Seminarium II – Warstwa usług i pełny backend UAR

Mirosław Franczak
Informatyka przemysłowa
28.11.2025

Cel i zakres etapu II

Cel etapu:

- dokończenie implementacji backendu UAR (model ARX + regulator PID + generator sygnałów + symulacja + serwis)
- przygotowanie warstwy usług (UARService)
- wykonanie testów jednostkowych i integracyjnych
- przedstawienie zmian w architekturze od seminarium I

Zakres:

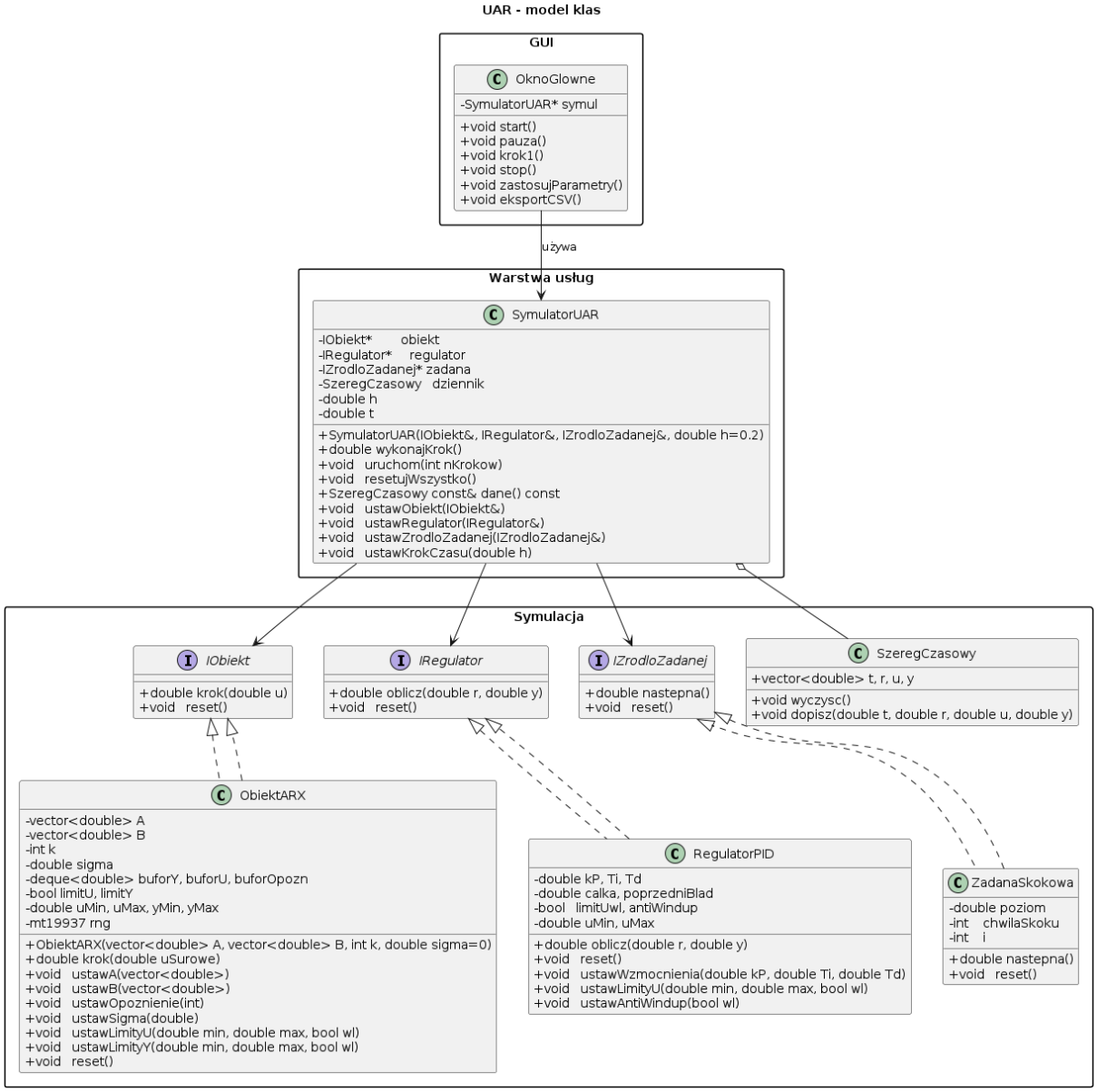
- finalny kod ARX, PID, Generatora
- nowa klasa Simulation
- nowa warstwa usług – UARService
- finalny diagram UML klas
- testy i ich wyniki

