Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 10

4 grudnia 2019

Treść dzisiejszego wykładu

- dziedziczenie i polimorfizm - przypomnienie
- klasy abstrakcyjne
- dziedziczenie wielobazowe
- interfejsy w C++
- SOLID

dziedziczenie i polimorfizm - przypomnienie

Składnia

```
class /*struct*/ KlasaPochodna
    : /*typ dziedziczenia 1*/ KlasaBazowa1 /*, */
      /*typ dziedziczenia 2 KlasaBazowa1, ... */
{ /*definicja klasy*/ };
```

- Klasa może dziedziczyć tylko po typach już zdefiniowanych
- Klasa może dziedziczyć w sposób publiczny, prywatny bądź chroniony (`public`, `private`, `protected`)
- Słowa kluczowe `public`, `private`, `protected` mogą być poprzedzone słowem kluczowym `virtual`
- Dla wyróżnika `class` domyślne jest dziedziczenie prywatne
- Dla wyróżnika `struct` domyślne jest dziedziczenie publiczne
- Nie można **bezpośrednio** dziedziczyć dwukrotnie po tym samym typie.

dziedziczenie i polimorfizm - przypomnienie

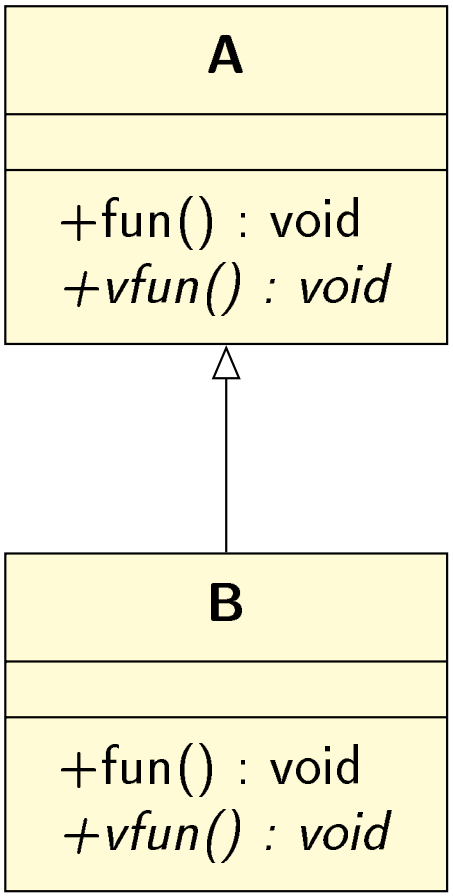
```
class A {
void fun() {}
virtual void vfun() {}
};
```

```
class B : public A {
void fun() {}
void vfun() override {}
};
```

przysłanianie nazw i metody wirtualne

- klasa pochodna ma dostęp do wszystkich publicznych oraz chronionych metod i pól klasy bazowej.
- o ile klasa nie dziedziczy w sposób prywatny (lub chroniony) to wszystkie metody i pola klasy bazowej są dostępne dla użytkowników klasy pochodnej
- klasa pochodna może redefiniować (przesłaniać) metody i pola klasy bazowej
- mechanizm funkcji wirtualnych pozwala na odwoływanie się do metody klasy w zależności od jej rzeczywistego typu
- mechanizm funkcji wirtualnych działa również dla metody chronionych i prywatnych [sic!]

dziedziczenie i polimorfizm - przypomnienie



- destruktory powinny być wirtualne; w ten sposób dysponując referencją bądź wskaźnikiem do klasy bazowej poprawnie usuwany jest obiekt klasy pochodnej
- słowo kluczowe `override` jeżeli reimplementujemy metodę wirtualną klasy bazowej
- nie istnieje symetryczne pojęcie konstruktora wirtualnego; w momencie konstrukcji obiektu znany jest jego dokładny typ.

dziedziczenie i polimorfizm - przypomnienie

```
class Vehicle{
public:
virtual
void run(const TimeInterval& s);
virtual ~Vehicle(void);
};

class Car : public Vehicle{
public:
/*...*/
void run(const TimeInterval& s)
    override;
};

class Truck : public Vehicle{
public:
/*...*/
void run(const TimeInterval& s)
    override;
};
```

```
int main(void)
{
std::vector <
    std::shared_ptr<
        Vehicle>> vehicles;

vehicles.emplace_back(
    new Car);
vehicles.emplace_back(
    new Car);
vehicles.emplace_back(
    new Truck);

/* ... */

for (auto& v : vehicles)
    v->run({ 0.1 });
}
```

dziedziczenie i polimorfizm - przypomnienie

W rozważanym przykładzie:

- metoda `run` dla każdego typu pojazdu odpowiedzialna jest za jego przemieszczenie.
- w zależności od typu pojazdu, jego ruch może być zdefiniowany w różnych sposób.
- mechanizm funkcji wirtualnych (polimorfizm dynamiczny) wykorzystywany jest do wywoływania właściwej metody `run`.

W jaki sposób powinna wyglądać implementacja metody `run` dla klasy `Vehicle` a jak dla klasy `Car` i `Truck` ?

dziedziczenie i polimorfizm - przypomnienie

W rozważanym przykładzie:

- metoda `run` dla każdego typu pojazdu odpowiedzialna jest za jego przemieszczenie.
- w zależności od typu pojazdu, jego ruch może być zdefiniowany w różnych sposób.
- mechanizm funkcji wirtualnych (polimorfizm dynamiczny) wykorzystywany jest do wywoływania właściwej metody `run`.

