Paradygmaty Programowania

Marcin Bączyk

Wykład 6

29 marca 2021

Treść dzisiejszego wykładu

- test
- solid
- diagramy sekwencji

Literatura

- Robert C. Martin: “Czysta architektura. Struktura i design oprogramowania. Przewodnik dla profesjonalistów”
- Robert C. Martin: “Zwinnej wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki”
- Grady Booch, James Rumbaugh, Ivar Jacobson: “UML przewodnik użytkownika”
- Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides: “Wzorce projektowe”

SOLID

- **S**ingle responsibility principle
  - Zasada jednej odpowiedzialności
- **O**pen/closed principle
  - Zasada otwarte-zamknięte
- **L**iskov substitution principle
  - Zasada podstawienia Liskowej
- **I**nterface segregation principle
  - Zasada segregacji interfejsów
- **D**ependency inversion principle
  - Zasada odwrócenia zależności

Zasada jednej odpowiedzialności

Powinien istnieć tylko jeden powód do modyfikacji klasy.
(ang. *A class should have only one reason to change.*)

- Odpowiedzialność rozumiana jest jako rodzina funkcji (metod), która służy **jednemu konkretnemu** aktorowi.
- Aktor odpowiedzialności jest jedynym źródłem zmiany tej odpowiedzialności.
- Odpowiedzialność w kontekście zasady jednej odpowiedzialności definiowana jest jako **powód do zmiany**.

Zbierz rzeczy, które zmieniają się z tych samych powodów. Oddziel te rzeczy, które zmieniają się z różnych powodów.

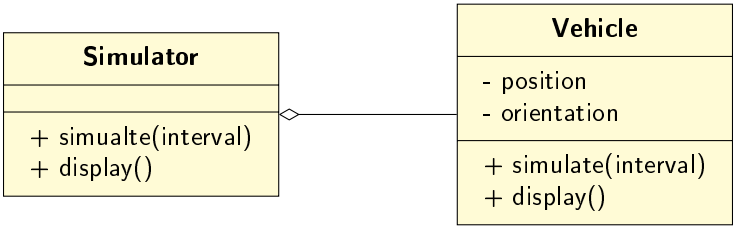
Zasada jednej odpowiedzialności

Zadanie

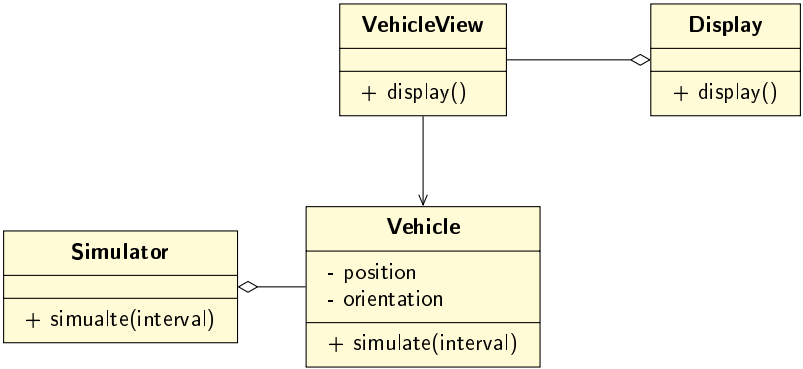
Zaprojektuj fragment symulatora ruchu ulicznego z wizualizacją aktualnego stanu symulacji.

Rozwiązanie (studenckie)

Projekt najczęściej przewiduje jedną klasę zbiorczą – symulator – oraz jedną klasę dla symulowanych pojazdów.



Zasada jednej odpowiedzialności



Zasada jednej odpowiedzialności

- Z obiektów klasy **Vehicle** korzystają de facto dwie różne aplikacje: moduł symulatora i moduł wyświetlający aktualny stan.
- Moduł symulacji nigdy nie korzysta z operacji wyświetlania.
- Moduł wyświetlający nigdy nie korzysta z operacji symulacji.
- Pierwotnie klasa Vehicle miała dwie osobne odpowiedzialności - naruszenie zasady SRP.
- Rozdzielenie klasy reprezentującej pojazd na dwa typy pozwala na znacząco mniejsze sprzęganie się dwóch osobnych aspektów aplikacji.

Zasada otwarte-zamknięte

Encje oprogramowania(klasy, moduły, funkcje itp.) powinny być otwarte na rozbudowę, ale zamknięte dla modyfikacji.
(ang. *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*)

- Klasa uważana jest za zamkniętą gdy może zostać skompilowana i dostarczana do użytkownika w postaci binarnej w ramach biblioteki.
- Klasa pozostaje otwarta jeżeli nowe klasy mogą wykorzystać ją jako klasę bazową, dodając nowe właściwości.

Stosuj abstrakcyjne klasy bazowe.