Architektura systemu przed zmianami

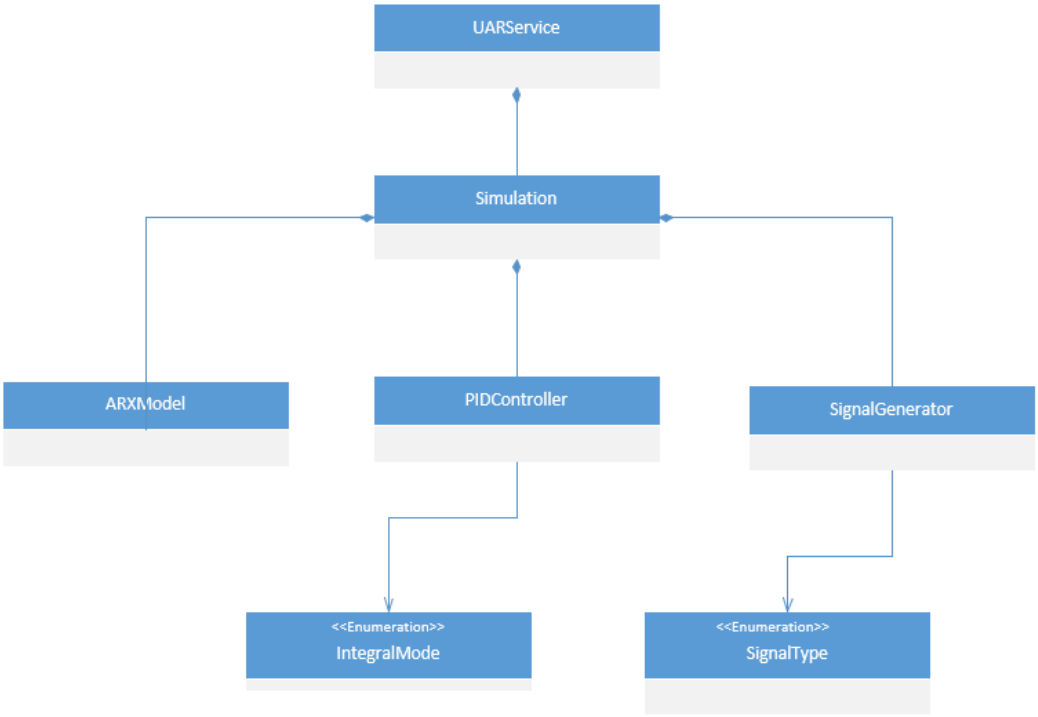
W seminarium I zrealizowano:

