Implementacja

Pliki źródłowe zostały podzielone na kilka pakietów:

* sk.sim.activities - zawiera klasy, odpowiadające za wszystkie aktywności poszczególnych obiektów (np. Przygotowywanie zamówienia)
* sk.sim.objects - zawiera klasy, które zawierają informacje o symulowanych obiektach np. Gość, Kelner
* sk.sim.gui - elementy odpowiedzialne za graficzny interfejs użytkownika.

Najważniejszą klasą aplikacji jest klasa RestaurantSimObject. Klasa przechowuje podstawowe informacje dotyczące przeprowadzania symulacji tj.: liczbę: kelnerów, kucharzy, potraw i napojów. Struktura ta odpowiada także za uruchamianie aktywności: przychodzenia nowych gości, uruchomienie przyjmowania gości przez kelnerów oraz uruchamia pracę kucharzy. Klasa przechowuje także kolejkę nowych gości. Znajdują się tu również dwie flagi, które oznaczają czy restauracja jest zamknięta, oraz czy nadchodzą kolejni goście. Jest to potrzebne w celu lepszej synchronizacji pracy   
kelnerów i kucharzy. Najważniejsze elementy kodu źródłowego klasy RestaurantSimObject:

|  |
| --- |
| **public class** RestaurantSimObject **extends** SimObject {  **public static int** *GUEST\_NUMBER* = 20;  **public static int** *WAITERS\_NUMBER* = 3;  **public static int** *MEALS\_NUMBER* = 10;  **public static int** *DRINKS\_NUMBER* = 5;  **public static int** *COOKER\_NUMBER* = 3;   **public static** AtomicBoolean *opened* = **new** AtomicBoolean(**true**);   **public static** AtomicBoolean *guestsComing* = **new** AtomicBoolean(**true**);   **public static** Deque<GuestSimObject> *expectantGuests* = **new** LinkedBlockingDeque<>();   **public static** List<GuestSimObject> *servicedGuests* =   Collections.*synchronizedList*(**new** ArrayList<>());   **private static** Map<CookSimObject, TakingOrderActivity> *cooks* = **new** HashMap<>();  **private static** Map<WaiterSimObject, InvitingGuestActivity> *waiters* =  **new** ConcurrentHashMap<>();    **private** SimActivity **newGuestComingActivity**;  **public** RestaurantSimObject()  {  initMealsAndDrinks();waitersInitialization();  cookerInitialization();   **newGuestComingActivity** = **new** NewGuestComingActivity();   SimActivity.*callActivity*(**this**, **newGuestComingActivity**);  **this**.callWaiters();  **this**.callCooks();  }  **public static void** newGuestCome(GuestSimObject guestSimObject)  {  RestaurantSimObject.*expectantGuests*.addLast(guestSimObject);  Logger.*log*(**"Nowy gosc w kolejce: "** + guestSimObject);   Guest guestVisualisation = **new** Guest(guestSimObject);  VisualisationLog.*addNewGuest*(guestVisualisation);  }  } |

Metoda newGuestCome(GuestSimObject guestSimObject), rejestruje nowego gościa w kolejce do restauracji.

Aktywności

Generowanie nowych gości

Klasa NewGuestComingActivity jest klasą, która odpowiada za generowanie nowych gości:

|  |
| --- |
| **public class** NewGuestComingActivity **extends** SimActivity {  **private** AtomicInteger **counter** = **new** AtomicInteger();   @Override  **public void** action()  {  **while** (**true**)  {  addGuest();  waitDuration(1000);  **if** (RestaurantSimObject.*expectantGuests*.size() > 6)  {  Logger.*log*(() -> **"\nNowi goscie pause\n"**);  waitDuration(100000);  }   **if** ( **counter**.getAndIncrement() > RestaurantSimObject.*GUEST\_NUMBER* )  {  **while**( RestaurantSimObject.*expectantGuestAreInRestaurant*() )  {  waitDuration(1000);  }  stopNewGuestComing();   **while**(areOrdersInProgress())  {  waitDuration(1000);  }   RestaurantSimObject.*close*();  **break**;  }  }   waitDuration(20000);  }   **private void** addGuest()  {  RestaurantSimObject.*newGuestCome*(**new** GuestSimObject());  }   **private void** stopNewGuestComing()  {  RestaurantSimObject.*guestsComing*.set(**false**);  }   **private boolean** areOrdersInProgress()  {  **return** OrderQueue.*orders*.size() != 0;  } } |

