Projekt 3: Układ automatycznej regulacji

Tematem projektu jest sterowanie temperaturą w wirtualnym pomieszczeniu, podlegającym prostej dynamice (pokój oddaje ciepło otoczeniu o określonej temperaturze). Na podstawie pomiaru aktualnej temperatury w pomieszczeniu należy tak sterować grzejnikiem, by utrzymywać temperaturę jak najbliższą zadanej.

Implementacja klasy `Pomieszczenie`, wraz z równaniami termodynamiki, jest podana i nie wolno jej w żaden sposób modyfikować.

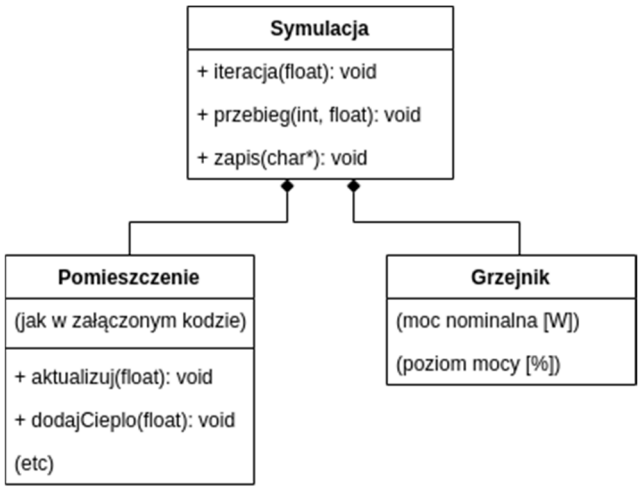
Przed przystąpieniem do wykonywania prac z danego etapu, przeczytaj uważnie wszystkie polecenia!

Etap 1: Symulacja UAR. Szablon std::vector.

Aby zasymulować działanie jakiegokolwiek układu dynamicznego, należy zasymulować upływ czasu. Czas „wirtualny" wcale nie musi (a wręcz nie powinien!) pokrywać się z czasem rzeczywistym. W tym projekcie wystarczy nam najbardziej podstawowe rozwiązanie, tzn. prosta pętla wykonująca pewną ilość iteracji, z których każda odpowiada kolejnej chwili czasu wirtualnego, oddalonej o jakąś ustaloną wartość Δt (czas próbkowania). W każdej iteracji stan obiektów wchodzących w skład symulacji aktualizowany jest według równań tych obiektów – np. w symulacji fizycznej obiekt poruszający się z prędkością v (która byłaby atrybutem tego obiektu) byłby przesuwany o odległość $v \cdot \Delta t$. W niniejszym projekcie równania termodynamiki pomieszczenia są już zaimplementowane w klasie `Pomieszczenie` (a konkretnie jej metodzie **`aktualizuj`**). Twoim pierwszym zadaniem będzie oprogramowanie klasy nadrzędnej `Symulacja`, służącej do zarządzania upływem czasu i regularnego aktualizowania stanu obiektu sterowania.

W następnej kolejności do symulacji dodasz drugi komponent: `Grzejnik`, który w zależności od swojej mocy maksymalnej (ustalanej przy konstrukcji) oraz obecnego poziomu mocy będzie emitował pewną ilość ciepła. Zmodyfikujesz `Symulację` w taki sposób, by ciepło emitowane przez `Grzejnik` wpływało na `Pomieszczenie`.

W ostatnim kroku wzbogacisz `Symulację` o możliwość logowania jej przebiegu (np. do generowania wykresów przy użyciu arkusza kalkulacyjnego). W tym celu zastosujesz szablon **`std::vector`**. Stan projektu po etapie pierwszym dany jest na poniższym diagramie UML:



PROJEKT 3: UKŁAD AUTOMATYCZNEJ REGULACJI

Zadania:

1. Oprogramuj klasę `Symulacja`. Metoda **`iteracja`** ma wykonywać pojedynczy krok symulacji, przyjmując liczbę zmiennoprzecinkową określającą czas upływający w tym kroku (Δt). Metoda **`przebieg`** ma pozwalać na wykonanie określonej liczby iteracji o podanym czasie próbkowania. W każdym kroku aktualizuj stan `Pomieszczenia` i wydrukuj jego aktualną temperaturę na konsoli.
   Przetestuj działanie symulacji obserwując, jak temperatura pomieszczenia stopniowo wyrównuje się z otoczeniem.
2. Oprogramuj klasę `Grzejnik`, wyposażoną w następujące atrybuty oraz wszelkie potrzebne do nich akcesory:
   - moc nominalna (wartość zmiennoprzecinkowa, stała, ustalana przy konstrukcji obiektu),
   - aktualny poziom mocy (wartość zmiennoprzecinkowa (0 – 1) oznaczająca procent mocy emitowanej w danej chwili)

   Dodaj możliwość obliczania aktualnie emitowanego ciepła.