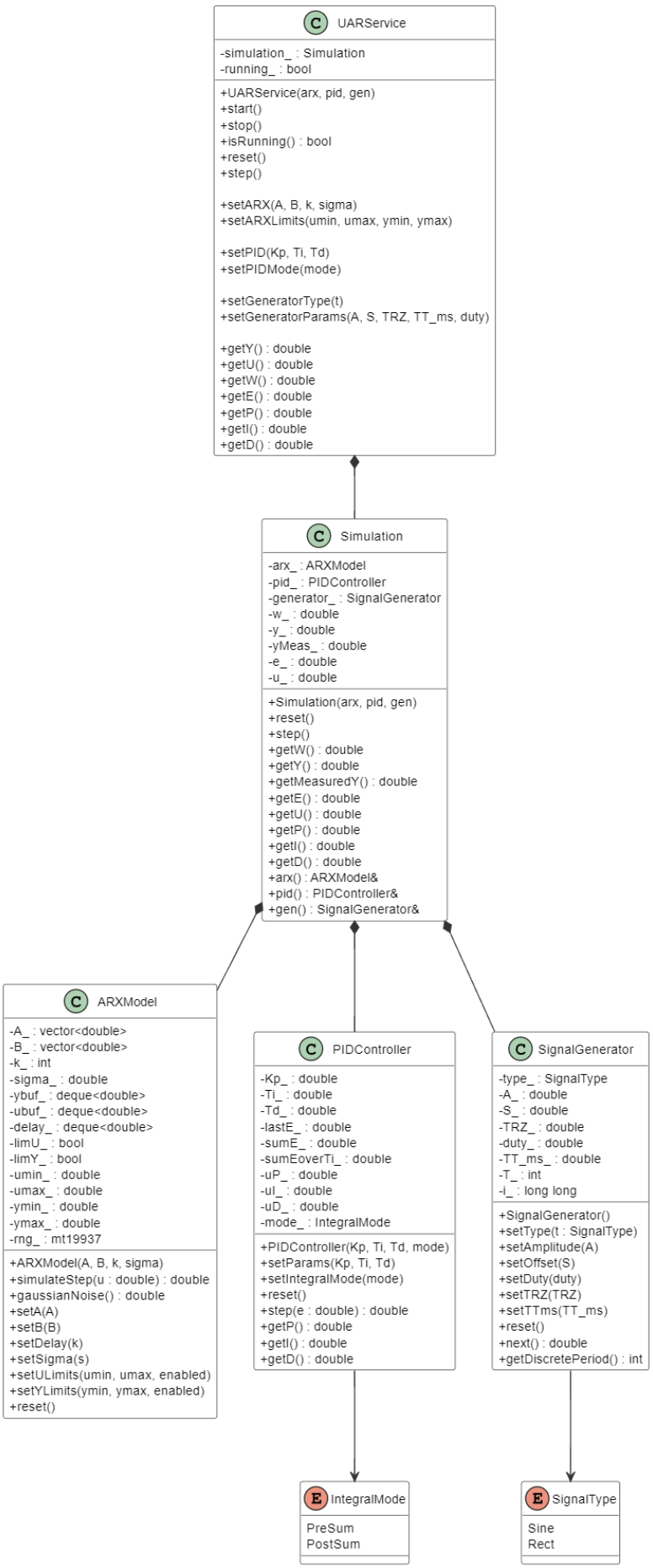
- implementację ARXModel
- testy ARX z wykładu
- podstawy konfiguracji modelu
- wstępne szkice PID i Generatora

Błędny diagram UML z poprzedniego seminarium:


UAR - model klas
GUI
OknoGlowne
-SymulatorUAR* symul
+void start()
+void pauza()
+void krok1()
+void stop()
+void zastosujParametry()
+void eksportCSV()
używa
Warstwa usług
SymulatorUAR
-IObiekt* obiekt
-IRegulator* regulator
-IZrodloZadanej* zadana
-SzeregCzasowy dziennik
-double h
-double t
+SymulatorUAR(IObiekt&, IRegulator&, IZrodloZadanej&, double h=0.2)
+double wykonajKrok()
+void uruchom(int nKrokow)
+void resetujWszystko()
+SzeregCzasowy const& dane() const
+void ustawObiekt(IObiekt&)
+void ustawRegulator(IRegulator&)
+void ustawZrodloZadanej(IZrodloZadanej&)
+void ustawKrokCzasu(double h)
Symulacja
IObiekt
+double krok(double u)
+void reset()
IRegulator
+double oblicz(double r, double y)
+void reset()
IZrodloZadanej
+double nastepna()
+void reset()
SzeregCzasowy
+vector<double> t, r, u, y
+void wyczysc()
+void dopisz(double t, double r, double u, double y)
ObiektARX
-vector<double> A
-vector<double> B
-int k
-double sigma
-deque<double> buforY, buforU, buforOpozn
-bool limitU, limitY
-double uMin, uMax, yMin, yMax
-mt19937 rng
+ObiektARX(vector<double> A, vector<double> B, int k, double sigma=0)
+double krok(double uSurowe)
+void ustawA(vector<double>)
+void ustawB(vector<double>)
+void ustawOpoznienie(int)
+void ustawSigma(double)
+void ustawLimityU(double min, double max, bool wl)
+void ustawLimityY(double min, double max, bool wl)
+void reset()
RegulatorPID
-double kP, Ti, Td
-double calka, poprzedniBlad
-bool limitUwl, antiWindup
-double uMin, uMax
+double oblicz(double r, double y)
+void reset()
+void ustawWzmocnienia(double kP, double Ti, double Td)
+void ustawLimityU(double min, double max, bool wl)
+void ustawAntiWindup(bool wl)
ZadanaSkokowa
-double poziom
-int chwilaSkoku
-int i
+double nastepna()
+void reset()