W jaki sposób powinna wyglądać implementacja metody `run` dla klasy `Vehicle` a jak dla klasy `Car` i `Truck` ?

Nie zawsze definicja metody w klasie bazowej ma sens.

klasy abstrakcyjne

Język C++ umożliwia zdefiniowane metody nieposiadającej własnej implementacji. Metoda niestatyczna zdefiniowana w następujący sposób

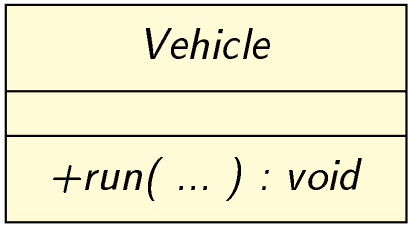
```
virtual TypWyniku nazwaMetody( Parametr1 p1 /*, ... */) = 0;
```

jest **metodą czysto wirtualną**.

- Klasa posiadająca co najmniej jedną metodę czysto wirtualną nosi nazwę **klasy abstrakcyjne**.
- Klasy dziedziczące po klasie abstrakcyjne, które nie implementują metod czysto wirtualnych, również pozostają abstrakcyjne.
- Klasy abstrakcyjne nie mogą być podstawą do tworzenia obiektów; mogą być jedynie klasami bazowymi.
- Metody czysto wirtualne **nie posiadają** implementacji.
- Metodą czysto wirtualną może być destruktor. Czysto wirtualny destruktor **musi** mieć własną implementację.

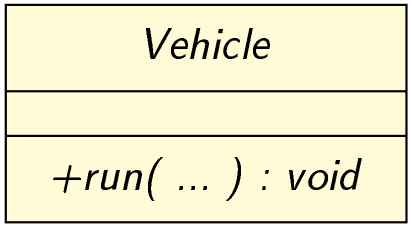
klasy abstrakcyjne, interfejsy

```
class Vehicle
{
public:
    virtual void run(const Time& s) = 0;
    virtual ~Vehicle(void) = 0 {};
};
```

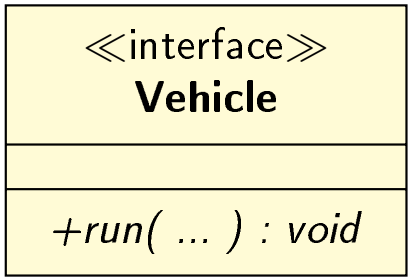


klasy abstrakcyjne, interfejsy

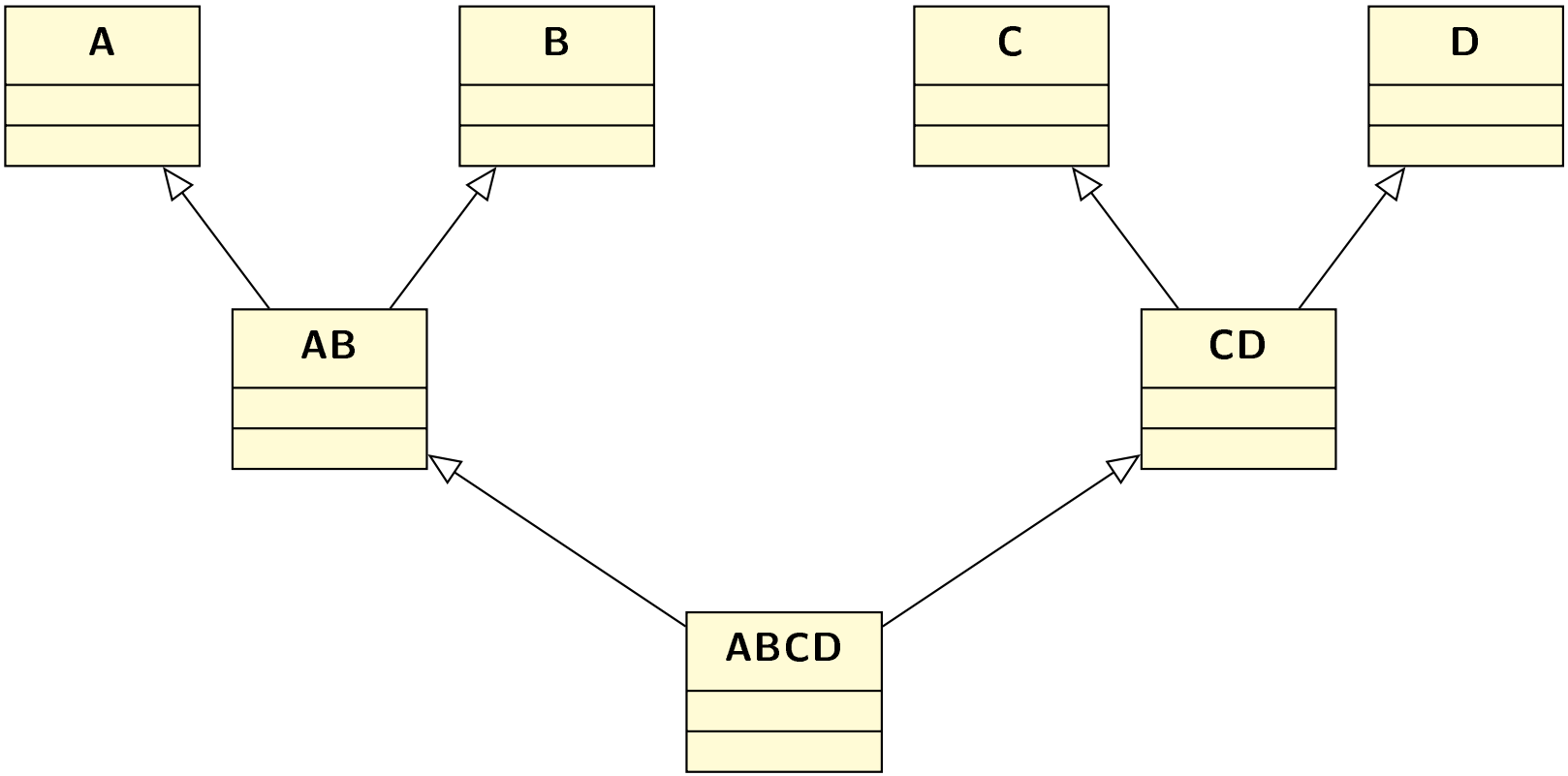
```
class Vehicle
{
public:
    virtual void run(const Time& s) = 0;
    virtual ~Vehicle(void) = 0 {};
};
```



