Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 11

3 stycznia 2019

Treść dzisiejszego wykładu

- rzutowanie w stylu C++
- pimpl
- omówienie sprawdzianu numer 2

Operatory rzutowania

Składnia

```
Typ_cast <nowy_typ> (wyrazenie)
```

typy rzutowania

- `const_cast`
- `static_cast`
- `reinterpret_cast`
- `dynamic_cast`

Operatory rzutowania

const_cast

Operator służący do usuwania modyfikatorów `const` i `volatile`.
Konwersja realizowana na etapie kompilacji - bezpieczna.

```
int i = 0;
const int& cri = i;
int& ri = i;
ri++; // OK
// cri++; // blad
const_cast<int&>(cri)++;

const int ci = 2;
// ci++; // blad
const_cast<int&>(ci)++;
```

Operatory rzutowania

static_cast

Operator służąc do dokonywania konwersji standardowych lub zdefiniowanych przez użytkownika pomiędzy różnymi typami danych. Konwersja realizowana na etapie kompilacji - bezpieczna.

```
class A {};
class B : public A {};

int main(void){
    auto sp_b = std::make_shared<B>();
    auto sp_a = static_cast<std::shared_ptr<A>>(sp_b);

    double d = 10;
    int i1 = d; // warning C4244 : 'initializing' : conversion
    // from 'double' to 'int', possible loss of data
    int i2 = static_cast<int>(d);

    return 0;
}
```

Operatory rzutowania

reinterpret_cast

Operator w postaci dyrektywy dla kompilatora by interpretować ciąg bitów wyrażenia jak gdyby reprezentowały dane typu nowy_typ. UWAGA - rzutowanie mogące być przyczyną wielu błędów trudnych do wykrycia.

```
struct Complex{ double x, y; };

template<typename T>
struct Pair { T pair[2]; };

int main(void){
    Complex c1{ 1,2 };
    auto d1 = reinterpret_cast<Pair<double>&>(c1);
// d1 = {pair=0x00c0fbf0 {1.0000000000000000, 2.0000000000000000}
    auto f1 = reinterpret_cast<Pair<float>&>(c1);
// f1 = {pair=0x00c0fbe0 {0.000000000, 1.87500000} }
    return 0;
}
```

Operatory rzutowania

dynamic_cast

Operator do konwersji wskaźników i referencji do obiektów polimorficznych powiązanych relacją dziedziczenia. Konwersja może zachodzić "w górę" , "w dół" i "na boki" hierarchii. Konwersja realizowana w czasie wykonywania programu.

Operatory rzutowania

```
struct A {
    void fun(void) { std::cout << "A" << std::endl; }
    virtual ~A(void) {}
};

struct B : public A {
    void fun(void) { std::cout << "B" << std::endl; }
};

int main(void){
    B* p_b = new B();
    A* p_a = p_b;

    p_a->fun();
    p_b->fun();

    auto p_b2 = dynamic_cast<B*>(p_a);
    p_b2->fun();

    delete p_b;
    return 0;
}
```

Operatory rzutowania

```
struct A {
    void fun(void) { std::cout << "A" << std::endl; }
    virtual ~A(void) {}
};

struct B : public A {
    void fun(void) { std::cout << "B" << std::endl; }
};

int main(void){
    B* p_b = new B();
    A* p_a = new A();

    p_a->fun();
    p_b->fun();

    auto p_b2 = dynamic_cast<B*>(p_a);
    p_b2->fun(); // blad p_b2 == nullptr

    delete p_b; delete p_a;
    return 0;
}
```

Operatory rzutowania

```
int main(void)
{
    auto sp_b = std::make_shared<B>();
    auto sp_a = std::static_pointer_cast<A>(sp_b);

    sp_a->fun();
    std::dynamic_pointer_cast<B>(sp_a)->fun();

    return 0;
}
```

Operatory rzutowania

```
int main(void)
{
    B b;
    A& ra = b;
    A a;

    b.fun();
    ra.fun();

    dynamic_cast<B&>(ra).fun();
    dynamic_cast<B&>(a).fun(); // std::bad_cast

    return 0;
}
```