Architektura systemu po zmianach




UARService
-simulation_ : Simulation
-running_ : bool
+UARService(arx, pid, gen)
+start()
+stop()
+isRunning() : bool
+reset()
+step()
+setARX(A, B, k, sigma)
+setARXLimits(umin, umax, ymin, ymax)
+setPID(Kp, Ti, Td)
+setPIDMode(mode)
+setGeneratorType(t)
+setGeneratorParams(A, S, TRZ, TT_ms, duty)
+getY() : double
+getU() : double
+getW() : double
+getE() : double
+getP() : double
+getI() : double
+getD() : double
Simulation
-arx_ : ARXModel
-pid_ : PIDController
-generator_ : SignalGenerator
-w_ : double
-y_ : double
-yMeas_ : double
-e_ : double
-u_ : double
+Simulation(arx, pid, gen)
+reset()
+step()
+getW() : double
+getY() : double
+getMeasuredY() : double
+getE() : double
+getU() : double
+getP() : double
+getI() : double
+getD() : double
+arx() : ARXModel&
+pid() : PIDController&
+gen() : SignalGenerator&
ARXModel
-A_ : vector<double>
-B_ : vector<double>
-k_ : int
-sigma_ : double
-ybuf_ : deque<double>
-ubuf_ : deque<double>
-delay_ : deque<double>
-limU_ : bool
-limY_ : bool
-umin_ : double
-umax_ : double
-ymin_ : double
-ymax_ : double
-rng_ : mt19937
+ARXModel(A, B, k, sigma)
+simulateStep(u : double) : double
+gaussianNoise() : double
+setA(A)
+setB(B)
+setDelay(k)
+setSigma(s)
+setULimits(umin, umax, enabled)
+setYLimits(ymin, ymax, enabled)
+reset()
PIDController
-Kp_ : double
-Ti_ : double
-Td_ : double
-lastE_ : double
-sumE_ : double
-sumEoverTi_ : double
-uP_ : double
-uI_ : double
-uD_ : double
-mode_ : IntegralMode
+PIDController(Kp, Ti, Td, mode)
+setParams(Kp, Ti, Td)
+setIntegralMode(mode)
+reset()
+step(e : double) : double
+getP() : double
+getI() : double
+getD() : double
SignalGenerator
-type_ : SignalType
-A_ : double
-S_ : double
-TRZ_ : double
-duty_ : double
-TT_ms_ : double
-T_ : int
-i_ : long long
+SignalGenerator()
+setType(t : SignalType)
+setAmplitude(A)
+setOffset(S)
+setDuty(duty)
+setTRZ(TRZ)
+setTTms(TT_ms)
+reset()
+next() : double
+getDiscretePeriod() : int
IntegralMode
PreSum
PostSum
SignalType
Sine
Rect

Warstwa danych

Warstwa danych obejmuje trzy niezależne komponenty modelu układu automatycznej regulacji. Każdy z nich odpowiada za inną część rzeczywistego systemu i jest wykorzystywany przez klasę Simulation.

1. ARXModel – model obiektu regulacji

- Implementuje model liniowy ARX o zadanych współczynnikach A i B
- Przechowuje historię wejść i wyjść (bufory U i Y)
- Obsługuje opóźnienie transportowe *k*
- Zawiera mechanizmy nasycania sygnałów i dodawania szumu Gaussa
- Odpowiada za generowanie wartości *Y(k)* obiektu

2. PIDController – regulator PID

- Implementacja regulatora proporcjonalno–całkująco–różniczkującego
- Obsługa trybów całkowania Pre-Sum / Post-Sum
- Osobne obliczenia członów *P*, *I*, *D*
- Zawiera integrator oraz mechanizmy resetu stanu
- Generuje wartość sterowania *U(k)* na podstawie uchybu *E(k)*

3. SignalGenerator – generator sygnału zadawanego

- Generuje sygnał odniesienia *W(k)*
- Obsługuje typy: sinusoidalny oraz prostokątny
- Parametry: amplituda, offset, okres, duty-cycle, czas trwania
- Pracuje krokowo, zwracając kolejne próbki sygnału
- Może symulować różne przebiegi wejściowe układu

Klasa Simulation (warstwa logiki systemu)

Klasa Simulation łączy wszystkie elementy układu automatycznej regulacji w jedną, spójną pętlę działania. Odpowiada za wykonywanie jednego kroku symulacji oraz utrzymanie aktualnego stanu całego systemu.

