hello world!"<<std::endl;

Z tego w ramach x można używać operatorów strumieniowych:

float x, y;  
...  
LOG\_INFO("Współrzędna x: " << x << "; Współrzędna y: " << y);

Jeżeli jesteś bardzo przywiązany do formatowania w stylu C (printf) można używać biblioteki boost::format:

float x, y;  
...  
LOG\_INFO(boost::format("X=%f, Y="%f") % x % y);

Dokładny opis biblioteki boost::format (w wersji używanej przez nas) znajduje się w http://www.boost.org/doc/libs/1\_42\_0/libs/format/doc/format.html.

To, gdzie zalogowana informacja się pojawi, określone jest na podstawie konfiguracji. Na ten temat napisane będzie więcej.

## Video

# 1         Video Service - Core (video)

## 1.1       Ogólne założenia

Ponieważ projekt zakłada przetwarzanie czterech strumieni HD (1920x1080) jednocześnie, konieczne jest dokonywanie optymalizacji na każdym kroku przetwarzania obrazu. W szczególności należy unikać kopiowania, zbędnych konwersji między formatami lub zmian rozmiaru z kosztownymi filtrami.

## 1.2       Funkcjonalność i skalowalność

Obecnie używaną biblioteką do dekodowania plików video jest FFmpeg na licencji LGPL. Na wypadek, gdyby warunki licencyjne uległy zmianie na np. bardziej restrykcyjne GPL , warto jest zapewnić pewien poziom abstrakcji oraz zdefiniować minimalny zestaw funkcji, które konkretna implementacja musi dostarczać.

Rysunek 1. Hierarchia klas strumieni video. Abstrakcyjne metody zaznaczone są kursywą.

Jak widać wyżej, klasa bazowa strumienia [VideoStream](https://www.assembla.com/wiki/show/edytor/VideoStream) zawiera tylko 5 abstrakcyjnych metod (plus jedna wirtualna). Prowadzi to do sytuacji, gdzie konkretne implementacje muszą dostarczyć wyłącznie absolutnie bazową (dla strumienia) logikę jak wskazanie i ustawianie pozycji oraz przekazanie wskaźników na odczytaną ramkę. Dodatkowo na tym etapie, celem utrzymania prostoty, w ogóle nie są poruszane tematy wielowątkowości lub zaawansowanego IO.

## 1.3       Typy strumieni

### 1.3.1       FFmpegVideoStream

[FFmpegVideoStream](https://www.assembla.com/wiki/show/edytor/FFmpegVideoStream) to strumień, który do odczytu plików video używa biblioteki FFmpeg. Wewnętrznie jego logika jest maksymalnie prosta, ale wykorzystuje najnowsze dodatki do wersji biblioteki. Nie następuje wobec tego żadne kopiowanie danych, zmiana rozmiaru bądź formatu.

Kulejącą funkcjonalnością jest przewijanie strumienia. FFmpeg, dla używanych formatów video, potrafi wykonać skok wyłącznie do ramki kluczowej. Wszelkie niekluczowe ramki muszą być odczytywane sekwencyjnie. Wobec tego przewijanie do tyłu zawsze polega na przejściu do ramki kluczowej przed zadanym czasem, a następnie odczyt ramek aż do określonego punktu. Przewijanie do przodu otwiera pole do optymalizacji w tym zakresie – nie zawsze trzeba przeskakiwać do ramki kluczowej, gdyż czasem szybciej jest po prostu odczytywać sekwencyjnie ramki aż do zadanego czasu. Kryterium określające kiedy wykonać skok a kiedy odczytywać ramki zostało dobrane empirycznie na podstawie pomiarów.

### 1.3.2       BufferedVideoStream

Klasa [BufferedVideoStream](https://www.assembla.com/wiki/show/edytor/BufferedVideoStream) jest eksperymentalną implementacją, która buforuje część wewnętrznego strumienia, umożliwiając tym samym szybsze przewijanie, jeżeli ramka o zadanym czasie jest w buforze. Uaktywnia się ją przełącznikiem --buffered.

Mechanizm buforów opiera się na typie [VideoBuffer](https://www.assembla.com/wiki/show/edytor/VideoBuffer) agregującym typy [VideoBufferChunk](https://www.assembla.com/wiki/show/edytor/VideoBufferChunk). Zaimplementowane mechanizmy pozwalają na wyszukiwanie ramki w czasie logarytmicznym, wykrywanie nieciągłości bądź automatyczne łącznie komplementarnych fragmentów.

## 1.4       Reprezentacja obrazu (PictureCommon, Picture, PictureLayered)

Zazwyczaj strumienie video są kodowane w formacie YUV, gdzie zazwyczaj każdy komponent teksela znajduje się w osobnym obszarze pamięci. Z kolei biblioteki takie jak [OpenSceneGraph](https://www.assembla.com/wiki/show/edytor/OpenSceneGraph) pozwalają wyłącznie na używanie obrazów, w których wszystkie dane znajdują się w jednym obszarze pamięci (np. RGB lub BGRA). Zaproponowana reprezentacja obrazu (struktury pochodne po [PictureCommon](https://www.assembla.com/wiki/show/edytor/PictureCommon)) zapewniają konwersję między oboma typami z minimalnym narzutem (przejście ze skompresowanych danych do reprezentacji wielowarstwowej nie wymaga żadnych alokacji, w drugą stronę niestety należy sporządzić kopię).

Dodatkowo, ponieważ omawiane typy są strukturami, w łatwy sposób można je przetwarzać i kopiować. Mimo tego jest zachowana pewna abstrakcja, a to dzięki niedostępnemu publicznie typowi [PictureContext](https://www.assembla.com/wiki/show/edytor/PictureContext), który nadaje buforom rozmiar, który wynika z formatu oraz rozdzielczości obrazka.

## 1.5       Konwersja między formatami (Converter)

FFmpeg dostarcza również funkcji służących do szybkiej konwersji między różnymi formatami obrazów. Możliwości te zostały zamknięte w klasie Converter, która jest używana wewnętrznie przez strumienie, jeżeli aplikacja będzie chciała otrzymać ramkę w formacie, który jest różny od formatu strumienia.

Rysunek 2. Hierarchia struktur reprezentujących obraz.

# 2         Video Service – OSG plugin (video::osgPlugin)

## 2.1       Ogólne założenia

Biblioteka vm\_core została zaprojektowana tak, żeby nie była zależna od żadnych zewnętrznych bibliotek poza tymi, które służą do dekodowania strumienia video. Biblioteka vm\_OSGplugin jest zwolniona z tego obowiązku i służy do powiązania strumieni video z logiką osgDB, czyli części [OpenSceneGraph](https://www.assembla.com/wiki/show/edytor/OpenSceneGraph) odpowiedzialnej za IO.

## 2.2       Strumienie zgodne z OSG - VideoImageStream

Strumienie z biblioteki vm\_core muszą być odpowiednio opakowane, aby móc integrować się z [OpenSceneGraph](https://www.assembla.com/wiki/show/edytor/OpenSceneGraph). Rozwiązaniem zgodnym z duchem tej biblioteki jest rozszerzenie typu osg::[ImageStream](https://www.assembla.com/wiki/show/edytor/ImageStream" \o "ImageStream), który jest abstrakcyjną bazą dla wszystkich obrazków zmieniających się w czasie. Rolę tę pełni [VideoImageStream](https://www.assembla.com/wiki/show/edytor/VideoImageStream) – klasa-pośrednik między logiką OSG a logiką vm\_core.

### 2.2.1       Autonomiczność

Obecnie odtwarzeniem strumieni kieruje moduł Timeline Manager (TM) będący osobną częścią projektu. Z czasów, gdy jeszcze go nie było, zachowała się zdolność typu [VideoImageStream](https://www.assembla.com/wiki/show/edytor/VideoImageStream) do autonomicznego odtwarzania w osobnym wątku. W związku z tym mimo faktu, że obecnie stan strumieni video kontrolowany jest przez osobny moduł, typ [VideoImageStream](https://www.assembla.com/wiki/show/edytor/VideoImageStream) jest w pełni funkcjonalny i gotowy do użycia w sytuacji, gdzie nie ma obiektu nadrzędnego.