- Język C++ nie definiuje pojęcia **interfejsu**. Tę rolę mogą pełnić klasy abstrakcyjne, które nie posiadają pół, a wszystkie metody mają zadeklarowane jako czysto wirtualne.
- Interfejsy informują jakie operacje muszą być implementowane w danym typie.
- Jeden typ pochodny może implementować wiele interfejsów, poprzez dziedziczenie wielobazowe.

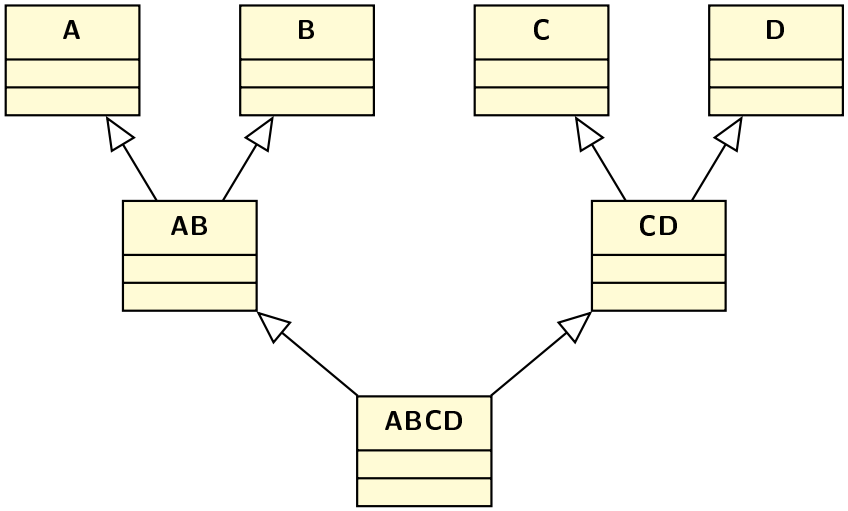


dziedziczenie wielobazowe



dziedziczenie wielobazowe

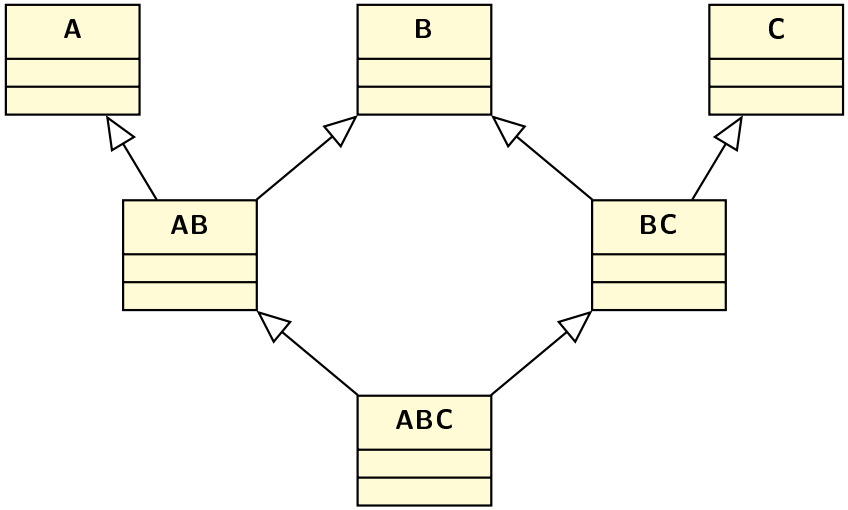
```
struct A {
    void funA() {}
};

struct B {
    void funB() {}
};

struct C {
    void funC() {}
};

struct D {
    void funD() {}
};

struct AB : A, B {};
struct CD : C, D {};
struct ABCD : AB, CD {};
```




```
int main(void)
{
    ABCD().funA();
    /* ... */
    ABCD().funD();
    return 0;
}
```