Przykład

```
class CircularMotion
{
    T t_;
    R r_;
    V v_;
    A a_;
    radius radius_;
    angle alpha_;

public:
    CircularMotion(const R & r, const V & v, const A &,
          const T & t, const radius & radius, const angle & alpha)
        : t_(t), r_(r), v_(v), a_(a),
          radius_(radius), alpha_(alpha) {}

    std::tuple<R, V, A> move(const T & t);
    R position(void) const;
};
```

Technika **PImpl**

Wady publicznych implementacji klas.

- Opublikowane prywatne atrybuty klasy.
- Zmiana implementacji klasy pociąga za sobą konieczność kompilacji wszystkich zależnych od niej komponentów ...
- ... oraz wszystkich komponentów zależnych od tych komponentów, itd.
- Poszczególne moduły zależą od prywatnych implementacji klas.

Technika **PImpl**

PImpl jest to technika, w której wszystkie dane składowe klasy są zastępowane wskaźnikiem do klasy implementacji. Implementację klasy stanowią jedynie jej publiczne (niekiedy chronione) metody oraz wskaźnik do struktury implementacji. Implementacja wszystkich metod publicznych (w tym konstruktorów oraz destruktora) polega na odwołaniu do obiektu (poprzez jego adres) prywatnej implementacji.

CircularMotion.hpp

```
class CircularMotion
{
    struct Impl;
    std::shared_ptr<Impl> impl_;
public:
    CircularMotion(const R & r, const V & v, const A &,
        const T & t, const radius & radius, const angle & alpha);

    std::tuple<R, V, A> move(const T & t);
    R position(void) const;
};
```

CircularMotion.cpp

```
struct CircularMotion::Impl
{
    T t_;
    R r_;
    V v_;
    A a_;
    radius radius_;
    angle alpha_;

    Impl(const R & r, const V & v, const A &, const T & t,
         const radius & radius, const angle & alpha);

    std::tuple<R, V, A> move(const T & t);
};
```

CircularMotion.cpp

```
CircularMotion::CircularMotion(const R & r, const V & v,
    const A & a, const T & t,
    const radius & radius, const angle & alpha)
    : pimpl_(std::make_shared<CircularMotion::Impl>(
        r, v, a, t, radius, alpha))
{
}

std::tuple<R, V, A> CircularMotion::move(const T & t)
{
    return pimpl_->move(t);
}

R CircularMotion::position(void) const
{
    return pimpl_->r_;
}
```

Technika **PImpl**

Moduły zależące od klasy `CircularMotion` nie zależą od jej implementacji, a jedynie od publicznego interfejsu.
Zmiana szczegółów implementacji pociąga konieczność kompilacji jedynie tego modułu.
Istnieje możliwość opublikowania interfejsu i dostarczenia klasy w postaci biblioteki dynamicznej bądź statycznej. Za każdym razem, gdy zmianie ulegnie implementacja prywatna danej klasy możliwe jest udostępnianie jedynie pliku wykonywalnego.

Przykładowe zadanie

Zaprojektuj prosty symulator ruchu lotniczego. Symulator musi spełniać następujące wymagania:

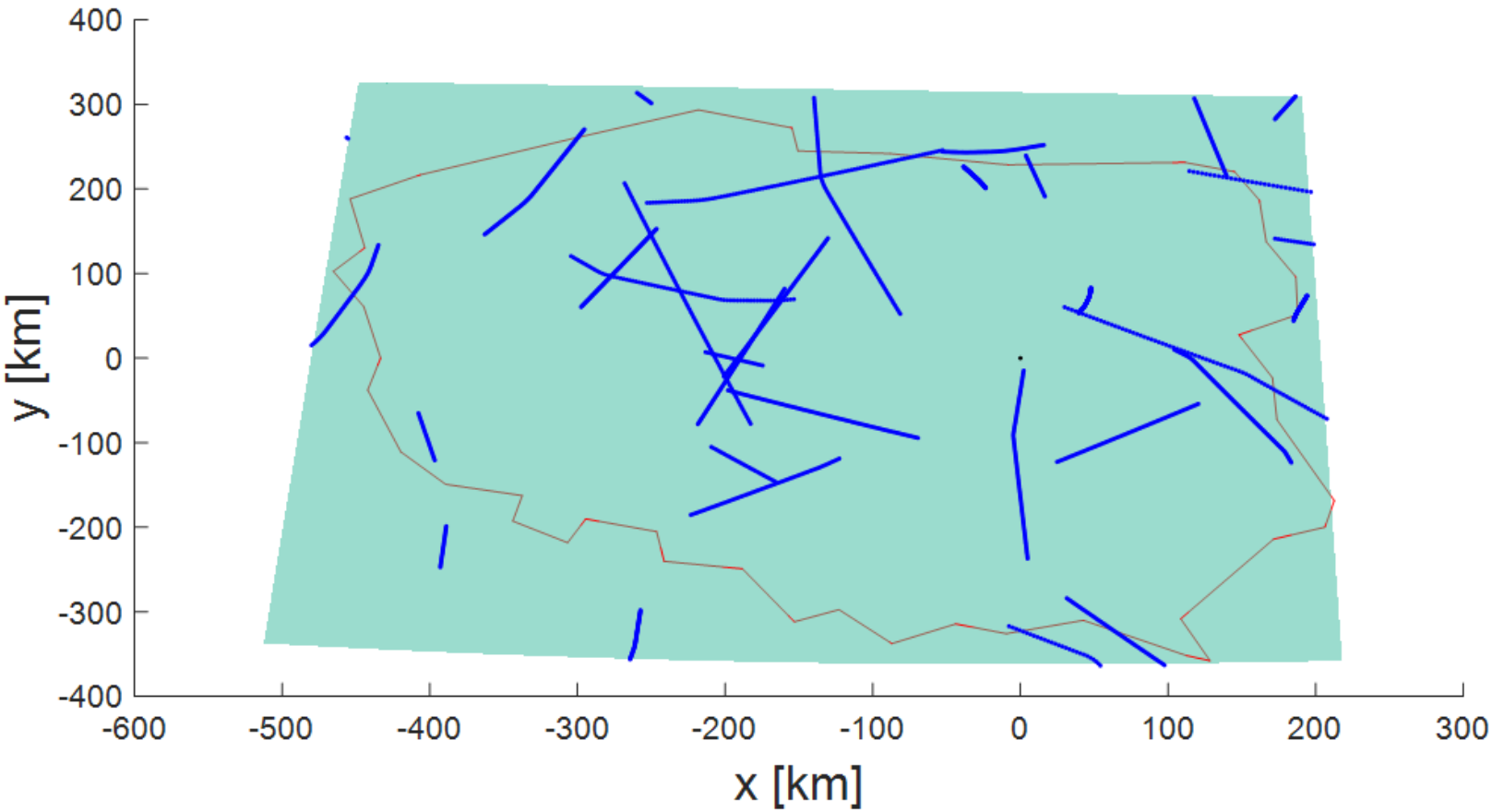
- Podstawowym zadaniem symulatora jest dostarczanie informacji o położeniu symulowanych obiektów w określonych odstępach czasowych.
- Zadany jest obszar symulacji, nowe obiekty pojawiają się na brzegu tego obszaru, obiekty, które wyleciały poza ten obszar są usuwane.
- Niektóre obiekty powinny mieć skomplikowaną trajektorię ruchu (np. wykonanie manewru skrętu)
- W chwili początkowej w obserwowanym obszarze znajdują się już pewne obiekty.

