Dokumentacja końcowa

Temat projektu to przeprowadzenie symulacji działania hotelu.

Zespół : Piotr Gierżatowicz-Sierpień, Wojciech Kukiełka

Uruchomienie symulacji i jej działanie

Symulacja uruchamiana jest poprzez wywołanie program z dwoma argumentami, pierwszym z nich powinien być *workers.json*, który zawiera listę pracowników w formacie:

```
{
    "id": "id1",
    "name": "name1",
    "pay": {
        "amount": "3200.00",
        "method": "Salary"
    },
    "type": "Receptionist"
},
{
```

Parametry *id* i *name* to dowolne ciągi znaków, w parametrze *pay* znajdują się pola *amount* i *method* wykorzystywane przez system rozliczeń hotelu.

Drugim z nich jest plik *rooms.json*, w formacie przedstawionym poniżej:

```
{
    "id": "237",
    "type": "FourRoom"
},
```

id pokoju jest dowolne, a *type* jednym z dostępnych w hotelu pokoi.

Wynikiem symulacji jest plik output.txt, w którym są zawarte informacje dotyczące zdarzeń, które wystąpiły w hotelu. Obsługiwanymi zdarzeniami są pojawienia się gości w hotelu, chcących zamówić pobyt.

```
2024-06-07 18:00:00: 3 guests came to the hotel, they booked a stay ending at 2024-06-09 10:00:00
2024-06-09 23:00:00: 1 guests came to the hotel, they booked a stay ending at 2024-06-11 10:00:00
2024-06-10 00:00:00: 4 guests came to the hotel, they booked a stay ending at 2024-06-12 10:00:00
2024-06-10 11:00:00: 1 guests came to the hotel, they booked a stay ending at 2024-06-12 10:00:00
2024-06-10 16:00:00: 1 guests came to the hotel, but there was no available room for them.
```

Wykorzystanie biblioteki STL

Najczęściej wykorzystywane są algorytmy i filtry do obsługi kolekcji obiektów. Mimo, że w projekcie głównymi klasami do reprezentacji czasu są *datetime* i *timespan*, niezwiązane z STL, *<chrono>* znalazło zastosowanie m. in. do znalezienia ostatniego dnia miesiąca. Do generowania liczb losowych użyto *<random>*, które daje większe możliwości niż *rand()* z C, np. są do wyboru rozkłady. Oczywiście program nie mógłby działać bez sprytnych wskaźników z *<memory>*.

Testy

Większość kodu hotelu zostało przetestowane przez testy jednostkowe. Samą symulację testowaliśmy, uruchamiając program.

Testy jednostkowe okazały się pomocne podczas modyfikacji wcześniej napisanych klas, albowiem było trochę błądzenia we mgle, zapobiegało to regresji. Merge można było zrobić tylko wtedy, gdy wszystkie testy znów przechodziły.

Systemy hotelu

Działanie hotelu to kooperacja systemów. Większość służy jako kontenery na typy z nimi związane, *GuestSystem* zawiera gości. *HotelSystem* zawiera wszystkie te systemy, oraz zegar, synchronizujący niektóre systemy. Daje ona dostęp do swoich systemów.

Interfejsy systemów

Systemy pozwalają na dodanie obiektu do kolekcji, musi mieć on unikatowe id, bo potem obiekty o tym samym ID są traktowane jako równe. Obiekt dodawany do kolekcji to nie ten sam, który został podany metodzie dodającej, metoda ta zwraca stałą referencję na nowy obiekt.

Dlaczego stałą? – Otóż chodzi o to, aby po dodaniu obiektu do systemu kontrolować jest stan. Obiekty nie związane z systemem mają większą swobodę, jeżeli chodzi o settery. Przykładowo, przed dodaniem do systemu, *Task'owi* można przypisać dowolnego pracownika odpowiedniego typu. Po dodaniu do *TaskSystemu* , musi to być pracownik znajdujący się w *WorkerSystemie*. Oczywiście jeśli *Task* ma przypisanego pracownika w momencie dodawania, to ten warunek jest sprawdzany i gdy jest niespełniony, zgłaszany jest *WorkerNotInSystemError*.

Można wyszukiwać obiekt po ID – metoda bezpieczna. Można go też żądać, trzeba się liczyć z wyjątkiem. Można obiekt usunąć, to jest ciekawe, bo co się wtedy dzieje z obiektami z innych systemów, które trzymały wskazanie na usunięty obiekt? Są to tzw. Obiekty zależne, będzie o nich dalej.

Oprócz tego można modyfikować te obiekty pod kontrolą systemu, tzn. korzystając z jej metod, system kontroluje czy ta modyfikacja jest dozwolona, *StaySystem* rzuci błąd, gdy spróbujemy zameldować *Stay*, który się jeszcze nie zaczął.