dziedziczenie wielobazowe

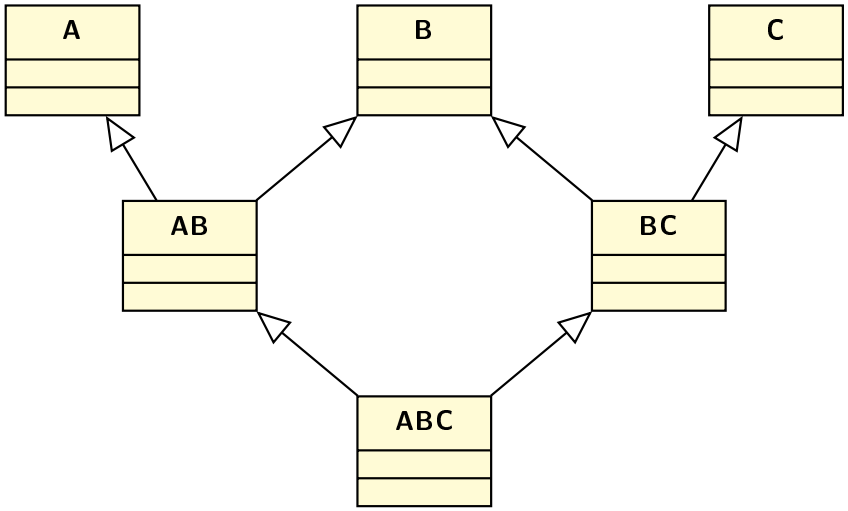
```
struct A {
    void funA() {}
};

struct B {
    void funB() {}
};

struct C {
    void funC() {}
};

struct AB : A, B {};
struct BC : B, C {};
struct ABC : AB, BC {};
```




```
int main(void)
{
    ABC().funA();
    // ABC().funB();
    // ambiguous access of 'funB'
    ABC().funC();
    return 0;
}
```

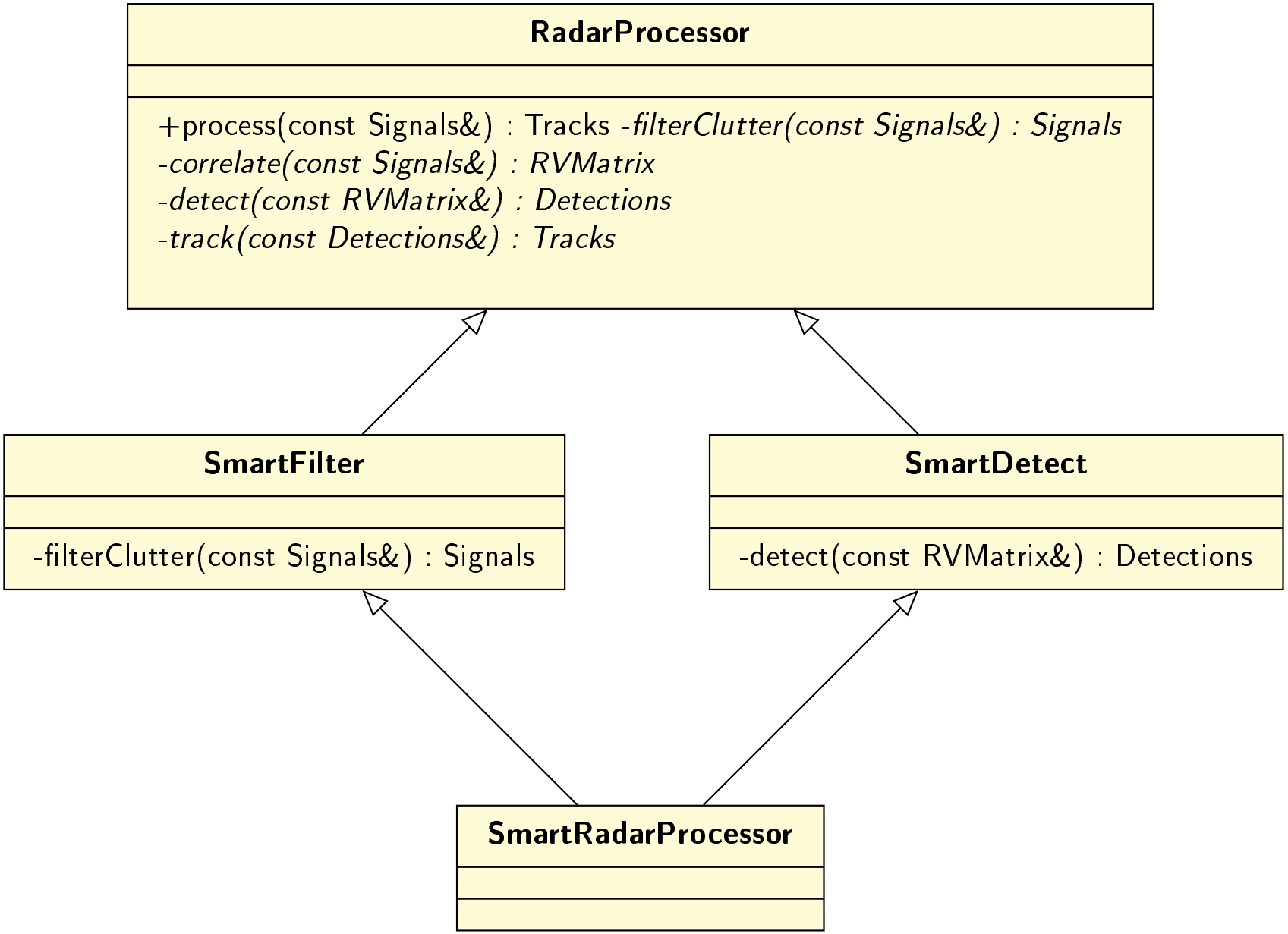
dziedziczenie wielobazowe

```
struct A {
    void funA() {}
};

struct B {
    void funB() {}
};

struct C {
    void funC() {}
};

struct AB : A, virtual B {};
struct BC : virtual B, C {};
struct ABC final : AB, BC {};
// struct D : ABC {};
// 'D': cannot inherit from
// 'ABC' as it has been
// declared as 'final'
```




```
int main(void)
{
    ABC().funA();
    ABC().funB();
    ABC().funC();
    return 0;
}
```