### 2.2.2       BufferedVideoImageStream

[BufferedVideoImageStream](https://www.assembla.com/wiki/show/edytor/BufferedVideoImageStream) to eksperymentalne, niedostępne publicznie rozszerzenie typu [VideoImageStream](https://www.assembla.com/wiki/show/edytor/VideoImageStream) o mechanizm asynchronicznego antycypacyjnego buforowania. Potencjalnie pozwoli rozwiązać problemy z wydajnością na słabszych maszynach, chociaż wiąże się ze znacznie zwiększonym zapotrzebowaniem pamięciowym. Algorytm w przybliżeniu opisuje Rysunek 4.

Rysunek 3. Hierarchia dziedziczania [VideoImageStream](https://www.assembla.com/wiki/show/edytor/VideoImageStream).

## 2.3       Integracja z OpenSceneGraph

Miejscem integracji pluginu i OSG jest klasa [ReaderWriterVMPlugin](https://www.assembla.com/wiki/show/edytor/ReaderWriterVMPlugin). Dzięki temu, gdziekolwiek w kodzie pojawi się próba odczytania pliku będącego strumieniem video, zostanie zwrócony obiekt obrazka osg::Image, który można potem rzutować na osg::[ImageStream](https://www.assembla.com/wiki/show/edytor/ImageStream" \o "ImageStream). Aplikacja kliencka nie musi więc znać nagłówków bibliotek VM, gdyż otrzymuje dane w postaci standardowych klas [OpenSceneGraph](https://www.assembla.com/wiki/show/edytor/OpenSceneGraph). Istnieje oczywiście możliwość rzutowania na typ [VideoImageStream](https://www.assembla.com/wiki/show/edytor/VideoImageStream) aby uzyskać specyficzną funkcjonalność, jednak w większości przypadków wystarczy warstwa abstrakcji zapewniona przez osg::[ImageStream](https://www.assembla.com/wiki/show/edytor/ImageStream" \o "ImageStream).

Dzięki takiemu zaprojektowaniu w znaczny sposób rośnie uniwersalność kodu. Opieranie się na klasach bazowych OSG sprawia, że komponenty mogą współpracować z funkcjami bibliotecznymi. Działa to również w drugą stronę – przykładowo, można by pokusić się o zastąpienie strumieni video np. plikami GIF lub innymi z sekwencją obrazków. W tym momencie wszystkie komponenty operujące uprzednio na strumieniu video za pośrednictwem osg::[ImageStream](https://www.assembla.com/wiki/show/edytor/ImageStream" \o "ImageStream) powinny ad hoc działać z innymi, w tym bibliotecznymi, rozszerzeniami tego typu.

Rysunek 4. Algorytm buforowania antycypacyjnego.

Rysunek 5. [ReaderWriterVMPlugin](https://www.assembla.com/wiki/show/edytor/ReaderWriterVMPlugin)

## Communication

# ****Opis****

Serwis komunikacji powinien obejmować zakresem transport plików oraz przetwarzanie danych w zakresie komunikacji. Całość jest zaprojektowana w ten sposób, aby szczegóły implementacji czy użytych technologii nie miały dużego wpływu na serwis. Dzięki takiemu rozwiązaniu klient jest niezależny od serwera, a sposób komunikacji może być zmieniany bez większych ingerencji w projekt.

Usługa transportu plików odnosi się do operacji pozwalających wysyłać i odbierać pliki między serwerem i klientem. Ze względu na dotychczas użyte technologie, operacje te koncentrują się na wysyłaniu i odbieraniu plików przy pomocy połączenia FTP oraz działaniach na bazie danych przy pomocy usług sieciowych WSDL.

Przetwarzanie danych polega na tworzeniu zapytań z danych wejściowych od klienta zrozumiałych dla serwera oraz odbieraniu rezultatów zapytań, formatowanie i prezentacja tych danych. W chwili obecnej komunikacja z serwerem na płaszczyźnie przetwarzania danych również odbywa się przy pomocy usług sieciowych.

# ****Użyte technologie****

Obecnie używa się wymienionych wcześniej FTP oraz Web Services. Serwer używany przez klienta znajduje się pod adresem:

v21.pjwstk.edu.pl

w przypadku problemów z adresowaniem można użyć adresu IP:

83.230.112.43

Połączenia FTP są szyfrowane za pomocą SSL, usługi sieciowe zostały uruchomione przy pomocy protokołu HTTP i wymagają autentykacji NTLM.

### ****FTPS****

Szyfrowany protokół FTP jest odpowiedzialny za wysyłanie i pobieranie plików między serwerem, a lokalnym komputerem klienta. Do zestawienia połączenia używamy adresu:

ftps://83.230.112.43/

nazwy użytkownika:

testUser

i hasła:

testUser

W projekcie implementacją FTPS zajmuje się biblioteka libcurl napisana w C. Libcurl to darmowa i wieloplatformowa biblioteka na licencji MIT/X pozwalająca na korzystanie z wielu protokołów sieciowych. Więcej informacji o samej bibliotece można znaleźć na stronie domowej projektu: <http://curl.haxx.se/libcurl/>.

### ****Web Services****

Usługi sieciowe zdefiniowane przy pomocy WSDL, są odpowiedzialne za komunikacje z bazą danych. Moduł Communication korzysta z definicji zawartych pod adresami:

http://v21.pjwstk.edu.pl/Motion/res/BasicQueriesWSStandalone.wsdl

http://v21.pjwstk.edu.pl/Motion/FileStoremanWS.svc?wsdl

Zestawienie połączenia wymaga autentykacji NTLM. Używana nazwa użytkownika:

applet\_user

hasło użytkownika:

aplet4Motion

Dokument [FileStoremanWS](https://www.assembla.com/wiki/show/edytor/FileStoremanWS) odpowiada za informacje związane z pobieraniem i wysyłaniem plików na serwer i operacje opisane w tym dokumencie są ściśle powiązane z klientem FTP. W przypadku wysyłania pliku na serwer, najpierw plik jest transportowany na serwer FTP, następnie uruchamiana jest operacja dokonująca odpowiedniego wpisu w bazie. Gdy plik jest pobierany, najpierw odpowiednia operacja dokonuje zapytania w bazie i przygotowuje plik do transportu z serwera na lokalny komputer.

Implementacja usług WSDL odbywa się przy pomocy biblioteki wsdlpull, wieloplatformowego API wydanego na licencji LGPL. Dostarcza wielu narzędzi pozwalających na prace z web serwisami przy pomocy języka C/C++. Informacje o bibliotece można znaleźć na stronie projektu <http://sourceforge.net/projects/wsdlpull/>. Ze względu na rodzaj autentykacji wykorzystywany przez usługi sieciowe, biblioteka musi być kompilowana z biblioteką libcurl.

# ****Szczegóły implementacji****

Moduł został dostosowany do edytora EDR: jest zbudowany jako plugin dołączany do aplikacji głównej, wykorzystuje model MVC, widok wykorzystuje bibliotekę QT. Funkcjonalność Communication zawiera się w interfejsie **ICommunication**.

Poniżej znajduje się lista i opis najważniejszych klas:

### ****CommunicationService****

Klasa implementująca interfejs ***ICommunication*** i ***IService***. Jest odpowiedzialna za udostępnienie funkcjonalności całego modułu, oraz konfiguracje połączeń, pełni rolę kontrolera modułu. Łączy moduł z aplikacją udostępniając widok.

### ****CommunicationManager****

Klasa pełniąca rolę modelu modułu. Kopia informacji o zasobach serwerowych znajduje się w pliku **Communication**, który jest plikiem tekstowym w formacie xml. Dane o zasobach lokalnych są wyciągane z edytora przy pomocy [***DataManagera***](https://www.assembla.com/wiki/show/edytor/DataManagera).

Pobieranie i uaktualnianie zasobów serwerowych odbywa się w osobnym wątku, z powodu komplikacji między wątkami i biblioteką QT, aktualizacja widoku odbywa się przy pomocy klasy [***CommunicationService***](https://www.assembla.com/wiki/show/edytor/CommunicationService).

Klasa korzysta ze wzorca singleton.

### ****QueryWSDL****

Implementuje interfejs ***IQueryable*** dodający funkcjonalność odpytywania bazy danych. Jak wskazuje nazwa klasy, odpytywanie bazy odbywa się przy pomocy technologii WSDL.