Zapraszanie gości przez kelnerów:

Klasa InvitingGuestActivity odpowiada za zapraszanie gości do restauracji (z kolejki) przez poszczególnych kelnerów. Dziedziczy po klasie SimActivity z biblioteki DESKIT:

|  |
| --- |
| **public class** InvitingGuestActivity **extends** SimActivity {  **private** WaiterSimObject **waiter**;   **public** InvitingGuestActivity(WaiterSimObject waiterSimObject)  {  **this**.**waiter** = waiterSimObject;  }  @Override  **public void** action()  {  Logger.*log*(() ->**"\n"** + debugMessage());  Logger.*log*(() ->**"[Thread InvitingGuestActivity]: "** + **super**.getName() + **super**.getId());   **while** ( RestaurantSimObject.*isOpened*() )  {  **if** ( expectantGuestAreNotInRestaurant() )  {  takeShortBreak();  }  **else** {  **if** ( **this**.**waiter**.isFree() )  {  GuestSimObject guest = RestaurantSimObject.*expectantGuests*.removeFirst();  invite(guest);}  **else** {  waitDuration(200);  }  }  }  }   **private boolean** expectantGuestAreNotInRestaurant()  {  **return** ! RestaurantSimObject.*expectantGuestAreInRestaurant*();  }   **private void** takeShortBreak()  {  waitDuration(100);  }   **private void** invite(GuestSimObject guest)  {  Logger.*log*(() -> debugMessage() + **"Wzial goscia nr: "** + guest.getId());  guest.setWaiter(**waiter**);   guest.getWaitingTimeInQueueCalculator().setEndTime(guest.getSimTime());   **waiter**.takeGuest();   startVisitOf(guest);  }   **private void** startVisitOf(GuestSimObject guest)  {  *callActivity*(guest, **new** GuestActivity(guest));  }   **private** String debugMessage()  {  **return "[Kelner "** + **waiter**.getId() + **"]:"**;  } } |

Metoda action odpowiada, za zapraszanie gości. Najpierw sprawdza czy restauracja jest otwarta, potem jeśli nie ma gości w kolejce to wątek jest zawieszany, w celu oczekiwania na kolejnych gości. W przeciwnym przypadku jeśli kelner nie jest zajęty to zaprasza gościa do stolika, w przeciwnym wypadku wątek jest zawieszany.

Pobyt gościa w restauracji

Klasa GuestAcitivity to aktywność, która uruchamiana jest w raz z zaproszeniem gościa do restauracji przez kelnera. Uruchamia ona cały szereg innych aktywności, dotyczących pobytu gościa w restauracji. Kolejno uruchamiane aktywności:

1. PlacingOrderActivity (składanie zamówienia),
2. OrderRealizationActivity (realizacja zamówienia przez kelnera),
3. PreparingActivity (przygotowywanie zamówienia przez kucharza),
4. BringingOrderForGuestActivity (zaniesienie zamówienia do gościa przez kelnera),
5. EatingActivity (spożywanie zamówienia),
6. RestActivity (odpoczynek po zjedzeniu),
7. OutingActivity (wychodzenie z restauracji).

Klasa GuestAcitivty za pomocą semafora jest wstrzymywana na czas działania powyższych aktywności.

Klasa GuestVisitAcivity to klasa abstrakcyjna, która stanowi bazę dla klas: EatingActivity, OutingActivity, PlacingOrderActivity, RestActivity. Klasy dziedziczące po GuestVisitActivity odpowiadają za poszczególne aktywności pobytu gościa w restauracji, zostało to opisane w poprzednim akapicie.