Działanie systemów

Działanie systemów zostanie omówione na czterech przykładach:

- Kontener na prosty typ, niezależny od innych typów
- Kontener na prosty typ, zależny od innych
- Kontener na typ polimorficzny, niezależny od innych
- Kontener na typ polimorficzny zależny od innych

Typ prosty niezależny

Obiekt jest niezależny, ale inne obiekty mogą być od niego zależne, chcemy zapewnić, aby znajdował się on w stałym miejscu pamięci, oraz chcemy powiadomić obiekty od niego zależne, gdy zostanie usunięty. Przykład, nie chcemy przechowywać *Tasków* w nieskończoność, po wykonaniu chcemy je usunąć, ale *Service'y*, które zależą od tego *Taska* (obchodzi je jedynie czy został wykonany) możemy chcieć trzymać do końca pobytu gościa, żeby się z nim rozliczyć. Service musi wiedzieć jak zareagować na taką sytuację

Stałe miejsce w pamięci można zagwarantować, własnoręcznie alokując obiekt na stercie. Robi się to trzymając w systemie wektor łebskich wskaźników.

System musi komunikować innym systemom, że obiekt został usunięty, mówi się, że jest on *OtherSystemPublisherem*. Wtedy zainteresowany system – *OtherSystemObserver* może mu zgłosić, że chce od niego otrzymywać powiadomienia o usunięciach.

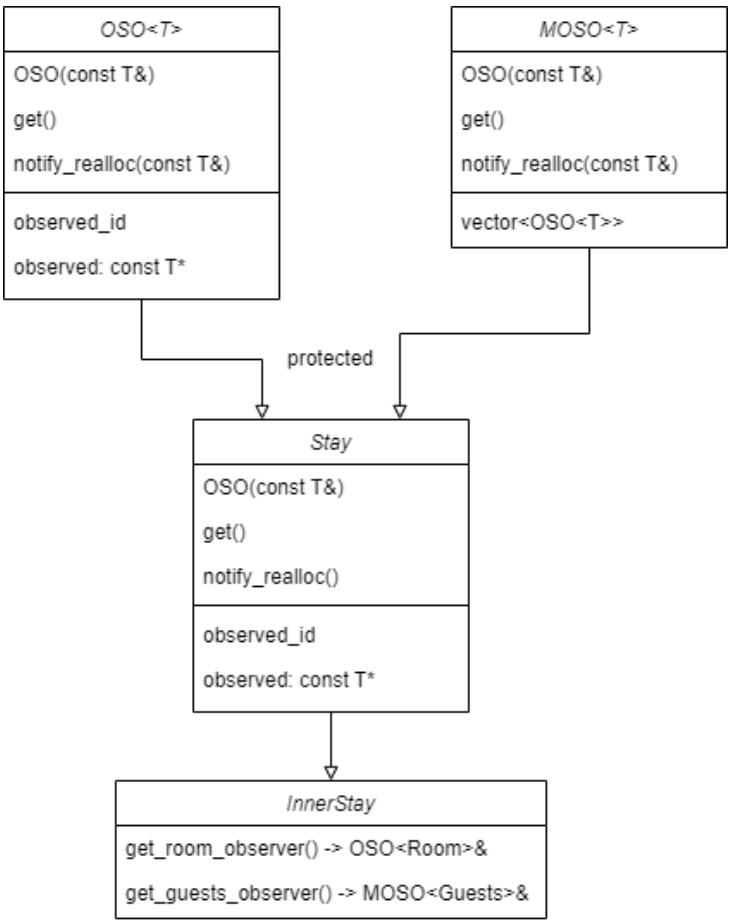
Typ prosty zależny

System obserwujący otrzymał powiadomienie o usunięciu obiektu typu takiego o id takim. Może on usunąć obiekty związane z tamtym obiektem.

Jest inny problem. Zanim obiekt zostanie dodany do systemu, chcemy, żeby korzystać z jego pełnego interfejsu. Po utworzeniu *Stay'a*, przed dodaniem, chcemy wywołać na nim *get_room()* i dostać referencję na pokój, a ten pokój wcale nie musi należeć do systemu pokoi. Z tych klas można korzystać niezależnie od systemów, klasy nie wiedzą, że są jakieś systemy. Po dodaniu, chcemy, aby wskazanie na pokój zostało przepięte, jeśli to możliwe, na ten właściwy, należący do systemu.