dziedziczenie wielobazowe - przykład



Przykład

```
class RadarProcessor
{
public:
    Tracks process(const Signals& s) const
    {
        return track(detect(correlate(filterClutter(s))));
    }

private:
    virtual Signals filterClutter(const Signals&) const;
    virtual RVMatrix correlate(const Signals&) const;
    virtual Detections detect(const RVMatrix&) const;
    virtual Tracks track(const Detections&) const;
};
```

```
class SmartFilter : virtual public RadarProcessor
{
    Signals filterClutter(const Signals&) const override;
};
```

```
class SmartDetect : virtual public RadarProcessor
{
    Detections detect(const RVMatrix&) const override;
};
```

Przykład

```
class SmartestRadarProcessor
    : public SmartFilter, public SmartDetect
{
};
```

```
struct RadarProcessorFactory
{
    std::shared_ptr<RadarProcessor> get() const;
}
```

```
int main(void)
{
    auto radarProcessor = RadarProcessorFactory().get();
    radarProcessor->process(Signals());
    return 0;
}
```

SOLID

- **S**ingle responsibility principle
  - Zasada jednej odpowiedzialności
- **O**pen/closed principle
  - Zasada otwarte-zamknięte
- **L**iskov substitution principle
  - Zasada podstawienia Liskowej
- **I**nterface segregation principle
  - Zasada segregacji interfejsów
- **D**ependency inversion principle
  - Zasada odwrócenia zależności

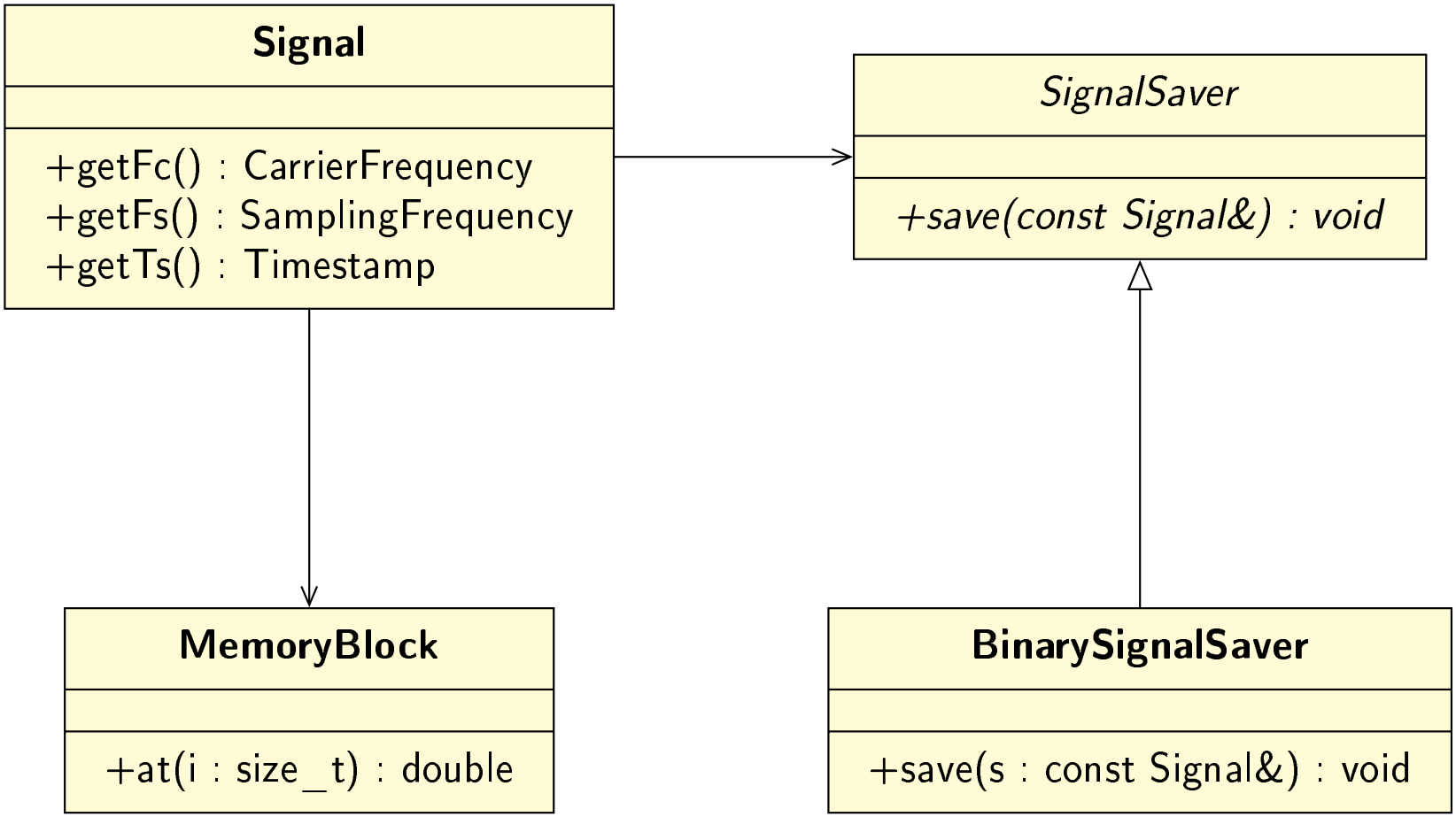
Zasada jednej odpowiedzialności

Powinien istnieć tylko jeden powód do modyfikacji klasy.
(ang. *A class should have only one reason to change.*)