Główne odpowiedzialności klasy Simulation

- Integruje działanie **ARXModel**, **PIDController** oraz **SignalGenerator**.
- Realizuje pełną pętlę regulacji w każdej iteracji:
- W(k) → E(k) → P/I/D → U(k) → ARX → Y(k)
- Przechowuje wszystkie bieżące sygnały układu:
    - wartość zadana *W(k)*
    - sygnał sterowania *U(k)*
    - sygnał wyjściowy *Y(k)*
    - błąd *E(k)*
    - składowe P, I, D regulatora

```cpp
#include "Simulation.h"

Simulation::Simulation(const ARXModel& arx,
    const PIDController& pid,
    const SignalGenerator& gen)
    : arx_(arx), pid_(pid), generator_(gen)
{
}

void Simulation::reset()
{
    arx_.reset();
    pid_.reset();
    generator_.reset();

    w_ = y_ = yMeas_ = e_ = u_ = 0.0;
}

void Simulation::step()
{
    // 1. generator
    w_ = generator_.next();

    // 2. wartość mierzona
    yMeas_ = y_;

    // 3. uchyb
    e_ = w_ - yMeas_;

    // 4. sterowanie
    u_ = pid_.step(e_);

    // 5. nowa wartość regulowana
    y_ = arx_.simulateStep(u_);
}

```

UARService – warstwa usług systemu UAR

Klasa **UARService** pełni rolę warstwy usług (service layer) pomiędzy logiką symulacji a przyszłym interfejsem użytkownika (GUI, sieć, panel sterowania). Udostępnia spójny, uporządkowany zestaw metod do konfigurowania i uruchamiania układu regulacji.

Główne zadania UARService

1. Zarządzanie przebiegiem symulacji

- uruchamianie (*start()*)
- zatrzymywanie (*stop()*)
- wykonywanie kolejnych kroków (*step()*)
- reset pełnego systemu (*reset()*)

2. Centralna konfiguracja całego układu

UARService pozwala w jednym miejscu zmienić parametry:

- **ARXModel** → (A, B, k, sigma, limity U/Y)
- **PIDController** → (Kp, Ti, Td, tryb integralny)
- **SignalGenerator** → (typ, amplituda, offset, duty, TT, TRZ)

Dzięki temu interfejs użytkownika nie musi znać szczegółów implementacji.

3. Udostępnianie wszystkich sygnałów wyjściowych

UARService udostępnia prosty zestaw getterów:

- getY() – wyjście modelu
- getU() – sterowanie
- getW() – wartość zadana
- getE() – błąd
- getP(), getI(), getD() – człony PID

Dzięki temu GUI lub testy mogą pobierać dane z jednego miejsca.

projekt_warunek

```cpp
#pragma once
#pragma once

#include "Simulation.h"

class UARService
{
private:
    Simulation simulation_;
    bool running_ = false;

public:
    UARService(const ARXModel& arx,
        const PIDController& pid,
        const SignalGenerator& gen);

    // Sterowanie symulacją
    void start();
    void stop();
    bool isRunning() const;
    void reset();
    void step();

    // Konfiguracja ARX
    void setARX(const std::vector<double>& A,
        const std::vector<double>& B,
        int k,
        double sigma);

    void setARXLimits(double umin, double umax,
        double ymin, double ymax);

    // Konfiguracja PID
    void setPID(double Kp, double Ti, double Td);
    void setPIDMode(PIDController::IntegralMode mode);

    // Konfiguracja generatora
    void setGeneratorType(SignalGenerator::Type t);
    void setGeneratorParams(double A, double S,
        double TRZ, double TT_ms, double duty);

    // Pobieranie sygnałów
    double getY() const;
    double getU() const;
    double getW() const;
    double getE() const;

    double getP() const;
    double getI() const;
    double getD() const;
};
```

Najtrudniejsza funkcjonalność: metoda simulateStep() w ARXModel

Implementacja metody simulateStep() okazała się najbardziej wymagającą częścią projektu.
Jest to kluczowy fragment odpowiadający za wygenerowanie kolejnej próbki wyjścia *Y(k)* modelu obiektu regulacji.