Przykładowe zadanie

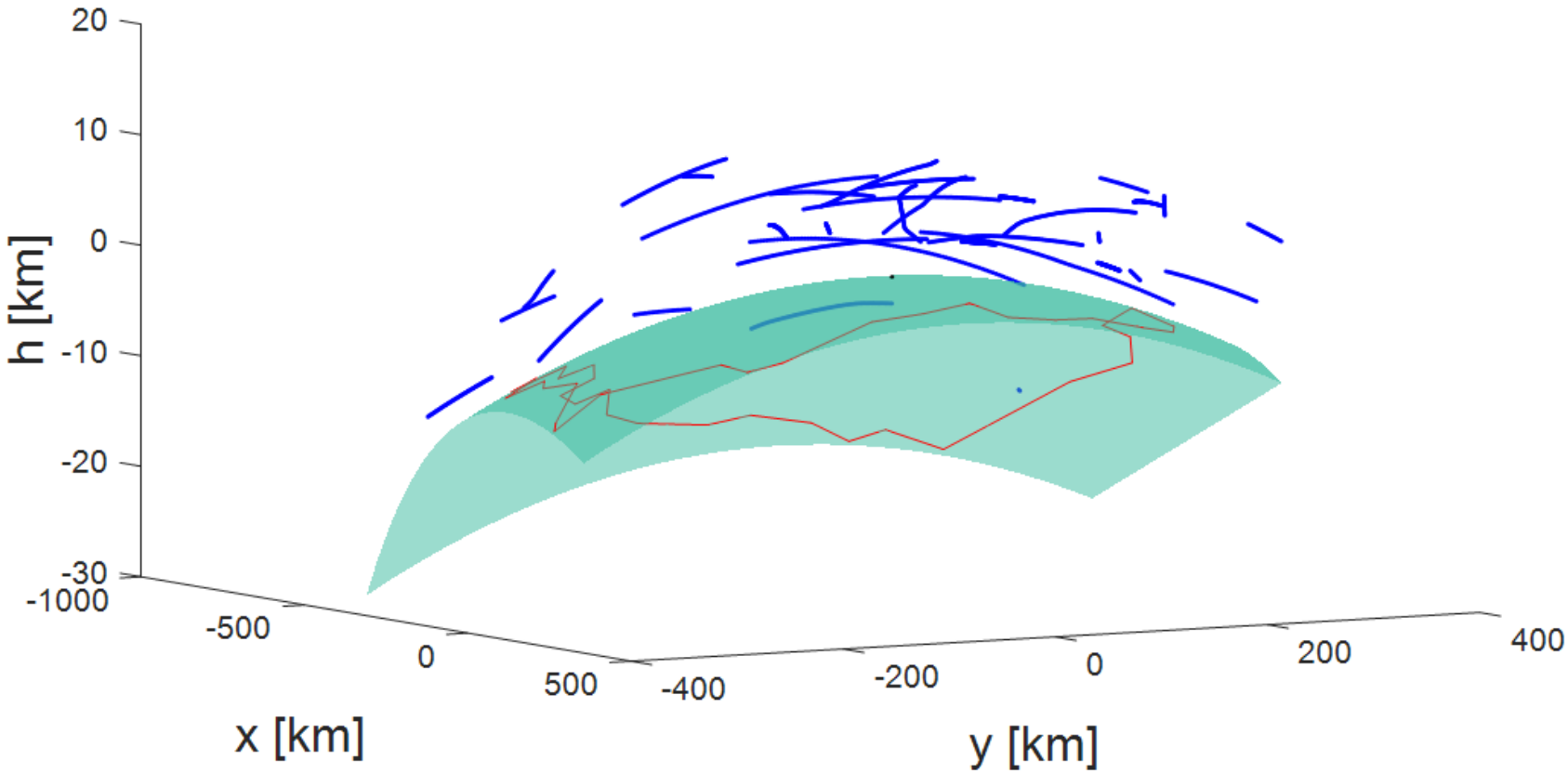
W projekcie symulatora uwzględnij:

- Mechanizm reagowania na sytuacje wyjątkowe (*jakie sytuacje wyjątkowe mogą się pojawić w takim symulatorze ?*).
- Możliwość rozszerzania funkcjonalności o dodatkowe trajektorie (przyspieszanie, zwalnianie).
- Dodawanie nowych typów obiektów i charakterystycznych dla nich typów ruchu (np. helikopter może znajdować się w zawisie).