### TransportWSDL\_FTPS

Implementuje interfejs ***ITransportable*** dodający funkcjonalność transportu danych. Ze względu na użyte technologie, klasa jest mieszanką operacji wsdlowych wyciągających pliki z bazy danych i poleceniami ftp pozwalającymi pobierać pliki z serwera na dysk lokalny.

# ****Diagram klas****

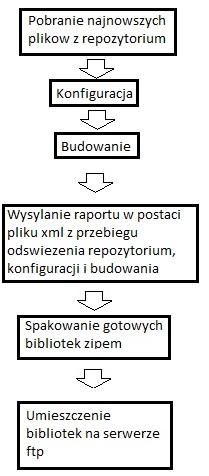
## Proces ciaglej integracji

# Rejestracja

By uczestniczyc w procesie ciągłej integracji nalezy zarejestrowac się na dashboardzie pod adresem: http://83.230.112.38/index.php . Klikamy w link register. Nastepnie wypelniamy pola i rejestrujemy sie. W kolejnym oknie logujemy sie do do dashboardu za pomoca adresu email i hasla podanego przy rejestracji. Subskrybujemy interesujacy nas projekt. Wybieramy role normalnego uzytkownika. W zakladce cvs/svn login wpisujemy swoj login z assembli. Email preferences oraz email category ustawiamy wedle swego uznania.

# Istota procesu CI

Schematyczny proces ciaglej integracji bibliotek przedstawi poniższy graf:



Proces ten jest wykonywany w rownych z gory okreslonych odstepach czasu, osobno dla wersji release oraz debug kazdej z bibliotek, pod windowsem i linuxem. W chwili obecnej wykonywany jest codziennie o 12 w nocy.

Budowe można uruchomić ręcznie poprzez wejscie na stronę: http://83.230.112.37/edr.php

Dokumentacja biblioteki DFMLib:

Przeznaczenie biblioteki:

Biblioteka jest przeznaczona do tworzenia sieci węzłów wraz z połączeniami między węzłami. Ma ona umożliwiać przepływ danych przez taką sieć węzłów. Innymi słowy można tworzyć graf skierowany, gdzie różne węzły mogą być ze sobą łączone i wymieniać dane w ściśle określony sposób. Każdy węzeł może mieć wiele wejść i wyjść, których używa się do łączenia węzłów ze sobą i wymiany informacji między nimi. Biblioteka jest inspirowana inną biblioteką – WireIt napisaną w JavaScript. Ponadto wzorowano się na aplikacji Orange.

Nazewnictwo:

|  |  |
| --- | --- |
| **Nazwa** | **Opis** |
| **Węzeł** | Element grupujący piny wejściowe i wyjściowe |
| **Pin** | Element węzła pozwalający go połączyć z innymi węzłami przez ich piny,  Wyróżniamy piny wejściowe i wyjściowe na tym etapie analizy |
| **Węzeł źródłowy** | Węzeł bez pinów wejściowych |
| **Węzeł liść** | Węzeł bez pinów wyjściowych lub nie posiadający połączenia wyjść z innymi węzłami |
| **Kompatybilność pinów** | Informacja o tym czy dane piny mogą być ze sobą połączone, odpowiedzialność leży zawsze po stronie pinu przyjmującego – wejściowego, on dokonuje sprawdzenia |
| **Wymagany pin** | Pin wejściowy, który musi mieć połączenie z innym pinem, gdyż wymaga tego funkcjonalność węzła |
| **Zależnośc pinów wyjsciowych** | Aktywność pinów wyjściowych może być uzależniona od podłączenia odpowiednich pinów wejściowych. W razie braku podłączenia wymaganego pinu wejściowego dany węzeł nie będzie realizował swojej funkcjonalności |
| **Połączenie** | Więź logiczna pomiędzy dwoma węzłami poprzez parę pinów – wejściowego jednego węzła i wyjściowego drugiego węzła |
| **Model** | Zarządza węzłami I połączeniami między nimi, przechowuje strukturę grafu/sieci |
| **Poprawność modelu** | Model jest poprawny, kiedy wszystkie zasady łączenia węzłów są zachowane |
| **Pętla/Cykl** | Z teorii grafów, możliwość powrotu do jednego z węzłów przechodząc przez kolejne podpięte węzły w kierunku od wejść do wyjść |
| **Przepływ danych** | Wymiana i przetwarzanie danych w węzłach, przepływ danych w zdefiniowanym kierunku – od pinów wejściowych do wyjściowych w węźle i pinów wyjściowych i wejściowych między węzłami |
| **Cykl danych** | Czas potrzebny na pełne przetworzenie danych w modelu – od węzłów źródłowych do wszystkich węzłów liści. |

Struktura biblioteki:

Warstwa logiczna (model sieci):

Bazując na ogólnym opisie przeznaczenia biblioteki oraz przytoczonych pojęciach można wyszczególnić podstawowy poziom funkcjonalności biblioteki DFMLib. Poziom ten odpowiada za łączenie węzłów, tworzenie węzłów, zarządzanie węzłami i połączeniami między nimi. Na tym poziomie jest obsługiwana kompatybilność pinów oraz zależność pinów wyjściowych od wejściowych. Na tym poziomie nie zajmujemy się danymi, ich przepływem i przetwarzaniem.

Węzły reprezentowane są przez instancje klasy Node, piny przez instancje klasy Pin a połączenia przez instancje klasy Connection. Zarządzanie modelem realizowane jest w instancji klasy Model.

Warstwa wymiany i przetwarzania danych:

Warstwa wyżej odpowiedzialna jest za przepływ danych oraz ich przetwarzanie. Warstwa ta w kodzie reprezentowana jest przez rozszerzenia poprzednich klas z prefiksem DF (od Data Flow). W tej warstwie następuje walidacja modelu odnośnie ustalonych reguł łączenia i reprezentacji modelu. Walidacja jest robiona lokalnie, przy każdej zmianie w modelu (dodanie węzła, usunięcie węzła, dodanie połączenia, usuniecie połączenia).

Jak już wspomniano klasy Node, Pin, Model zostały rozszerzone do klas DFNode, DFPin and DFModel aby realizować tak sprecyzowaną funkcjonalność. Wymianę i przetwarzanie danych osiągnięto się poprzez interfejs klasy DFInterface.

Zaimplementowane zasady połączeń między węzłami sprawdzane on-line (nie zezwalające na operację łączenia):

W klasie Model można połączyć dwa węzły wtedy i tylko wtedy gdy:

* Łączymy pin wyjściowy jednego węzła z pinem wejściowym drugiego węzła ( dokładnie w takiej kolejności, graf skierowany),
* Pin wyjściowy jest kompatybilny z pinem wejściowym patrząc ze strony pinu wejściowego (on dokonuje walidacji kompatybilności, on pobiera dane z pinu wyjściowego),
* Nie istnieje połączenie pomiędzy danymi pinami (brak redundantnych połaczeń w modelu).

W klasie DFModel można połączyć węzły jeśli warunki klasy Model zostały spełnione oraz:

* Połączenie dwóch węzłów nie wprowadzi cyklu/pętli do grafu,
* Pin wejściowy nie jest jeszcze podłączony (jednoznaczna identyfikacja źródła danych – co najwyżej jedno połączenie przychodzące).

Realizacja wymiany danych:

Wymiana danych jest realizowana w sposób asynchroniczny przez węzły. Sam przepływ danych przez graf jest realizowany synchronicznie – węzły wejściowe wprowadzają dane do modelu, dane muszą dotrzeć do węzłów liści i być przez nie przetworzone, aby było możliwe wprowadzenie kolejnej porcji danych przez węzły wejściowe. Przepływ danych pomiędzy węzłami jest oparty na mechanizmie informowania o dostępności danych w danym węźle (konkretnie w jego poszczególnych pinach). Informacje między węzłami są wymieniane za pomocą pinów, zaś ich przetwarzanie odbywa się w węźle. Węzeł jest odpowiedzialny za weryfikowanie dostępnych danych na swoich wejściach, stwierdzenie kompletności tych danych (dostępność danych na wszystkich wejściach, które są podłączone) przetworzenie tych danych. Przetworzone dane są następnie kierowane w odpowiedniej formie do poszczególnych pinów wyjściowych, które po otrzymaniu danych do przekazania poprzez podpięte połączenia informują piny wejściowe kolejnych węzłów (związanych z danym połączeniem) o dostępności danych. Każdy pin wejściowy na taka informację informuje rodzica (węzeł) o dostępności nowych danych. Węzeł rozpoczyna weryfikację, czy wszystkie niezbędne dane są dostępne i jeśli tak to proces przetwarzania i przekazywania danych zostaje rozpoczęty na tym wźle wg opisanego już schematu.