Dlaczego była najtrudniejsza?

1. Złożona synchronizacja buforów

W metodzie działają trzy różne bufory:

- delay_ – implementuje opóźnienie transportowe
- ubuf_ – historia sterowania
- ybuf_ – historia wyjść obiektu

Każdy z nich przesuwa dane w czasie i wymaga precyzyjnej obsługi**.**

2. Sploty wejść i wyjść z wektorami B i A

Realizacja równania ARX wymaga poprawnego obliczania:

$Y(k) = B*u(k-i) - A*y(k-i)$

To oznacza implementację dwóch niezależnych pętli i zachowanie poprawnej kolejności próbek historycznych.

3. Zgodność wyników z testami z wykładu

Metoda musiała przejść:

- test braku pobudzenia,
- trzy testy skoku jednostkowego,
- testy stabilizacyjne,
- testy z nasyceniami.

Nawet drobna pomyłka psuje całą sekwencję.

4. Kilka mechanizmów w jednej funkcji

Opóźnienia, limity, szum, bufory, sploty — wszystko w jednej metodzie → wysoka złożoność.

```cpp
double ARXModel::simulateStep(double u_raw) {
    // 1) LIMIT U - przed jakimikolwiek obliczeniami
    if (limU_) u_raw = std::max(umin_, std::min(u_raw, umax_));

    // 2) Opóźnienie transportowe k (transport delay)
    delay_.push_front(u_raw);
    double u_delayed = delay_.back();
    delay_.pop_back();

    // 3) Aktualizacja bufora sterowania (dla splotu z B)
    ubuf_.push_front(u_delayed);
    if (static_cast<int>(ubuf_.size()) > static_cast<int>(B_.size()))
        ubuf_.pop_back();

    // 4) Sploty: część wejściowa (B·U) i wyjściowa (A·Y)
    double partB = 0.0;
    for (size_t i = 0; i < B_.size(); ++i) partB += B_[i] * ubuf_[i];

    double partA = 0.0;
    for (size_t i = 0; i < A_.size(); ++i) partA += A_[i] * ybuf_[i];

    double y = partB - partA;

    // 5) Szum Gaussa N(0, sigma_) - dodajemy przed nasyceniem Y
    if (sigma_ > 0.0) {
        std::normal_distribution<double> dist(0.0, sigma_);
        y += dist(rng_);
    }

    // 6) LIMIT Y - zaraz po obliczeniu (po szumie), zanim trafi do bufora
    if (limY_) y = std::max(ymin_, std::min(y, ymax_));

    // 7) Aktualizacja bufora wyjścia (pamięci Y)
    ybuf_.push_front(y);
    if (static_cast<int>(ybuf_.size()) > static_cast<int>(A_.size()))
        ybuf_.pop_back();

    return y;
}
```

Najbardziej satysfakcjonująca funkcjonalność: klasa Simulation

Spośród wszystkich elementów projektu najprzyjemniejsza i najłatwiejsza do implementacji okazała się klasa Simulation.
Jej konstrukcja jest przejrzysta, intuicyjna i idealnie odzwierciedla rzeczywistą pętlę regulacji znaną z teorii sterowania.

Dlaczego implementacja Simulation była najłatwiejsza?

1. Naturalna struktura

Simulation jest bezpośrednim odwzorowaniem pętli automatycznej regulacji:

W(k) → PID → U(k) → ARX → Y(k)

Wszystkie elementy działają krok po kroku, co ułatwia implementację.

2. Minimalna logika „wewnętrzna"

W odróżnieniu od:

- ARXModel (bufory, sploty, opóźnienia)
- PIDController (integratory, tryby, P/I/D)
- SignalGenerator (różne typy sygnałów)

Klasa Simulation jedynie koordynuje te elementy.

3. Bardzo mało miejsca na błędy

Brak skomplikowanej matematyki.
Najważniejsze było zachowanie poprawnej kolejności:

1. pobranie wartości zadanej W
2. wyliczenie uchybu E
3. wyliczenie sterowania U
4. obliczenie Y z ARX
5. zapisanie wyników

Proste, przewidywalne, czyste.