- Odpowiedzialność rozumiana jest jako rodzina funkcji (metod), która służy **jednemu konkretnemu** aktorowi.
- Aktor odpowiedzialności jest jedynym źródłem zmiany tej odpowiedzialności.
- Odpowiedzialność w kontekście zasady jednej odpowiedzialności definiowana jest jako **powód do zmiany**.

Zbierz rzeczy, które zmieniają się z tych samych powodów. Oddziel te rzeczy, które zmieniają się z różnych powodów.

Zasada jednej odpowiedzialności



Zasada jednej odpowiedzialności

- Obiekty klasy Signal agregują blok danych reprezentowany przez klasę MemoryBlock oraz podstawowe informacje o parametrach sygnału: częstotliwość nośną, częstotliwość próbkowania, czas akwizycji.
- Przez większość czasu życia obiektu typu Signal, jego dane są odczytywane / modyfikowany przez moduł odpowiedzialny za przetwarzanie danych.
- Zapis danych (np. na dysk) następuje na żądanie zupełnie innego modułu niż moduł przetwarzający dane.
- Klasa abstrakcyjna SignalSaver dostarcza interfejsu jaki muszą implementować konkretne klasy zapisujące dane.
- Zmiana (lub dodanie) sposobu zapisu danych nie pociąga za sobą konieczności modyfikowania klasy Signal.

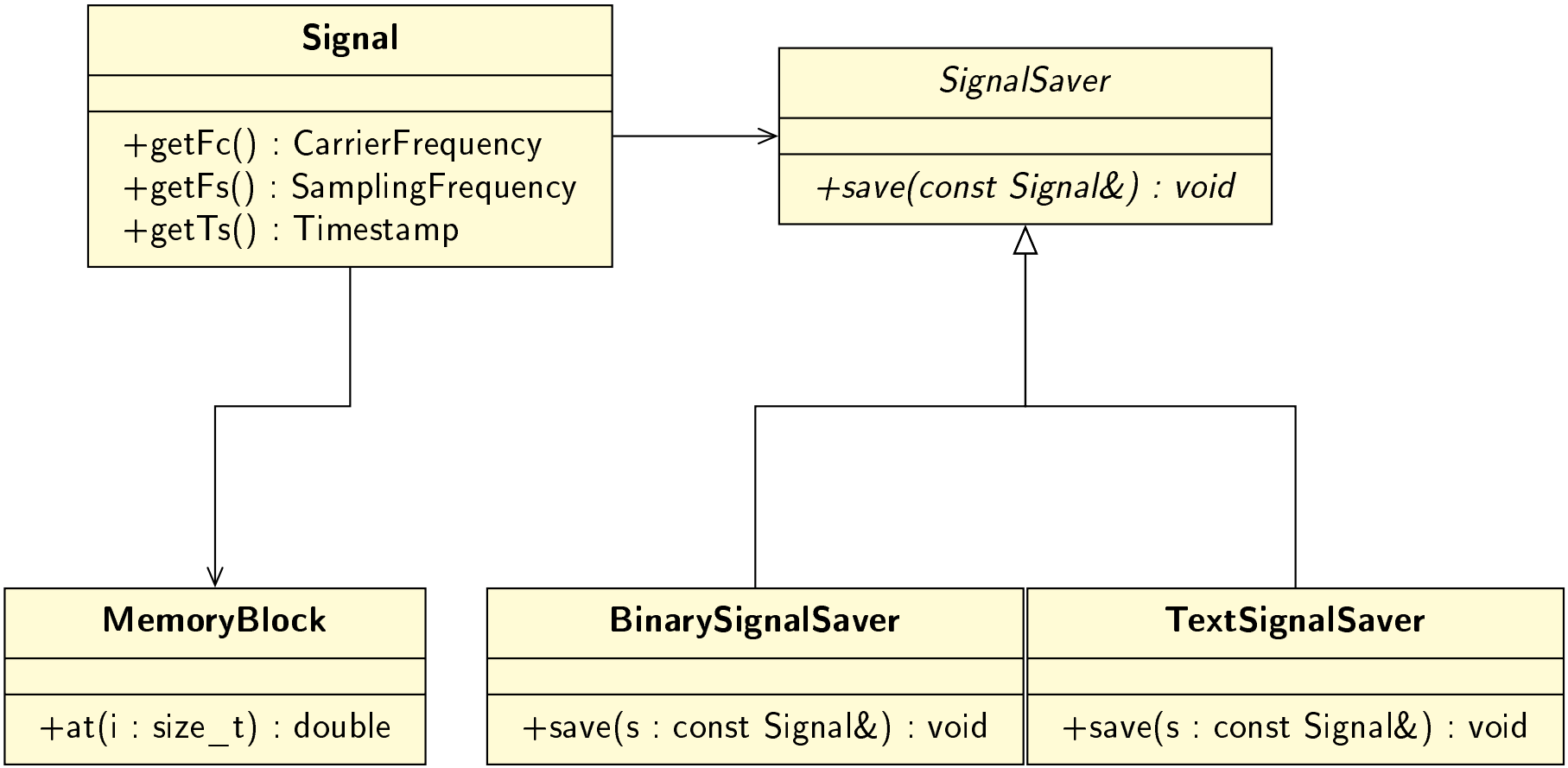
Zasada otwarte-zamknięte

Encje oprogramowania(klasy, moduły, funkcje itp.) powinny być otwarte na rozbudowę, ale zamknięte dla modyfikacji.
(ang. *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*)