   Zadbaj o zabezpieczenia! Ustawienie poziomu mocy 200% nie powinno być możliwe.

   Zmodyfikuj kod symulacji tak, by ciepło grzejnika wpływało na pomieszczenie. Przetestuj działanie symulacji przy różnych ustawieniach mocy grzejnika.

3. Rozszerz klasę `Symulacja` o automatyczne zapisywanie przebiegów: temperatury, aktualnego czasu, oraz aktualnej mocy wyjściowej grzejnika używając szablonu **`std::vector`**. Dopisz do niej metodę **`zapis`**, umożliwiającą eksport tych danych do pliku CSV. Przetestuj jej działanie, rysując w ulubionym arkuszu kalkulacyjnym wykresy temperatury i mocy (sposób zapisu liczb zmiennoprzecinkowych z przecinkiem zamiast kropki przedstawiono na końcu instrukcji *Strumienie*). Sprawdź, co stałoby się, gdyby w połowie symulacji zmienić poziom mocy grzejnika.

Etap 2: Sterowanie procesem. Polimorfizm.

W tej części projektu zaimplementujesz proste algorytmy sterowania procesami. Będą to regulator dwupołożeniowy oraz regulator PID. Ogólnie, zadaniem regulatora jest generowanie takiego sygnału sterującego, by obiekt sterowany zachowywał się w określony sposób. W naszym przypadku: takie sterowanie mocą grzejnika, by temperatura w pomieszczeniu utrzymywała się na zadanym poziomie. Oczywistą konsekwencją jest konieczność wykonywania *pomiaru* obecnego stanu obiektu (temperatury pomieszczenia).

Regulator dwupołożeniowy (także *dwustawny* lub „bang-bang") jest najprostszym możliwym regulatorem: jak sama nazwa wskazuje, może generować tylko dwa możliwe sygnały sterujące: 0 (brak sterowania) i 1 (sterowanie maksymalne) – czyli brak grzania lub grzanie z pełną mocą. Jego algorytm jest więc bardzo prosty: jeśli temperatura jest mniejsza od zadanej – grzej, w przeciwnym razie – nie grzej.

Regulator PID jest bardziej złożonym regulatorem, działającym na podstawie *uchybu* (e), czyli różnicy pomiędzy wartością zadaną (w) a obecnie zmierzoną (y). W istocie składa się on z trzech prostych regulatorów składowych:

- części proporcjonalnej (P) – która generuje sygnał sterujący (u) *proporcjonalny* do uchybu, zgodnie z równaniem:
  $u_p = K_p \cdot e$
- części całkującej (I) – która sumuje (*całkuje*) uchyb w czasie i steruje proporcjonalnie do tej całki:
  $e_{calka} = e_{calka} + e \cdot \Delta t$
  $u_i = K_i \cdot e_{calka}$
- części różniczkującej (D) – której sygnał sterujący zależy od pochodnej uchybu, którą można oszacować poprzez podzielenie różnicy pomiędzy poprzednią a obecną wartością uchybu przez czas, który upłynął między pomiarami:
  $\Delta e = \frac{e - e_{poprzednia}}{\Delta t}$
  $u_d = K_d \cdot \Delta e$

Całkowity sygnał sterujący regulatora PID to suma sygnałów z poszczególnych jego komponentów:

$$u = u_p + u_i + u_d$$

Co jest potrzebne, żeby zaimplementować taki regulator? Jak wynika z równań powyżej, musisz znać wartość zadaną, do której układ ma dążyć (dowolna) oraz wartość obecną (przechowywaną w obiekcie `Pomieszczenie`) aby obliczyć uchyb w danej chwili, sumę uchybu we wszystkich chwilach poprzednich, oraz uchyb w chwili poprzedniej. Zastanów się, które z tych wartości będą lokalne, a które będą stanowić atrybuty klasy `RegulatorPID`.