Składanie zamówienia

Jak zostało wspomniane wyżej, za składanie zamówienia odpowiada klasa PlacingOrderActivity. Kod klasy:

|  |
| --- |
| **public class** PlacingOrderActivity **extends** GuestVisitActivity {  **public** PlacingOrderActivity(GuestSimObject guest, Semaphore semaphore)  {  **super**(guest, semaphore);  }   @Override  **public void** action()  {  VisualisationLog.*log*(**guest**.getId(), **new** GuestPlacingOrderEvent());  Logger.*log*(() -> **guest**.debugMessage() + **"Składa zamowienie"**);   browseMenu();  chooseMealAndDrink();   Logger.*log*(() -> **guest**.debugMessage() + **"Złożył zamowienie (m:"** +  **guest**.getOrder().getMealsNumber() + **" |d:"** +  **guest**.getOrder().getDrinksNumber() + **")."**);   waitForWaiter();  VisualisationLog.*log*(**guest**.getId(), **new** GuestWaitingForOrder());   Logger.*log*(() -> **guest**.debugMessage() + **"Oczekuje na zamowienie."**);  }   **private void** browseMenu()  {  **int** time = designateWaitingTimeOfPlacingOrder();  waitDuration(time);  }   **private void** chooseMealAndDrink()  {  OrderSimObject order = **new** OrderSimObject();   chooseDrinks(order);  chooseMeals(order);   OrderQueue.*allOrders*.add(order);   **guest**.setOrder(order);  order.setGuest(**guest**);   }   **private void** waitForWaiter()  {  **guest**.getOrder().setStartTime(**guest**.getSimTime());  WaiterSimObject waiter = **guest**.getWaiterSimObject();  **while** (waiter.isBusy())  {  waitDuration(1000);  Logger.*consoleLog*(() -> waiter.debugMessage() + **" zbyt zajety"**);  }  waiter.setBusy(**true**);   waitDuration(*random*.nextInt(1000) \* waiter.getStressRate());  Logger.*consoleLog*(() -> waiter.debugMessage() + **" przyjal zamowienie i zanosi do**  **realizacji"**);  *callActivity*(waiter, **new** OrderRealizationActivity(**guest**.getOrder(), **semaphore**));  }   **private int** designateWaitingTimeOfPlacingOrder()  {  **return** (RestaurantSimObject.*MEALS\_NUMBER* + RestaurantSimObject.*DRINKS\_NUMBER*) \* *random*.nextInt(15);  }   **private void** chooseDrinks(OrderSimObject order)  {  **int** numberOfDrinksToPrepare = *random*.nextInt(6) + 1;   **for**(**int** i = 0; i < numberOfDrinksToPrepare; i++)  {  **int** drinkId = *random*.nextInt(RestaurantSimObject.*DRINKS\_NUMBER*);  order.addDrink(Menu.*getDrink*(drinkId));  }  }   **private void** chooseMeals(OrderSimObject order)  {  **int** numberOfMealsToPrepare = *random*.nextInt(6) + 1;  **for**(**int** i = 0; i < numberOfMealsToPrepare; i++)  {  **int** mealId = *random*.nextInt(RestaurantSimObject.*MEALS\_NUMBER*);  order.addMeal(Menu.*getMeal*(mealId));  }  } } |

Zamawianie działa w ten sposób, że gość najpierw wybiera co najmniej jeden posiłek i co najmniej jeden napój. Symulowane jest to w ten sposób, że wątek wstrzymywany jest na kilka milisekund - czas ten może być dłuższy im więcej jest dostępnych posiłków i napojów. Następnie gość oczekuje na przybycie kelnera. Kelner jeśli, nie jest zajęty np. zbieraniem zamówienia od innego gościa, bądź jest w trakcie przynoszenia zamówienia do innego gościa, to umieszcza zamówienie w kolejce realizacji. Dalej zmówieniem zajmuje się kucharz.