Obiekt dziedziczy po *OwnSystemObserverze* (*OSO*), albo *MultipleOwnSystemObserverze* (*MOSO*), dziedziczenie jest protected. *Stay* dziedziczy po *OSO*<*Room*> i *MOSO*<*Guest*>. *OSO* trzyma wskazanie na obiekt o danym ID i kontroluje dostęp do niego, nie nastąpi dereferencja *nullptr'a*. *OSO* i *MOSO* mają metody *notify_realloc()*, którymi można przepiąć wskazanie na inny obiekt o tym samym ID, wykorzystanie przy dodawaniu do systemu.

Stay jest odseparowany od tego, dostaje on dostęp m. in. do metody *get()*, która zwraca obserwowany obiekt. Dodatkowo użytkownik jest odseparowany od *observerów* w ogóle, nie widzi metod get() i *notify_realloc()*. Ale jak to? Więc skąd system wie o tym dziedziczeniu? – po *Stayu* dziedziczy *InnerStay* i to obiekty tego typu są tworzone w kontenerze, użytkownik dostaje do nich dostęp przez referencję do *Stay*. *InnerStay* ma metody zwracające referencje do *OSO* i *MOSO*, zwraca samego siebie.



Typ polimorficzny niezależny

Tym razem system ma w pewnym sensie „wiedzieć mniej" o typie, który przechowuje. W każdej chwili może się dowiedzieć, robiąc *dynamic_cast*, ale w wielu sytuacjach wystarcza mu korzystać z interfejsu klasy bazowej, w tym z jej metod wirtualnych. Upraszcza to jego kod i sprawia, że jest przyjemniejszy w utrzymaniu.

WorkerSystem przy dodawaniu alokuje na stercie *Workera* odpowiedniego typu. Trzyma wektor *unique_ptr'ów* na *Workera*. Metoda *get_workers()* zwraca użytkownikowi wektor zwykłych pointerów na *const Workera*.

Typ polimorficzny zależny

RoomCleaningTask różni się od *TaxiTask*. Do tego pierwszego można przypisać kilka pokojówek, do drugiego tylko jednego recepcjonistę. Ważną i problematyczną różnicą jest to, że pierwszy trzyma wskazanie na pokój, drugi nie, ale za to trzyma wskazanie na gościa.

TaskSystem, wiedząc mniej o *Taskach*, które trzyma, musi liczyć się z tym, że każdy z nich potrzebuje każdej informacji od innych systemów. Tak więc, będzie powiadamiał o wszystkim, a Task sam zadecyduje co zrobi z tą informacją.

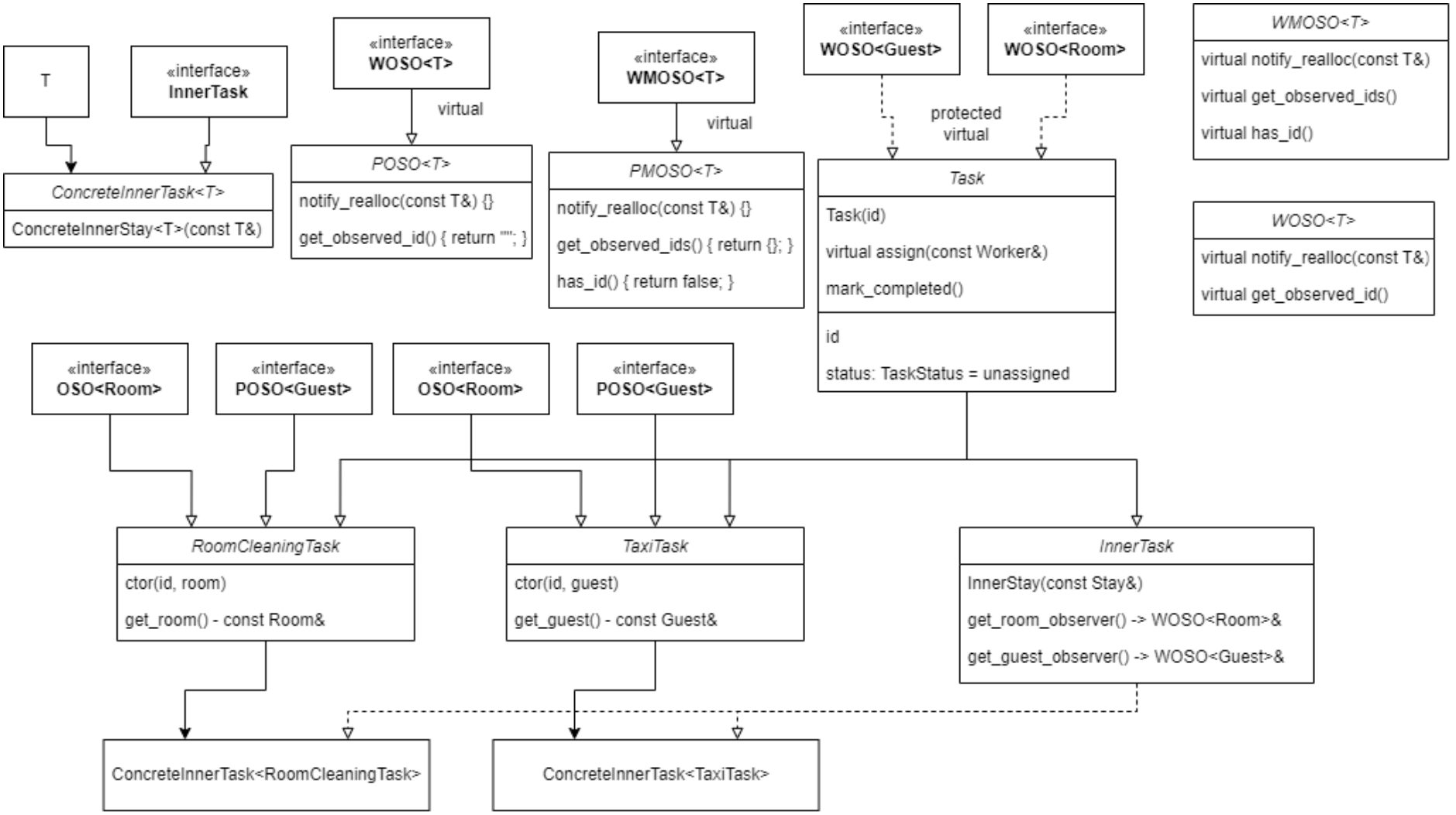
Przytoczone wcześniej klasy *OSO* i *MOSO* mają atrybuty i metody, które mają sens jedynie, gdy faktycznie obserwujemy jakiś obiekt. *TaskSystem* nie potrzebuje informacji o tych metodach, nawet lepiej żeby jej nie miał. Definiujemy ABC *WeakOwnSystemObserver* (*WOSO*) oraz wersję mnogą (*WMOSO*), które mają wirtualne metody potrzebne *TaskSystemowi*. *Task* dziedziczy (protected) po wszystkich *WeakObserverach*, które są stosowne do jego klas potomnych. Klasa potomna musi implementować te metody, może to być implementacja, która nic nie robi, ignoruje otrzymaną informację.