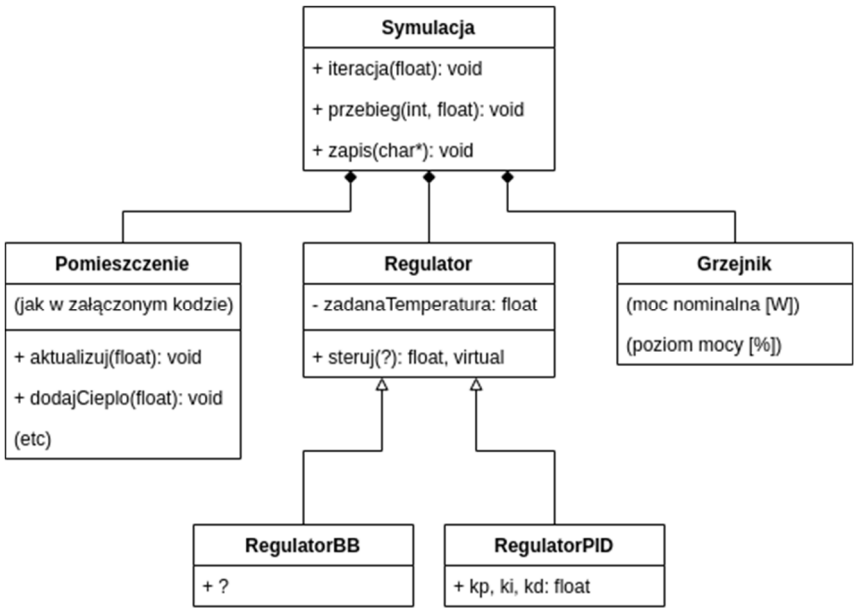
Co jest potrzebne, żeby z takiego regulatora skorzystać? Oczywiście, nastawy poszczególnych członów K_p, K_i, K_d. Wyznaczanie ich jest tematem złożonym, o którym dowiesz się jeszcze w toku studiów. Na potrzeby tego projektu wypróbuj wartości, odpowiednio: $4.0, 0.02, 0.1$, lub poeksperymentuj z innymi.

Ponieważ regulatory co do zasady wszystkie działają tak samo – wchodzi wartość zadana oraz obecna, wychodzi sygnał sterujący – to naturalne będzie zaimplementowanie ich w sposób polimorficzny. Twoim zadaniem będzie:

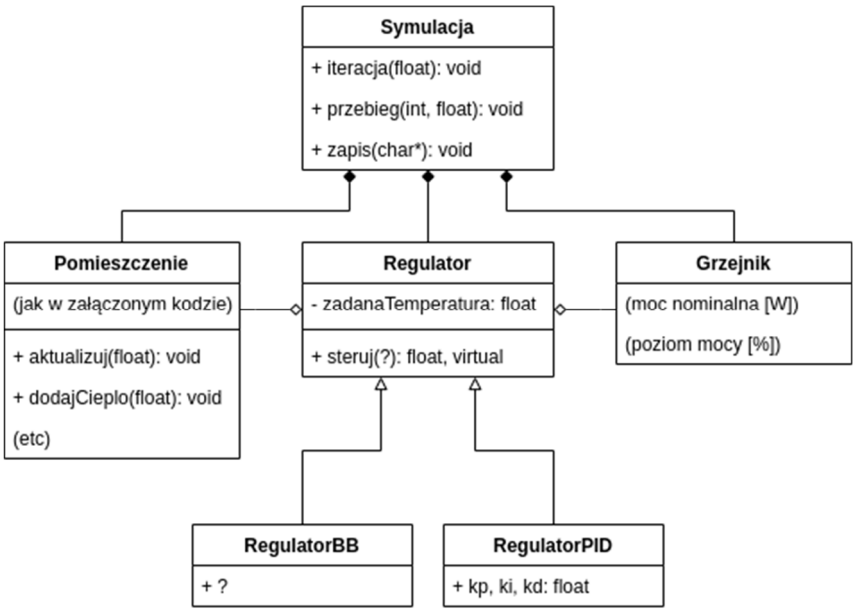
PROJEKT 3: UKŁAD AUTOMATYCZNEJ REGULACJI