Ponadto węzły wejściowe są zablokowane aż do momentu kiedy wszystkie liście skończą przetwarzanie danych. Wtedy, jeśli węzły wejściowe posiadają jeszcze jakieś dane do przetworzenia mogą je umieścić w modelu i ich proces przetwarzania rozpocznie się wg wcześniej opisanego schematu.

Z tej prostej przyczyny natychmiastowe wstrzymanie działania modelu może być niemożliwe – jeśli aktualnie w modelu są przetwarzane jakieś dane, to muszą one przejść przez cały model aż do liści, dopiero wtedy może zostać wstrzymane dostarczenie nowych danych do modelu przez źródła. Wstrzymanie modelu następuje więc natychmiast po zakończeniu aktualnego cyklu danych.

Biblioteka nie rozważa żadnych szczególnych typów danych. Jest to zadanie użytkownika, aby zaimplementować przetwarzanie danych oraz ich przekazywania pomiędzy węzłami w pinach i węzłach oddzielnie.

Zależność pinów wyjściowych:

Niektóre piny wyjściowe mogą być zależne od pinów wejściowych. Jeśli pin wyjściowy jest zależny od pinu wejściowego, który nie jest jeszcze podłączony, to taki pin wyjściowy jest nieaktywny w tym sensie, iż nie będzie mógł przekazywać danych dalej. Jeśli podłączymy taki niekompletny pin wyjściowy z innym węzłem to walidacja modelu się nie powiedzie.

Walidacja modelu:

Jak już wcześniej wspomniano walidacja modelu jest robiona lokalnie przy każdej zmianie modelu – dodaniu/usunięciu węzła oraz dodaniu/usunięci połączenia. W każdej chwili można odpytać model o listę miejsc (węzłów i pinów), gdzie podstawowe zasady zostały złamane. Po uzupełnieniu takich braków model powinien być już poprawny i jego uruchomienie będzie możliwe. Braki reprezentowane są jako mapa węzłów do których dołączona jest lista pinów sprawiających problemy. Jeśli jakiś węzeł ma pustą listę pinów to znaczy że nie jest on wcale podłączony z innymi węzłami co również jest niedozwolone – wszystkie węzły w modelu muszą być podłączone w sposób poprawny – minimum jedno wejście węzła musi być podłączone, nawet gdy nie jest ono wymagane.

Ponadto w modelu musi znajdować się przynajmniej jeden węzeł źródłowy i jeden węzeł przetwarzający dane, które muszą być ze sobą poprawnie połączone.

Dokumentacja biblioteki VDFMLib

Przeznaczenie biblioteki:

Biblioteka VDFMLib przeznaczona jest do wizualnej edycji modelu przepływu danych opisanego w DFMLib. Z pomocą tej biblioteki użytkownik może graficznie tworzyć sieć połączeń miedzy węzłami, tym samym samemu definiując jakiego typu dane i ich analizy go interesują. Graficzna realizacja bazuje na bibliotece OpenSceeneGraph (OSG). Biblioteka wspiera użytkownika w tworzeniu modelu poprzez wskazywanie braków w modelu i błędów wskazując takie miejsca graficznie. Dąży się do uzyskania rozwiązania podobnego w bibliotece Orange. Klasy z przedrostkiem DF są reprezentowane przez analogiczne klasy z przedrostkiem osgVDF.

Pomysły:

* Biblioteka wizualizuje węzły, piny i połączenia.
* Toolbar z 4 grupami węzłów:
  + Source (źródłowe)
  + Analysis (analiza)
  + Processing (przetwarzające)
  + Visualize (prezentujące).
* W każdej grupie wyświetlają się typy węzłów które można stworzyć, powiązane z tą grupą.
* Użytkownik przez klik na danym typie węzła w toolbarze aktywuje proces jego tworzenia. Po kolejnym kliknięciu na przestrzeni użytkownika, gdzie przechowywane są węzły zadany węzeł zostanie stworzony, zwizualizowany i dodany do modelu logicznego.
* Tak długo jak użytkownik nie kliknie ponownie na toolbar, na dany przycisk węzła kolejne kliknięcia na przestrzeni użytkownika będą tworzyć nowe węzły wybranego typu.
* Jeśli użytkownik aktywuje inny typ węzła na toolbarze poprzez kliknięcie jego przycisku to proces tworzenia nie zostanie przerwany, jedynie inny typ węzła będzie tworzony w przestrzeni użytkownika
* Węzły można usuwać
* Łączenie pinów odbywa się za pomocą metody Drag & Drop (przeciągnij i upuść), przez chwycenie jednego pinu, przeciągnięcie myszki i upuszczenie na pinie docelowym. Jeśli piny są kompatybilne i nie ma innych zastrzeżeń dla takiego połączenia zostanie ono utworzone i zwizualizowane.
* Podczas procesu przeciągnij i upuść przy łączeniu pinów podświetlają się piny kompatybilne z pinem od którego cały proces się rozpoczął.
* Węzły i piny nie powinny na siebie nachodzić. Połączenia mogą się krzyżować gdyż konfiguracja pinów w węzłach może uniemożliwić rozwiązanie tego problemu.
* Węzły powinny być wyrównywane do siatki.
* Węzły można zaznaczać
* Zaznaczoną grupę węzłów można przesuwać – znajduje się ona na pierwszym planie
* Raz stworzony model musi być zapisywalny i wczytywalny z pliku.

**TODO!! Grupowanie węzłów, zarządzanie grupami!!**

Klasy przestrzeni osgVDF:

|  |  |
| --- | --- |
| **Klasa** | **Opis** |
| **osgVDFBaseModel** | klasa bazowa modeli graficznych dla data flow, pochodna klasy osgWidget::WindowManager + osgUI::KeyboardMapper + osgUI::TooltipHost, oferuje podstawową funkcjonalność oraz API do wpięcia własnych elementów graficznych |
| **osgVDFBaseNode** | klasa bazowa graficznych węzłów, jest pochodną klasy osgWidget::Window |
| **osgVDFBasePin** | klasa bazowa graficznych pinów, jest pochodną klasy osgWidget::Widget |
| **osgVDFNodeTypeDescriptor** | klasa opisująca typ węzła na potrzeby jego graficznej realizacji i możliwości jego tworzenia. Za jej pomocą można zarejestrować dany węzeł w graficznym modelu, aby potem go móc tworzyć |
| **osgVDFModel** | template realizacja klasy osgVDFBaseModel, pozwala stosować własne typy węzłów, pinów, selekcji i połączeń, które spełniają zdefiniowane w kodzie wymagania |
| **osgVDFNode** | konkretna realizacja klasy osgVDFBaseNode, zarządza graficznymi pinami, ma ściśle zdefiniowaną strukturę graficzną i style |
| **osgVDFPin** | template realizacji klasy osgVDFBasePin, zarządza wizualną reprezentacją połączeń związanych z danym pinem |

API klasy osgVDFBaseModel:

* bool registerNodeType(osgVDFNodeTypeDescriptor \* descriptor)

rejestruje dany typ węzła w modelu pozwalając tym samym graficznie tworzyć obiekty tego typu węzła

* bool isEmptyArea(const osgWidget::XYCoord & posToStart)

sprawdza czy dany punkt przestrzeni modelu jest pusty (nie ma tam żadnego obiektu)

* bool addNode(dflm::NPtr node, osg::ref\_ptr<osg::Image> image = osg::ref\_ptr<osg::Image>(), const std::string & name = std::string())

dodaje dany węzeł logiczny do graficznej reprezentacji modelu oraz jego logicznego odpowiednika, generuje wizualną reprezentację takiego węzła