Żeby to prosto zrobić, wykorzystując napisane już *OSO* i *MOSO*, mówimy, że dziedziczą one wirtualnie odpowiednio po *WOSO* i *WMOSO*, implementując wymagane metody, oraz parę innych. *Task* również dziedziczy wirtualnie po *WOSO* i *WMOSO*. Można w ten sposób zaimplementować te metody w klasie pochodnej, zwyczajnie dziedzicząc po *OSO* lub *MOSO*.

Żeby zaimplementować metody ignorujące, piszemy klasy *PseudoOwnSystemObserver* (*POSO*) i *PMOSO*. Nie mają żadnych pól, nie śmiecą, implementują interfejs *WOSO* i *WMOSO*.

Implementując pochodny *Task*, wystarczy dla każdego *WOSO* i *WMOSO*, po których dziedziczy *Task*, dobrać dziedziczenie po odpowiednio *OSO* albo *POSO* i *MOSO* albo *PMOSO*.

Typy, które przechowuje *TaskSystem* muszą mieć informacją o dokładnym typie Taska, muszą też dziedziczyć po interfejsie *InnerTask*. Stąd *ConcreteInnerTask*<*PrepareDishTask*> itd..


T
«interface»
InnerTask
ConcreteInnerTask<T>
ConcreteInnerStay<T>(const T&)
«interface»
WOSO<T>
virtual
POSO<T>
notify_realloc(const T&) {}
get_observed_id() { return ""; }
«interface»
WMOSO<T>
virtual
PMOSO<T>
notify_realloc(const T&) {}
get_observed_ids() { return {}; }
has_id() { return false; }
«interface»
WOSO<Guest>
«interface»
WOSO<Room>
protected
virtual
Task
Task(id)
virtual assign(const Worker&)
mark_completed()
id
status: TaskStatus = unassigned
WMOSO<T>
virtual notify_realloc(const T&)
virtual get_observed_ids()
virtual has_id()
WOSO<T>
virtual notify_realloc(const T&)
virtual get_observed_id()
«interface»
OSO<Room>
«interface»
POSO<Guest>
«interface»
OSO<Room>
«interface»
POSO<Guest>
RoomCleaningTask
ctor(id, room)
get_room() - const Room&
TaxiTask
ctor(id, guest)
get_guest() - const Guest&
InnerTask
InnerStay(const Stay&)
get_room_observer() -> WOSO<Room>&
get_guest_observer() -> WOSO<Guest>&
ConcreteInnerTask<RoomCleaningTask>
ConcreteInnerTask<TaxiTask>