1. Zaimplementowanie klasy abstrakcyjnej (może być czysto wirtualna) `Regulator`, której metoda **`steruj`** przyjmie wartość zadaną temperatury, wartość zmierzoną i czas między pomiarami (Δt), a zwróci pożądany poziom mocy grzejnika.
   Uwzględnij w `Symulacji` możliwość posiadania w sposób polimorficzny instancji `Regulatora` wraz z odpowiednimi metodami do jej ustawienia, oraz kodem do obsługi jego działania w trakcie poszczególnych iteracji symulacji.
2. Zaimplementowanie klasy `RegulatorBB` (od „bang-bang") dziedziczącej po `Regulatorze`. Ma ona być wyposażona w implementację metody **`steruj`** wykonującą opisany powyżej algorytm sterowania dwupołożeniowego. W tym regulatorze czas pomiędzy pomiarami nie będzie wykorzystany.
   Przetestuj działanie takiego regulatora w symulacji.
3. Zaimplementowanie na podobnej zasadzie klasy `RegulatorPID`, również dziedziczącej po `Regulatorze`.
   Przetestuj regulator PID w symulacji.
4. Wprowadzenie agregacji `Regulatora` z `Pomieszczeniem` i `Grzejnikiem`. Dzięki temu to metoda `Regulator::steruj` (a nie `Symulacja::iteracja`) będzie pobierać obecną temperaturę z `Pomieszczenia`, wyznaczyć sygnał sterujący i odpowiednio ustawić moc wyjściową `Grzejnika`. Czy metoda `Regulator::steruj` nadal musi zwracać poziom mocy grzejnika i przyjmować wartość zmierzoną temperatury? Niech konstruktory regulatorów pozostaną możliwie najprostsze – jeśli potrzebujesz przypisać do nich instancje klas posiadanych, uczyń to w osobnej metodzie.
5. Spełnienie w programie zasad programowania obiektowego: enkapsulacji, zasadę trzech.
6. Oprogramowanie prostego menu pozwalającego użytkownikowi wybrać rodzaj regulatora przed rozpoczęciem symulacji.
7. Dla ambitnych: oddelegowanie konstrukcji konkretnych regulatorów do osobnej klasy zgodnie ze wzorcem projektowym *Factory*. Wskazówka: prototyp metody do konstrukcji regulatora PID mógłby wyglądać następująco:
   ```
   static Regulator* stworzRegulatorPID(float kp, float ki, float kd);
   ```

Do punktu 4 projekt powinien wyglądać tak jak na poniższym diagramie UML:



Finalnie natomiast, jak poniżej:



Ta drobna zmiana w diagramie ma duże konsekwencje w logice programu. Być może uznasz, że można od razu program napisać w tej ostatecznej formie – patrząc od strony końcowego programu, jest to jak najbardziej słuszna koncepcja. Jednak dobrym ćwiczeniem będzie dla Ciebie prześledzenie zmian zachodzących w kodzie przy takiej zmianie struktury, dlatego do tego zachęcamy.

Etap 3: Wyjątki.

Użycie regulatora z poprzedniego etapu w wersji z agregacją wymagało uprzedniego przypisania obiektowi `Regulator` odniesień do obiektów `Pomieszczenie` i `Grzejnik`, najprawdopodobniej poprzez wywołanie jakiejś metody (akcesora). Tylko wtedy możliwe było bezpośrednie działanie regulatora na tych obiektach (bez pośrednika w postaci `Symulacji`). Co jednak wtedy, gdy symulacja zostanie wykonana bez ustawienia tych instancji?

Może Ci się wydawać, że w tak prostym programie w zupełności wystarczy oprogramowanie zabezpieczenia na poziomie interfejsu użytkownika (jeśli nie stworzono regulatora – nie pozwól na uruchomienie symulacji). W pracy nad większymi projektami nie będziesz jednak mieć kontroli nad całym kodem i może być od Ciebie wymagane wprowadzenie zabezpieczeń na poziomie Twojego fragmentu kodu: na przykład, programując Regulator, należy zadbać, by nie „wysypywał się" przy nieprawidłowych operacjach, takich jak np. próba wywołania metody **`steruj`** bez uprzedniego skonfigurowania regulatora.

Zadania:

1. Wykryj próbę uruchomienia regulatora bez skonfigurowania powiązań z innymi obiektami i zabezpiecz ją poprzez rzucenie wyjątku. Zaplanuj i zaimplementuj obsługę tego wyjątku poza obiektem `Regulator`.
   Obsługa wyjątków w programie może być kosztowna (mechanizm zwijania/rozwijania stosu). Upewnij się, że blok `try` nie obejmuje jednocześnie zbyt wielu wywołań – na przykład objęcie nim całej zawartości funkcji **`main`** byłoby zdecydowanie błędem. Czy powiązanie regulatora z innymi obiektami musi być sprawdzane w każdej iteracji?
   Przetestuj wychwyt wyjątku, celowo wykonując program w „błędny" sposób. Jeśli posiadasz zabezpieczenia na poziomie menu – deaktywuj je na czas testów.
2. Przed jakimi innymi sytuacjami należałoby zabezpieczyć program? Jest co najmniej jedna, ważna zwłaszcza w regulatorze PID.
3. Dla ambitnych: skorzystaj z pliku nagłówkowego `exception` i wyjątków standardowych. Rozważ implementację swojej własnej klasy wyjątków.
4. Przejrzyj poprzednie projekty. Czy dostrzegasz potencjalne problemy, przed którymi można się zabezpieczyć przy pomocy wyjątków? W każdym projekcie jest co najmniej jeden. W ramach ćwiczenia, zaimplementuj obsługę wyjątków (to ćwiczenie nie będzie sprawdzane – jest ono tylko dla Ciebie).