* bool deleteNode(dflm::NPtr node)

usuwa węzeł z modelu logicznego i graficznego

* dflm::Model::NODES\_SET getNodesInArea(const osg::BoundingBox & area)

zwraca zbiór węzłów logicznych, których graficzne reprezentacje znajdują się w zadanym obszarze

* virtual osgWidget::XYCoord getFreePlaceForNode(dflm::NPtr node, const osgWidget::XYCoord & posToStart)

metoda poszukująca wolnego miejsca w graficznym modelu dla zadanego węzła poczynając od proponowanej lokalizacji początkowej

* void selectNode(dflm::NPtr node)

metoda zaznaczająca węzeł

* void selectNodes(const dflm::Model::NODES\_SET & nodes)

metoda zaznaczająca zbiór węzłów

* void selectNodes(const osg::BoundingBox & rect)

metoda zaznaczająca węzły w zadanym obszarze

* void selectAllNodes()

metoda zaznaczająca wszystkie węzły modelu

* void deselectNode(dflm::NPtr node)

metoda odznaczająca węzeł

* void deselectNodes(const dflm::Model::NODES\_SET & nodes)

metoda odznaczająca węzły

* void deselectNodes(const osg::BoundingBox & rect)

metoda odznaczająca węzły w zadanym obszarze

* void deselectAllNodes()

metoda odznaczająca wszystkie zaznaczone węzły

* bool isNodeSelected(dflm::NPtr node) const

metoda sprawdza czy dany węzeł jest zaznaczony

* dflm::Model::NODES\_SET getSelectedNodes() const

metoda zwraca zaznaczone węzły logiczne

* osg::BoundingBox getSelectedNodesBoundingBox() const

metoda zwraca ramkę która okala wszystkie zaznaczone węzły

* bool isNodeInCollision(dflm::NPtr node) const

metoda sprawdza czy dany węzeł tworzy wizualnie kolizję z innym węzłem

* bool isNodeInVisibleArea(dflm::NPtr node) const

metoda sprawdza czy węzeł znajduje się w widzialnym obszarze modelu

* bool isNodeFullyInVisibleArea(dflm::NPtr node) const

metoda sprawdza czy węzeł jest w pełni widzialny

* osgWidget::XYCoord getNodePosition(dflm::NPtr node)

metoda zwraca położenie danego węzła

* osgWidget::XYCoord getNodeSize(dflm::NPtr node)

metoda zwraca wielkość danego węzła

* void moveNode(dflm::NPtr node, const osgWidget::XYCoord & moveStep)

metoda przesuwa węzeł o zadany krok względem aktualnej pozycji węzła

* void moveSelectedNodes(const osgWidget::XYCoord & moveStep)

metoda przesuwa zaznaczone węzły o zadany krok względem lewego dolnego rogu ramki ich okalającej

* void setNodePosition(dflm::NPtr node, const osgWidget::XYCoord & nodePos)

metoda pozycjonująca zadany węzeł w wybranym punkcie wg lewego dolnego rogu węzła

* void setSelectedNodesPosition(const osgWidget::XYCoord & nodesPos)

metoda pozycjonuje zaznaczone węzły w zadanym miejscu względem lewego dolnego rogu ramki ich okalającej

* dflm::ConnPtr connect(dflm::PinPtr src, dflm::PinPtr dest)

metoda łączy dwa piny jeśli to możliwe

* bool disconnect(dflm::ConnPtr connection)

metoda usuwająca dane połączenie

* void showDefaultToolbar(bool show)

metoda pokazująca / ukrywająca domyślny toolbar z zarejestrowanymi typami węzłów

* osgWidget::point\_type getAreaRatioToSelect() const

metoda zwraca współczynnik minimalnego graficznego zaznaczenia powierzchni węzła aby móc go dodać do zaznaczenia

* void setAreaRatioToSelect(osgWidget::point\_type areaRatio)

metoda ustawia współczynnik minimalnego graficznego zaznaczenia powierzchni węzła aby móc go dodać do zaznaczenia. Współczynnik ten jest z przedziału (0;1>

* const osgWidget::KeyboardMapper::KEYS\_SET & getSelectionActionKeys() const

metoda zwraca klawisze specjalne przy zaznaczaniu które mogą odwracać zaznaczenie

* void setSelectionActionKeys(const osgWidget::KeyboardMapper::KEYS\_SET & keys)

metoda ustawia klawisze specjalne przy zaznaczaniu które mogą odwracać zaznaczenie

* osgWidget::point\_type getMinDistToDelConnection() const

metoda zwraca promień, który będzie rozpatrywany w przypadku usuwania połączeń po kliknięci na graficznej reprezentacji modelu

* void setMinDistToDelConnection(osgWidget::point\_type dist)

metoda ustawia promień, który będzie rozpatrywany w przypadku usuwania połączeń po kliknięci na graficznej reprezentacji modelu

* osgWidget::Color getVisualConnectionNormalColor() const;

metoda zwraca normalny kolor połączeń

* void setVisualConnectionNormalColor(osgWidget::Color val)

metoda ustawia normalny kolor połączeń

* osgWidget::Color getVisualConnectionHighlightColor() const

metoda zwraca kolor zaznaczonych połączeń

* void setVisualConnectionHighlightColor(osgWidget::Color val)

metoda ustawia kolor zaznaczonych połączeń

* dflm::CDFMPtr getLogicalModel() const

metoda zwraca logiczny model przepływu danych

Abstrakcyjne metody klasy osgVDFBaseModel:

* virtual osg::ref\_ptr<osgVDFBaseNode> createVisualNode(dflm::NPtr node, osg::ref\_ptr<osg::Image> image = osg::ref\_ptr<osg::Image>(), const std::string & name = std::string()) = 0;

metoda tworzy wizualną reprezentację węzła będącą pochodną klasy osgVDFBaseNode na bazie węzła logicznego, dodatkowo może być dostaerczony charakterystyczny obrazek dla tego typu węzłów oraz nazwa węzła

* virtual osg::ref\_ptr<osg::Geode> createVisualConnection(osgVDFBasePin \* pinA, osgVDFBasePin \* pinB) = 0;

metoda tworzy /zwraca wizualne połączenie pomiędzy danymi pinami

* virtual osg::ref\_ptr<osg::Geode> startVisualConnection(osgVDFBasePin \* startPin) = 0;

metoda tworzy /zwraca wizualne połączenie pomiędzy które rozpoczyna się w zadanym pinie

* virtual void setVisualConnectionEnd(osg::Geode \* connection, const osgWidget::XYCoord & endPos) = 0;

metoda kontynuuje uprzednio rozpoczęte zaznaczenie do zadanych współrzędnych

* virtual void setVisualConnectionStart(osg::Geode \* connection, const osgWidget::XYCoord & startPos) = 0;

metoda ustawia punkt początkowy połączenia, aktualizuje kształt połaczenia

* virtual void setVisualConnectionZ(osg::Geode \* connection, osgWidget::point\_type z) = 0;

metoda ustawia warstwę połączenia, służy do chowania /pokazywania połączeń pod / nad węzłami i innymi elementami modelu

* virtual osg::ref\_ptr<osg::Geode> createVisualSelection(const osgWidget::XYCoord & startPos) = 0;

metoda tworzy wizualną selekcję o starcie w zadanym punkcie. Jest to prostokąt, gdzie elementy które okala będą dodane do zaznaczenia

* virtual void continueVisulaSelection(osg::Geode \* selection, const osgWidget::XYCoord & continuePos) = 0;

metoda kontynuuje poprzednio rozpoczętą selekcję do wybranych współrzędnych

* virtual void setVisualConnectionColor(osg::Geode \* connection, const osgWidget::Color & color ) = 0;

metoda ustawia kolor zaznaczenia (ramki)

API klasy osgVDDFBaseNode:

* dflm::NPtr getModelNode() const

metoda zwraca skojarzony z wizualnym węzłem węzeł logiczny

* osgVDFBaseModel \* getModel() const

metoda zwraca wizualny model w którym osadzony jest dany węzeł

* bool addInPin(osgVDFBasePin \* inPin, const std::string & pinName = std::string())

metoda dodaje do węzła wizualną reprezentację pinu wejściowego, opcjonalnie można podać nazwę pinu