- Klasa uważana jest za zamkniętą gdy może zostać skompilowana i dostarczana do użytkownika w postaci binarnej w ramach biblioteki.
- Klasa pozostaje otwarta jeżeli nowe klasy mogą wykorzystać ją jako klasę bazową, dodając nowe właściwości.

Stosuj abstrakcyjne klasy bazowe.

Zasada otwarte-zamknięte



Zasada otwarte-zamknięte

- Zarówno klasy BinarySignalSaver jak i TextSignalSaver implementują interfejs dany przez SignalSaver.
- Klasa Signal nic nie wie o konkretnych metodach zapisujących.
- Istnieje możliwość dalszego **rozszerzania** funkcjonalności systemu o nowe mechanizmy zapisu danych (np. w postaci plików XML).

Zasada podstawienia Liskowej

Typy pochodne muszą móc być podstawione za ich typy bazowe.
(ang. *"Subtypes must be substitutable for their base types."*)

- Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów

Obiekty klas potomnych zachowują się jak obiekty klas bazowych.

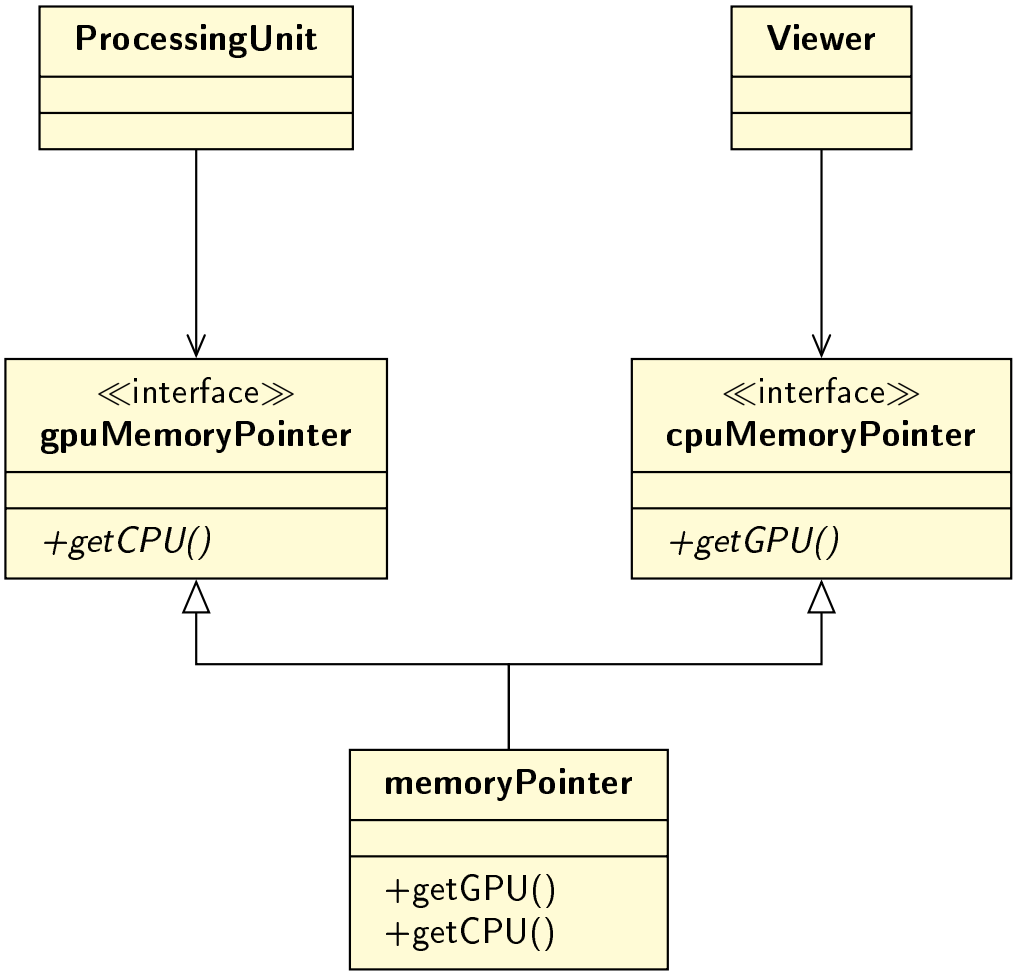
Zasada segregacji interfejsów

Klasy klientów nie powinny być zmuszone do zależności od metod, których nie używają.
(ang. *"No client should be forced to depend on methods it does not use."*)

- Klasy, które charakteryzują się grubymi" interfejsami, to klasy których interfejsy nie są spójne. Wydzielone grupy metod obsługują rożne zbiory klientów.
- Interfejsy powinny być małe, żeby później klasy nie implementowały metod, których nie potrzebują.