Przykładowe zadanie



Przykładowe zadanie



Przykładowe zadanie

simulator.hpp

```
class Simulator
{
    TargetFactory targetFactory_;
    Area area_;
    std::list<std::shared_ptr<Target>> targets_;

public:
    Simulator(const Area& area);
    std::vector<R> simulate(const T& delta_t);
};
```

Przykładowe zadanie

target_factory.hpp

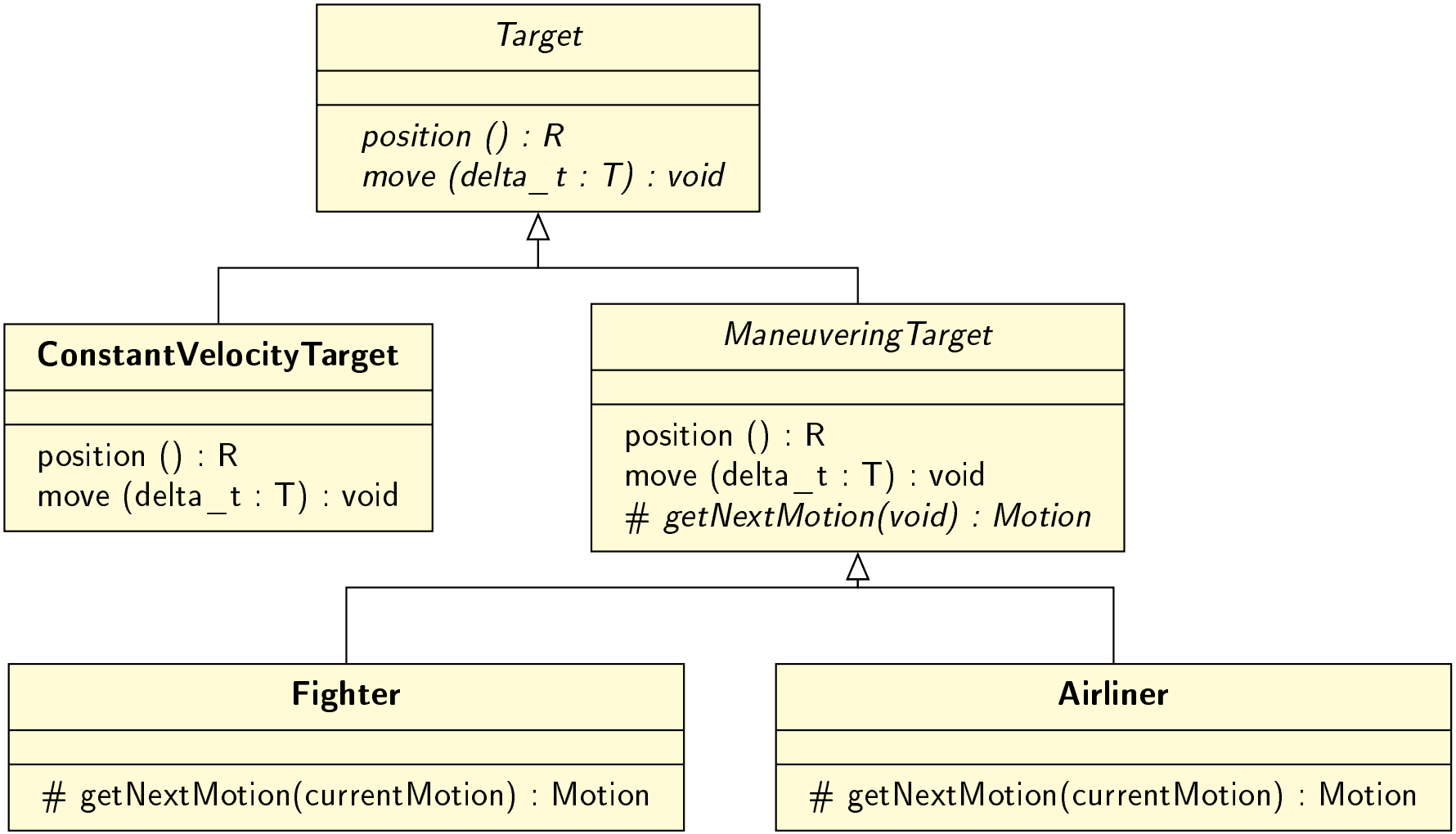
```
class Area
{
    double x, y;
};

class TargetFactory
{
public:
    TargetFactory();

    std::list<std::shared_ptr<Target>>
        getNewTargets(const Area& area);
};
```