* bool addOutPin(osgVDFBasePin \* outPin, const std::string & pinName = std::string())

metoda dodaje do węzła wizualną reprezentację pinu wyjściowego, opcjonalnie można podać wyświetlaną nazwę pinu

* const VISUAL\_PIN\_SET & getInPins() const

metoda zwraca zbiór wizualnych reprezentacji pinów wejściowych

* const VISUAL\_PIN\_SET & getOutPins() const

metoda zwraca zbiór wizualnych reprezentacji pinów wyjściowych

* virtual void setName(const std::string & name)

metoda ustawia nazwę węzła

* virtual std::string getName() const

metoda zwraca nazwę węzła

* void setNodeVisualStatus(NODE\_VISUAL\_STATUS nodeVisualStatus)

metoda ustawia wizualny stan węzła

* NODE\_VISUAL\_STATUS getNodeVisualStatus() const

metoda zwraca wizualny stan

Abstrakcyjne metody klasy osgVDFBaseNode:

* virtual void setName(const std::string & name) = 0;
* virtual std::string getName() const = 0;

API klasy osgVDFBasePin:

* dflm::PinPtr getModelPin() const

metoda zwraca skojarzony z wizualnym pinem pin logiczny

* osgVDFBaseNode \* getParentNode() const

metoda zwraca wizualną reprezentację węzła z którym skojarzony jest aktualny pin

* PIN\_VISUAL\_STATUS getStaticPinVisualStatus() const

metoda zwraca satyczny wizualny stan pinu na bazie jego logicznego stanu w modelu

* PIN\_VISUAL\_STATUS getDynamicPinVisualStatus(dflm::PinPtr refPin, dflm::MPtr model) const

metoda zwraca dynamiczny wizualny stan pinu podczas próby połączenia go z innym pinem ze względu na model logiczny w którym oba piny występują

* PIN\_VISUAL\_STATUS getDynamicPinVisualStatus(osgVDFBasePin \* refPin, dflm::MPtr model) const

metoda zwraca dynamiczny wizualny stan pinu podczas próby połączenia go z innym pinem ze względu na model logiczny w którym oba piny występują

* virtual const osgWidget::XYCoord & getCenterPosition()

metoda zwraca środek pinu, używana podczas tworzenia / edycji wizualnych połączeń

* virtual osgWidget::XYCoord getCenterPosition() const

metoda zwraca środek pinu, używana podczas tworzenia / edycji wizualnych połączeń

* const CONNECTIONS\_MAP & getConnections() const

metoda zwraca mapę kojarzącą wizualne połączenia aktualnego pinu z pinami po drugiej stronie

* void setPinVisualStatus(PIN\_VISUAL\_STATUS pinVisualStatus)

metoda ustawia wizualny stan pinu, używana podczas tworzenia / usuwania połączeń

* PIN\_VISUAL\_STATUS getPinVisualStatus() const

metoda zwraca wizualny stan pinu

Abstrakcyjne metody klasy osgVDFBasePin:

* virtual void graphSetPinStatus(PIN\_VISUAL\_STATUS pinVisualStatus) = 0

metoda modyfikująca wygląd pinu ze względu na zadany wizualny stan pinu, charakterystyczna dla konkretnej implementacji pinu

Style w OSG

OSG natywnie wspiera style dla biblioteki osgWidget. Klasy Window i Widget oraz ich pochodne posiadają metodę setSyle(const std::string & styleName), która pozwala ustawić zadany styl. Sam style rejestruje się w obiekcie klasy StyleManager. Można obiekt tej klasy pobrać z WindoManagera lub stworzyć własny i potem go ustawić w WindowManager jako nowy StyleManager. Style są reprezentowane przez klase Style, która cechuje się nazwą stylu oraz opisem stylu. Opis stylu to ciąg znaków w następującej formie: cecha wartość, gdzie zarówno nazwa cechy jak i format jej wartości są ściśle zdefiniowane. Klasa Style pozwala na zarządzanie cechami podstawowych elementów, w dodatku w dość ograniczonym zakresie. Można ja jednak rozszerzyć i tam obsługiwać własne, rozszerzone cechy obiektów. Tak tez czyni klasa StyleEx, która obsługuje Widgety z ramką (klasa Borderized, implementująca interfejs IBorderized). Rozszerzanie stylów opiera się na implementacji włsnej funkcji applyStyle klasy Style, gdzie możemy rozpoznawać własne cechy i ich wartości a także nieobsługiwane przez Style interfejsy czy obiekty. Poniżej przedstawiono podstawowe cechy klasy Style oraz dodatkowe cechy klasy StyleEx. Warto zaznaczyć że cechy w stylu są oddzielone od siebie znakiem końca linii.

Klasa Style – podstawowe cechy i ich format:

* Widget:
  + pos – ustawia pozycję Widgeta:
    - %i %i (x,y) – wersja dla współrzędnych opartych na liczbach całkowitych
    - %f %f (x,y) - wersja dla współrzędnych opartych na liczbach zmiennoprzecinkowych
  + pos-x – ustawia współrzędna x Widgeta:
    - %i
    - %f
  + pos-y – ustawia współrzedną y Widgeta:
    - %i
    - %f
  + size – ustawia rozmiar Widgeta:
    - %i %i (width, height)
    - %f %f (width, height)
  + width – ustawia szerokośc Widgeta:
    - %i
    - %f
  + height – ustawia wysokość Widgeta:
    - %i
    - %f
  + color – ustawia kolor tła Widgeta:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.0
  + padding – odstęp wokół Widgeta
    - %i – odstęp w pixelach
  + padding-left – odstęp po lewej stronie Widgeta:
    - %i – odstęp po lewej w pixelach
  + padding-right – odstęp po prawej stronie Widgeta:
    - %i – odstęp po prawej w pixelach
  + padding-top – odstęp od gory Widgeta:
    - %i – odstęp od gory w pixelach
  + padding-bottom – odstęp od doły Widgeta:
    - %i – odstęp od dołu w pixelach
  + layer – warstwa Widgeta w oknie, w którym występuje:
    - %w – ciąg znaków, jedna z wartości: top, high, middle, low, bg
  + valign – wyrównanie Widgeta w pionie:
    - %w – ciąg znaków, jedna z wartości: center, top, bottom
  + haling – wyrównywanie Widgeta w poziomie:
    - %w – ciąg znaków, jedna z wartości: center, left, right
  + coordmode – ustawia sposób zadawania współrzednych Widgeta:
    - %w – ciąg znaków, jedna z wartości: absolute, relative
  + fill – ustawia możliwość automatycznego wypełniania wolnej przestrzeni przez Widget:
    - %w – ciąg znaków, jedna z wartości: true, false
  + image – ustawia texture dla Widgeta:
    - %s – ścieżka do tekstur
* Window:
  + pos – ustawia pozycję Window:
    - %i %i (x,y) – współrzedne Window, licby całkowite
    - %f %f (x,y) - współrzedne Window, licby zmiennoprzecinkowe
  + pos-x – ustawia współrzędną x Window:
    - %i – współrzędna x Widnow
    - %f – współrzedna x Window
  + pos-y – ustawia współrzędną y Window:
    - %i – współrzędna y Window
    - %f – współrzędna y Window
  + size – ustawia wielkośc Window:
    - %i %i (width, height)
    - %f %f (width, height)
  + width – ustawia szerokośc Window:
    - %i – szerokośc Window w pixelach
    - %f – szerokośc Window w pixelach
  + height – ustawia wysokość Window:
    - %i – wysokość Window w pixelach
    - %f – wysokość Window w pixelach

Klasa StylEx:

* Ogólnie – Widget, Window, inne:
  + rect\_image – ustawia teksture elementu z optymalizacją:
    - %s – ścieżka tekstury
* IBorderized (interfejs Widgetów z ramką):
  + border\_color – ustawia kolor ramki:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.0
  + border\_width – ustawia grubość ramki:
    - %i – grubośc ramki w pixelach
    - %f – grubość ramki w pixelach
* Label:
  + font\_color – ustawia kolor czcionki dla Label:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + font\_size – ustawia wielkość czcionki dla Label:
    - %i – wielkość czcionki w pixelach
    - %f - wielkośc czcionki wp pixelach
  + font\_type – ustawia typ czcionki dla Label:
    - %s – nazwa czcionki
  + font\_fit – pozwala dopasować wielkość widgeta do wielkości tekstu:
    - %w – jedna z następujących wartości:
      * vertical – dopasowuje widget tylko do szerokości tekstu – wysokość bez zmian
      * horizontal – dopasowuje widget tylko do wysokości tekstu – szerokość pozostaje bez zmian
      * all – pełne dopasowanie wymiarów widgeta do wielkości tekstu
  + font\_align – wyrównanie czcionki w widgecie (NIE ZAIMPLEMENTOWANE!!)
* Interfejs osgUI::DefaultPolicy – obsługuje elementy typu osgUI::Buttonized:
  + hoover\_color – kolor widgeta po najechaniu na niego myszką:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + toggle\_color – kolor widgeta po kliknięci na niego jeśli może zmieniać stan:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + pushed\_color – kolor widgeta w momencie kliknięcia go (wraca do hoovered lub toogled po puszczeniu):
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + normal\_color – domyślny kolor widgeta, gdy żaden z poprzednich stanów nie jest aktywny:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
* Interfejs osgUI::LabelStylePolicy – styl osgWidget::Label w wersji rozszerzonej do osgUI::Buttonized:
  + label\_hoover\_color – kolor tekstu po najechaniu na widget myszką:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + label\_toggle\_color – kolor tekstu po kliknięci widget jeśli może zmieniać stan:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + label\_pushed\_color – kolor tekstu w momencie kliknięcia widgeta (wraca do hoovered lub toogled po puszczeniu):
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + label\_normal\_color – domyślny kolor tekstu, gdy żaden z poprzednich stanów nie jest aktywny:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
* Interfejs osgUI::IBorderStylePolicy – styl widgeta z ramką rozszerzonego o osgUI::Buttonized:
  + border\_hoover\_color – kolor ramki po najechaniu na widget myszką:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + border\_toggle\_color – kolor ramki po kliknięci widget jeśli może zmieniać stan:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + border\_pushed\_color – kolor ramki w momencie kliknięcia widgeta (wraca do hoovered lub toogled po puszczeniu):
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
  + border\_normal\_color – domyślny kolor ramki, gdy żaden z poprzednich stanów nie jest aktywny:
    - %i %i %i %i – RGBA, wartości 0-255
    - %i %i %i – RGB, kanał alpha = 255 (1.0)
    - %f %f %f %f – RGBA, wartości 0.0-1.0
    - %f %f %f – RGB, wartości 0.0-1.0, kanał ał alpha = 1.
* Interfejs osgUI::TextStylePolicy – styl widgeta rozszerzonego o osgUI::Buttonized oparty na stylach dla poszczególnych stanów:
  + hoover\_style – styl stosowany do widgeta w momencie najechania na niego
  + toggle\_style – styl stosowany do widgeta po kliknięciu go, jeśli może on zmieniać swój stan (atrybut togglable)
  + pushed\_style – styl stosowany do widgeta w momencie kliknięcia go
  + normal\_style – styl stosowany do widgeta jeśli nie wystąpił żaden z powyższych stanów.

osgVDF namespace styles:

Schemat węzła:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Border.TL | Border.TM | | | | | Border.TR |
| Border.ML | Body.TL | Body.TNameLabel  (nazwa węzła) | Body.cfgBCG  (konfiguracja węzła) | Body.delBCG  (usuń węzeł) | Body.TR | Border.MR |
| Body.Center | | | | |
| Body.BL | Body.MB | | | Body.BR |
| Border.BL | Border.BM | | | | | Border.BR |

Wyróżnia się następujące stany węzłów:

* selected/active – węzeł/y zaznaczone
* collision – stan węzłów niezaznaczony biorących udział w kolizji podczas zmiany położenia węzłów zaznaczonych
* normal – węzeł niezaznaczony, nie biorący udziału w kolizji, poprawny stan węzła, stan neutralny

Wyróżnia się następujące stany pinów:

* ok – pin jest neutralny, nie trzeba względem niego podejmować żadnych akcji
* connected – pin podłączony, tak zaznaczamy również pin który rozpoczyna proces łączenia
* incompatible – pin niekompatybilny z aktualnie wybranym do połączenia
* required – pin wejściowy wymagany, który nie został jeszcze podłączony
* incomplete – pin wyjściowy, zależny od pinów wejściowych, którego zależność nie jest spełniona
* active – oznacza węzły kompatybilne z aktualnie wybranym

Na bazie stanów jakie wyróżniono oraz schematu węzła można wyszczególnić następujące style dla węzła i pinów.

Style jakie znajdziemy w obecnej implementacji biblioteki VDFMLib dla węzłów i pinów:

* Piny:
  + Wyjściowe:
    - vdf.pin.out.ok
    - vdf.pin.out.active
    - vdf.pin.out.connected
    - vdf.pin.out.incomplete
    - vdf.pin.out.incompatible
  + Wejściowe:
    - vdf.pin.in.ok
    - vdf.pin.in.active
    - vdf.pin.in.connected
    - vdf.pin.in.incompatible
    - vdf.pin.in.required
* Węzły:
  + Ramka:
    - vdf.node.border.TL.normal
    - vdf.node.border.TM.normal
    - vdf.node.border.TR.normal
    - vdf.node.border.ML.normal
    - vdf.node.border.MR.normal
    - vdf.node.border.BL.normal
    - vdf.node.border.BM.normal
    - vdf.node.border.BR.normal
    - vdf.node.border.TL.collision
    - vdf.node.border.TM.collision
    - vdf.node.border.TR.collision
    - vdf.node.border.ML.collision
    - vdf.node.border.MR.collision
    - vdf.node.border.BL.collision
    - vdf.node.border.BM.collision
    - vdf.node.border.BR.collision
    - vdf.node.border.TL.active
    - vdf.node.border.TM.active
    - vdf.node.border.TR.active
    - vdf.node.border.ML.active
    - vdf.node.border.MR.active
    - vdf.node.border.BL.active
    - vdf.node.border.BM.active
    - vdf.node.border.BR.active
  + Ciało węzła:
    - vdf.node.body.TL.normal
    - vdf.node.body.TR.normal
    - vdf.node.body.TNameLabel.normal
    - vdf.node.body.TConfigBCG.normal
    - vdf.node.body.TDeleteBCG.normal
    - vdf.node.body.config.normal
    - vdf.node.body.delete.normal
    - vdf.node.body.BL.normal
    - vdf.node.body.BR.normal
    - vdf.node.body.BM.normal
    - vdf.node.body.Center.normal
    - vdf.node.body.TL.active
    - vdf.node.body.TR.active
    - vdf.node.body.BL.active
    - vdf.node.body.BR.active
* Typy węzłów w domyślnym toolbarze (patrz interfejs osgUI::Buttonized):
  + osgVDF.toolbar.button.normal
  + osgVDF.toolbar.button.hoover
  + osgVDF.toolbar.button.toggle
  + osgVDF.toolbar.button.pushed

osgUI namespace default styles:

* Klasa Toolbar:
  + osgUI.toolbar.tab.normal – styl taba w stanie normalnym
  + osgUI.toolbar.tab.active – styl taba aktualnie aktywnego
  + osgUI.toolbar.tab.disable – styl taba nieaktywnego
  + osgUI.toolbar.upperHalf – styl górnej belki z tabami
  + osgUI.toolbar.lowerHalf – styl dolnej belki z elementami związanymi z danym tabem
* Klasa Tooltip:
  + osgUI.tooltip.base – ogólny styl dla tooltipa, tło, rozmiar
  + osgUI.tooltip.text – styl dla tekstu tooltipa, nadpisuje ewentualne ustawienia stylu ogólnego
* Klasa MenuContext:
  + osgUI.contextmenu.menu – ogólny styl menu kontekstowego, tło, rozmiar
  + osgUI.contextmenu.submenuitem.normal – styl elementu menu kontekstowego odpowiedzialnego za submenu w stanie normalnym
  + osgUI.contextmenu.submenuitem.hoovered – styl elementu menu kontekstowego odpowiedzialnego za submenu po najechaniu na niego
  + osgUI.contextmenu. item.normal – styl elementu menu kontekstowego w stanie normalnym
  + osgUI.contextmenu. item.hoovered – styl elementu menu kontekstowego po najechaniu na niego
  + osgUI.contextmenu.checked.normal – styl zaznaczenia element menu kontesktowego w stanie normalnym
  + osgUI.contextmenu.checked.hoovered – styl zaznaczenia element menu kontesktowego po najechaniu na niego
  + osgUI.contextmenu.unchecked.normal – styl odznaczenia element menu kontesktowego w stanie normalnym
  + osgUI.contextmenu.unchecked.hoovered – styl odznaczenia element menu kontesktowego po najechaniu na niego