Unikaj sytuacji, w której moduł klienta musi wiedzieć więcej niż potrzebuje do wykonania swojej zadania (odpowiedzialności).

Zasada segregacji interfejsów



Zasada segregacji interfejsów

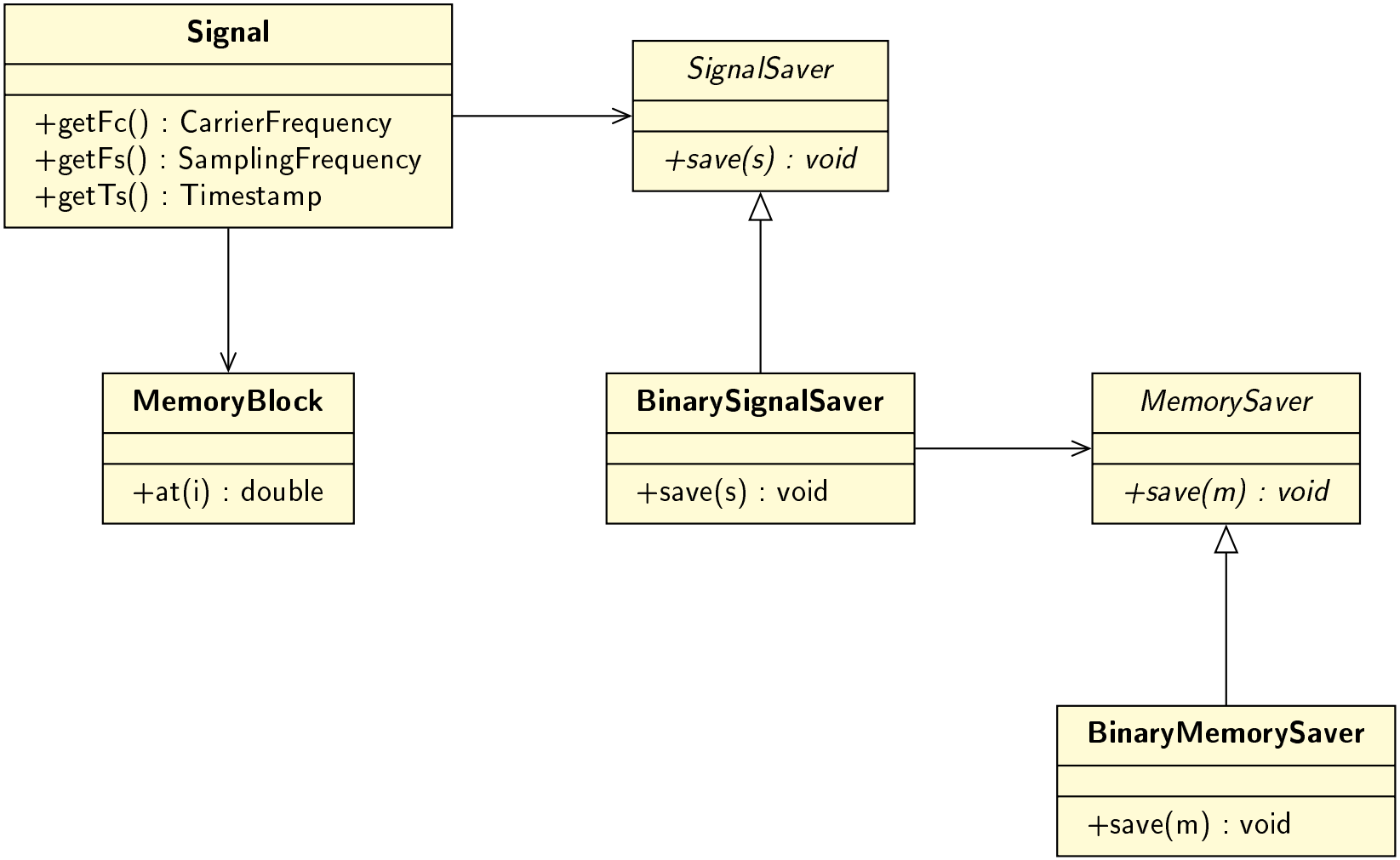
- Klasa Viewer nie musi nic wiedzieć ani o metodach przetwarzania danych ani o specjalnym rodzaju pamięci.
- Moduł ProcessingUnit wykorzystuje jedynie pamięć operacyjną karty graficznej. Nie musi znać metod zarządzania pamięcią RAM procesora.
- Istnieje możliwość implementacji typu stricte wskazującego na pamięć w CPU lub GPU.

Zasada odwrócenia zależności

Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. I jedne, i drugie powinny zależeć od abstrakcji.
(ang. *"High-level modules should not depend on low-level modules. Both should depend on abstractions."*)
Abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji
(ang. *"Abstractions should not depend upon details. Details should depend upon abstractions."*)

Projektuj jasno określone warstwy w architekturze systemu. Używaj interfejsów i klas abstrakcyjnych wszędzie tam, gdzie wydaje się, że może to być użyteczne w przyszłości.

Zasada odwrócenia zależności



Zasada odwrócenia zależności

- Ponieważ klasa reprezentująca blok pamięci może być wykorzystywana wielokrotnie, nie ma sensu wiązać mechanizmu jej zapisu z zapisem klasy samych sygnałów.
- W ten sposób mechanizm zapisu może być rozwijany zgodnie z zasadą otwarte - zamknięte

Dziękuję za uwagę