Przykładowe zadanie

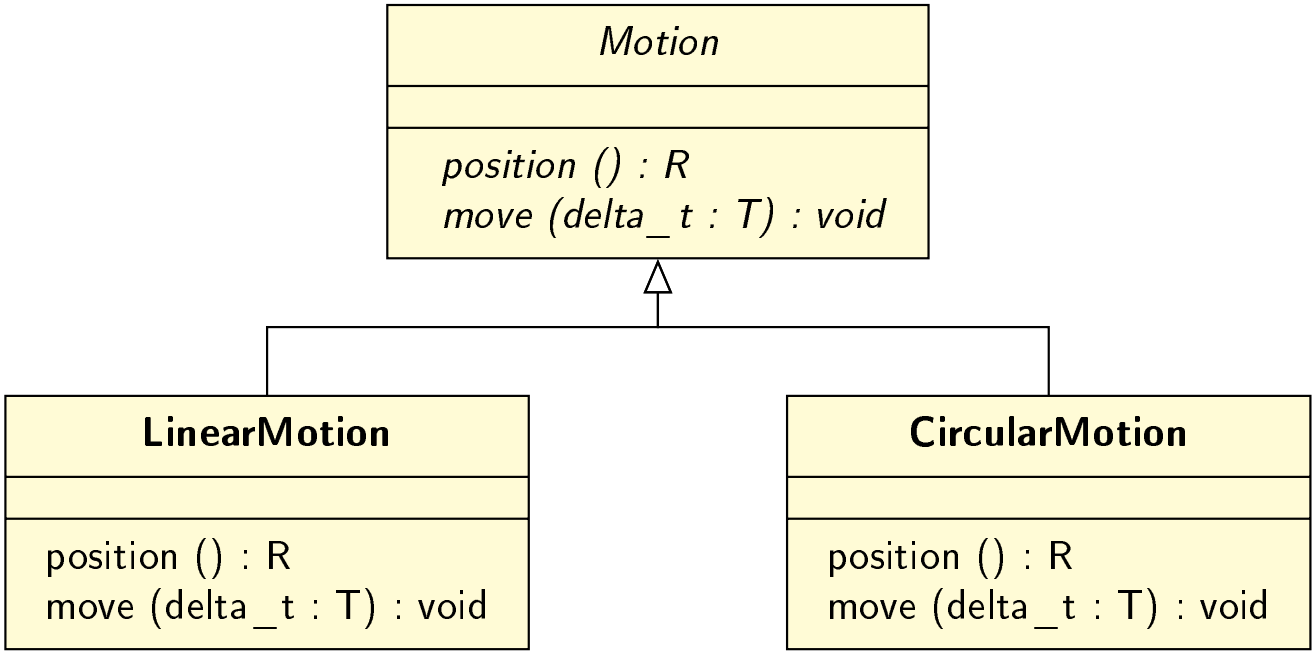
simulator.cpp

```
std::vector<R> Simulator::simulate(const T & delta_t)
{
    targets_.merge(targetFactory_.getNewTargets(area_));
    removeDistantTargets();

    std::vector<R> result;
    result.reserve(targets_.size());

    for (auto& sp_target : targets_)
    {
        sp_target->move(delta_t);
        result.push_back(sp_target->position());
    }

    return result;
}
```

Przykładowe zadanie



Przykładowe zadanie



Dziękuję za uwagę