Dokumentacja Timeline

Podstawowe pojęcia:

|  |  |
| --- | --- |
| **Pojęcie** | **Opis** |
| Kanał | Niezależna porcja/agregat danych czasowych (kanałów) |
| Długość | Długość kanału wynikająca z jego faktycznych danych dostarczonych przez klienta i skali, bądź z agregacji dzieci (ich offsety i długości) |
| Offset | Przesunięcie początku kanału w czasie |
| Offset lokalny | Offset kanału względem początku rodzica – domyślnie 0 |
| Offset globalny | Offset kanału względem 0 (uwzględnia offsety w hierarchii kanałów) |
| Skala | Rozdzielczość danych czasowych kanału:   * 1 oryginalna rozdzielczość * < 1 mniejsza rozdzielczość * > 1 większa rozdzielczość |
| Skala lokalna | Rozdzielczość kanału względem rozdzielczości rodzica – domyślnie 1 |
| Skala globalna | Rozdzielczość kanału wynikająca z hierarchii kanałów |
| Maska | Pozwala wybrać zadany fragment kanału do aktualizacji czasu, mieści się w przedziale  [0; długość kanału] |
| Aktywność | Informacja czy kanał bierze udział w aktualizacji czasu (automatycznie maska dla kanałów dzieci) – domyślnie kanał aktywny |
| Czas | Aktualny czas kanału (globalny) – domyślnie globalny offset kanału (lokalnie to 0) |

Dodatkowe pojęcia:

|  |  |
| --- | --- |
| **Pojęcie** | **Opis** |
| Tag | Wydarzenie w danym momencie czasu danego kanału, punkt w czasie kanału |
| Zaznaczenie | Wydarzenie występujące w zadanych ramach czasowych kanału (od swojego początku do końca w ramach czasu tego kanału) |
| Split | Operacja rozdzielenie kanału na dwa niezależne kanały w zadanym punkcie czasu |
| Merge | Operacja łączenia kanałów w jeden kanał |

Podstawowe założenia względem struktury kanałów:

* Kanały tworzą strukturę podobną do systemu plików
* Timeline ma strukturę drzewiastą kanałów
* Jeden kanał może mieć jedno lub więcej dzieci, ma zawsze jednego rodzica (wyjątek root – brak rodzica)
* Na danym poziomie hierarchii kanały mają unikalne nazwy
* Offset lokalny kanału inicjowany jest wartością 0
* Offset globalny kanału inicjowany jest globalnym offsetem kanału rodzica
* Skala lokalna kanału inicjowana jest wartością 1
* Skala globalna kanału inicjowana jest wartością skali globalnej kanału rodzica
* Długość kanału inicjowana jest wartością długości kanału \* skala globalna
* Maska inicjowana jest wartościami 0 i długość kanału ( cały kanał aktywny przy aktualizacji czasu)
* Dodanie i usunięcie kanału powoduje aktualizację długości kanału rodzica (ewentualne wydłużenie lub skrócenie długości w zależności od operacji)

Podstawowe założenia dla operacji czasowych:

* Każdy kanał można edytować niezależnie, część właściwości propaguje się automatycznie w hierarchii, część działa w momencie aktualizacji czasu – szczegóły w dalszej części dokumentacji.
* Offset i skalę można zadawać w formie lokalnej i globalnej
* Czas automatycznie propaguje się w hierarchii kanałów
* Kanały liście aktualizują czas dla kanałów z danymi dostarczonymi przez klienta w formie: (globalny czas – globalny offset kanału) / globalna skala kanału
* Maska i aktywność kanału mogą blokować propagację czasu w dzieciach tego kanału – cała gałąź nie aktualizuje swojego czasu

Podstawowe założenia dla timera (globalnego zegara dla timeline):

* Odlicza czas od 0
* Krok inkrementacji czasu nas nie interesuje – steruje on tylko szybkością odtwarzania/aktualizacji czasu w timeline
* Maksymalna wartość również nie gra roli – timeline sam kontroluje czy globalny czas jest jeszcze w ramach czasu timeline

Szczegóły aktualizacji czasu w timeline mając na uwadze założenia dla timera:

* Root timeline przesuwa ustawiany czs o swój globalny (lokalny) offset – to jest rzeczywisty/ globalny czas dla timeline
* Globalny czas jest propagowany do wszystkich kanałów dzieci aż do kanałów liści
* Aktualizacja czasu propaguje się dalej jeśli kanał jest aktywny i zadany czas mieści się w masce

Szczegóły operacji na offsetach:

* Zmiana offsetu nie powoduje zmiany skali na żadnym poziomie hierarchii kanałów
* Zmiana offsetu wiąże się z aktualizacją długości kanału rodzica
* Offset lokalny przyjmuje wartości z przedziału [0; + ….]
* **Offset lokalny można zadawać ujemny!! (przykład w dalszej części dokumentu)**
* **Zmian offsetu lokalnego/globalnego może zmienić lokalny i globalny offset rodzica/ów oraz lokalne offsety jego/ich kanałów dzieci**
* Zmiana offsetu lokalnego/globalnego nie modyfikuje offsetów lokalnych dzieci, jedynie ich globalne offsety
* Każdą zmianę globalnego offsetu można przetłumaczyć na zmianę lokalnego offsetu

Szczegóły operacji na skalach:

* Zmiana skali lokalnej nie powoduje aktualizację offsetów związanych z kanałem (lokalny i globalny)
* Zmiana skali propaguje się w dół hierarchii aż do kanałów liści
* Zmiana skali lokalnej powoduje aktualizację długości kanałów w hierarchii
* **Zmiana skali lokalnej aktualizuje maski oraz offsety od aktualnego kanału w dół hierarchii ??**
* Zmiana skali lokalnej aktualizuje skalę globalną kanału:

skala globalna = skala globalna rodzica \* skala lokalna || skala globalna = skala lokalna (dla roota)

* Zmiana skali globalnej powoduje aktualizację długości kanałów w hierarchii
* Zmiana skali globalnej powoduje aktualizację skali lokalnej (analogicznie do zmiany skali lokalnej)

Wprowadzenie do przykładów:

Kolor czerwony – kanał rodzic, agregujący inne kanały

Kolor zielony – kanały dzieci

Kolor czarny – offset lokalny dziecka względem rodzica

Kolor pomarańczowy – dodawany offset

Kolor niebieski – różnica w czasie

Przykłady:

* Offset lokalny ujemny (analogicznie offset globalny mniejszy niż offset globalny rodzica):

przed:

po (krok 1):

po (krok 2): + **-**

rodzic wydłużył się o loklany

offset który ustawiamy, ale

jego koniec czasu oparł się

teraz na końcu innego

kanału dziecka

loklany offset zmianianego

kanału równa się 0 a nie

wartości ustawianej,

lokalne offsety innch dzieci

rodzica również uległy

zmianie ale ich bezwzględna

pozycja w czasie się nie zmianiła

(globalne offsety)

Zmiana propaguje się w hierarchi do góry, ponieważ nasz rodzic może teraz mieć lokalny offset ujemny względem swojego rodzica!!

Zmianiany kanał aktualizuje jedynie globalne offsety swoich dzieci aż do liści o wartość ustawianego offsetu.

Każdy globalny offset można przetłumaczyć na aktualizację loklanego offsetu.

* Zmiana skali lokalnej (dwukrotne wydłużenie czasu):

przed:

po:

Każdą zmianę globalnej skali można przetłumaczyć na zmianę skali lokalnej.

Zmiana skali powoduje aktualizację długości rodzica zmienianego kanału – może się on wydłużyć lub skrócić albo pozostać niezmieniony.