Realizacja zmówienia

Realizacja zmówienia rozpoczyna się od klasy OrderRealizationActivity:

|  |
| --- |
| **public class** OrderRealizationActivity **extends** SimActivity {  **private** OrderSimObject **order**;  **private** Semaphore **semaphore**;   **private** WaiterSimObject **waiter**;   **public** OrderRealizationActivity(OrderSimObject order, Semaphore semaphore)  {  **this**.**order** = order;  **this**.**semaphore** = semaphore;  }    @Override  **public void** action()  {  Semaphore semaphore = **new** Semaphore(-1);  OrderNote orderNote = **new** OrderNote(**order**, semaphore);   OrderQueue.*addOrder*(orderNote);  Logger.*log*(() -> **order**.debugMessage() + **"Przekazane do kolejki realizacji"**);   **waiter** = **order**.getGuestSimObject().getWaiterSimObject();  **waiter**.setBusy(**false**);   semaphore.wait(**this**);  waitOnSemaphore(semaphore);  Logger.*log*(() -> **order**.debugMessage() + **"Gotowe"**);   bringToGuest();  Logger.*log*( () -> **order**.debugMessage() + **"Zamowienie zrealizowane"** );  }   **private void** bringToGuest()  {  **while** (**waiter**.isBusy())  {  waitDuration(1000);  Logger.*consoleLog*(() -> **waiter**.debugMessage() + **"zbyt zajety"**);  }**waiter**.setBusy(**true**);   *callActivity*(**waiter**, **new** BringingOrderForGuestActivity(**order**, **this**.**semaphore**));  } } |

W pierwszej kolejności zamówienie odkładane jest do kolejki zamówień. Następnie przy pomocy semafora wątek jest wstrzymywany do czasu przygotowania zamówienia przez kucharza. W momencie kiedy wątek odpowiedzialny za przygotowanie zamówienia zwolni semafor to kelner może zanieść zamówienie do danego gościa. Podobnie jak przy składaniu zamówienia przez gościa, kelner może być zajęty zbieraniem zamówień lub zanoszeniem innego zamówienia.

Klasa OrderQueue zawiera kolejkę zamówień składanych przez klienta. Zamówienia pobierane są z tej kolejki przez kucharzy w celu ich realizacji. Za pobieranie kolejnego zamówienia odpowiada klasa PreparingOrderActivity:

|  |
| --- |
| **public class** PreparingOrderActivity **extends** SimActivity {  **private** CookSimObject **cook**;  **private** OrderNote **orderNote**;   **public** PreparingOrderActivity(CookSimObject cook, OrderNote orderNote)  {  **this**.**cook** = cook;  **this**.**orderNote** = orderNote;  }   **public** PreparingOrderActivity(OrderNote orderNote)  {  **this**.**orderNote** = orderNote;  }  @Override  **public void** action()  {  Logger.*log*(() -> **cook**.debugMessage() + **orderNote**.getOrder().debugMessage() + **"W fazie**  **przygotowania"**);  **double** preparingTime = **orderNote**.getOrder().getPreparingTime()\***cook**.getStressRate();  waitDuration(preparingTime);  Logger.*log*(() -> **cook**.debugMessage() + **orderNote**.getOrder().debugMessage() +  **"Przygotowywanie skonczone."**);  **cook**.getCurrentOrderNumber().decrementAndGet();  **orderNote**.getOrder().**done**.set(**true**);  **orderNote**.getSemaphore().signal();  }  } |

Najpierw wyznaczany jest czas przewidziany na przygotowywanie zamówienia, następnie wątek jest wstrzymywany co symuluje przygotowywanie zamówienia.

Za pobieranie przez kucharzy zamówień, przeznaczonych do realizacji, odpowiada klasa TakingOrderActivity.

Klasa BringingOrderForGuestActivity realizuje aktywność przynoszenia zamówienia do gościa. Uruchamia ona aktywność gościa odpowiadającą za spożywanie zamówienia.