4. Szybka integracja z testami

Simulation natychmiast działała poprawnie, bo opiera się na wcześniej przetestowanych elementach (ARX, PID, Generator).

```cpp
#include "Simulation.h"

Simulation::Simulation(const ARXModel& arx,
    const PIDController& pid,
    const SignalGenerator& gen)
    : arx_(arx), pid_(pid), generator_(gen)
{
}

void Simulation::reset()
{
    arx_.reset();
    pid_.reset();
    generator_.reset();

    w_ = y_ = yMeas_ = e_ = u_ = 0.0;
}

void Simulation::step()
{
    // 1. generator
    w_ = generator_.next();

    // 2. wartość mierzona
    yMeas_ = y_;

    // 3. uchyb
    e_ = w_ - yMeas_;

    // 4. sterowanie
    u_ = pid_.step(e_);

    // 5. nowa wartość regulowana
    y_ = arx_.simulateStep(u_);
}

```

Testy jednostkowe i integracyjne

W ramach etapu II wykonano pełny zestaw testów, aby potwierdzić poprawność działania zarówno poszczególnych klas, jak i ich współpracy w pętli regulacji.

Testy obejmowały:

Testy jednostkowe (dla każdej klasy osobno)

1. ARXModel

Cel: sprawdzić, czy równanie ARX i bufory działają poprawnie.

2. PIDController

Cel: upewnić się, że regulator generuje prawidłowe sterowanie U(k).

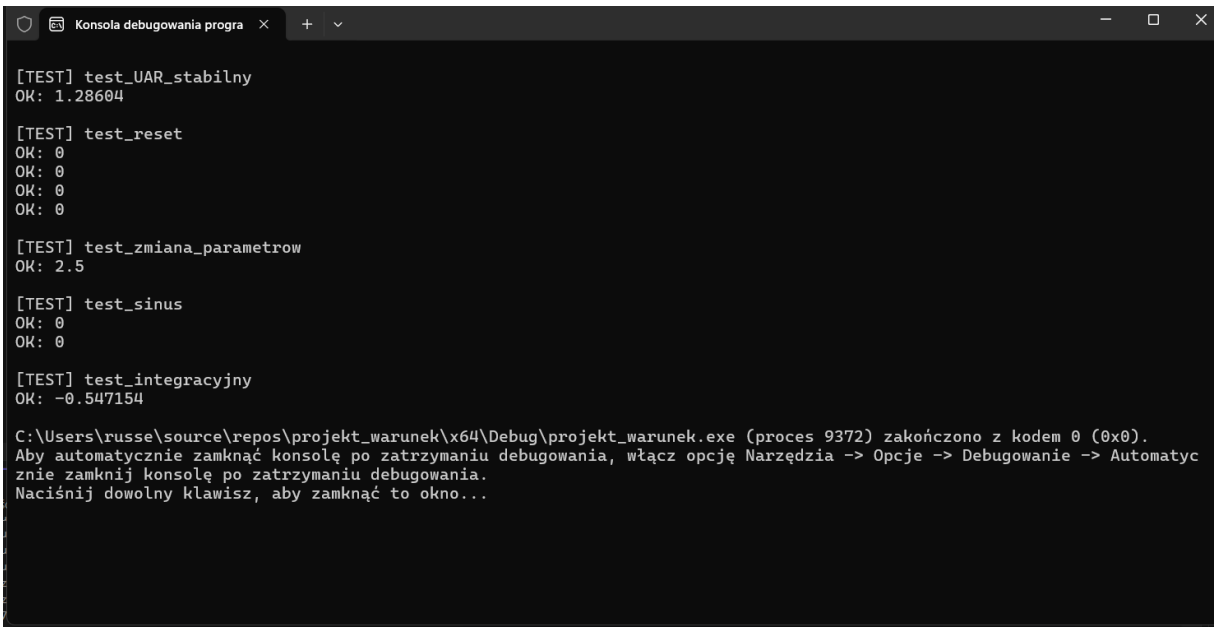
3. SignalGenerator

Cel: zweryfikować poprawność przebiegów W(k).

Testy integracyjne (pełny układ regulacji)

W testach integracyjnych uruchomiono całą pętlę regulacji:

W(k) → E(k) → PID → U(k) → ARX → Y(k)



Podsumowanie

W drugim etapie seminarium zakończono implementację pełnego backendu układu automatycznej regulacji (UAR). Prace obejmowały zarówno rozbudowę architektury, jak i szczegółowe testowanie elementów systemu.