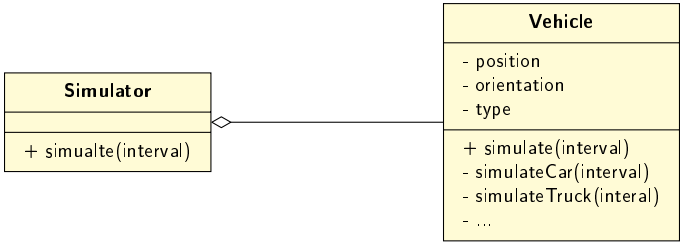
Zasada otwarte-zamknięte

Zadanie

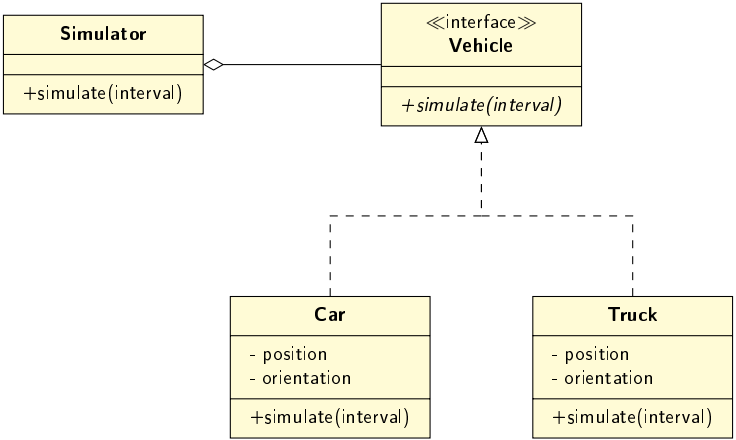
W systemie istnieją różne typy pojazdów i w przyszłości baza typów może być rozszerzana.

Rozwiązanie (studenckie)

Klasa **Vehicle** zawiera informację jakiego typu jest pojazdem a sposób symulacji zależy od ustawionej wartości.



Zasada otwarte-zamknięte



Zasada otwarte-zamknięte

- Obiekt klasy **Simulator** używa konkretnych pojazdów poprzez wyabstrahowany typ **Vehicle**.
- Symulator nic "nie wie" o konkretnych implementacjach pojazdów – nie jest od nich zależny.
- Konkretne klasy **Car** oraz **Truck** nie zależą od siebie, a jedynie od wspólnego interfejsu w postaci klasy **Vehicle**.
- Aby dodać nowy typ pojazdu do symulacji wystarczy dodać nową klasę implementującą interfejs.
- Wydzielenie wspólnego interfejsu i implementacja zachowani różnych pojazdów w osobnych klasach pozwala na dalsze zmniejszenie sprzęgania się poszczególnych modułów aplikacji.

Zasada podstawienia Liskowej

Typy pochodne muszą móc być podstawione za ich typy bazowe.
(ang. *"Subtypes must be substitutable for their base types."*)

- Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów

Obiekty klas potomnych zachowują się jak obiekty klas bazowych.

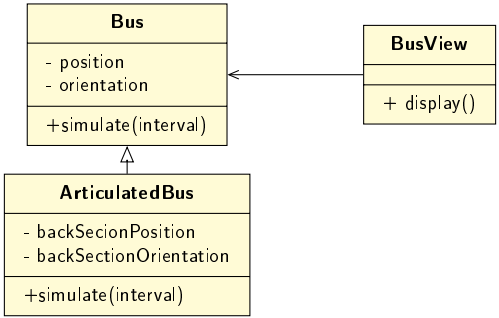
Zasada podstawienia Liskowej

Zadanie

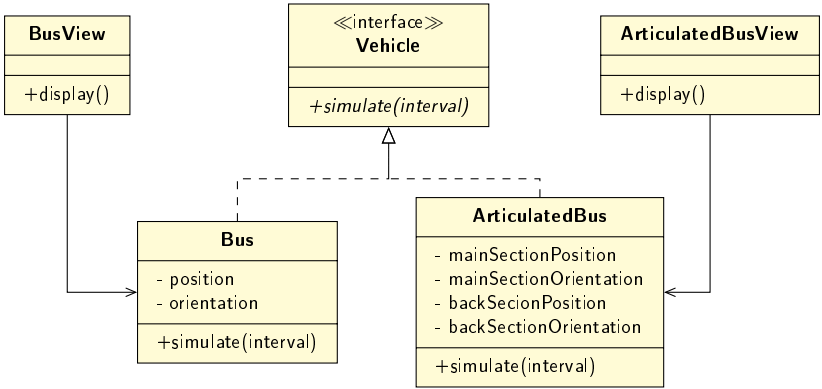
W symulatorze należy uwzględnić specjalny rodzaj autobusów jakim są autobusy przegubowe.

Rozwiązanie (studenckie)

Dodanie nowej klasy **ArticulatedBus** dziedziczącej po klasie **Bus**.



Zasada podstawienia Liskowej



Zasada podstawienia Liskowej

- Obiekt typu **BusView** nie potrafią w sposób prawidłowy wyświetlić obiektów typu **ArticulatedBus**.
- Bezpośrednie dziedziczenie klasy **ArticulatedBus** po klasie **Bus** powoduje, że możliwe jest przekazanie referencji do autobusu przegubowego jako argumentu obiektu rysującego zwykły autobus.

Zasada segregacji interfejsów

Klasy klientów nie powinny być zmuszone do zależności od metod, których nie używają.
(ang. *"No client should be forced to depend on methods it does not use."*)

- Klasy, które charakteryzują się grubymi" interfejsami, to klasy których interfejsy nie są spójne. Wydzielone grupy metod obsługują rożne zbiory klientów.
- Interfejsy powinny być małe, żeby później klasy nie implementowały metod, których nie potrzebują.

Unikaj sytuacji, w której moduł klienta musi wiedzieć więcej niż potrzebuje do wykonania swojej zadania (odpowiedzialności).

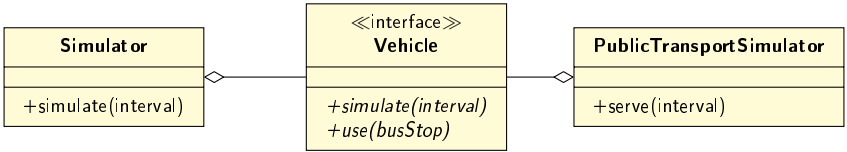
Zasada segregacji interfejsów

Zadanie

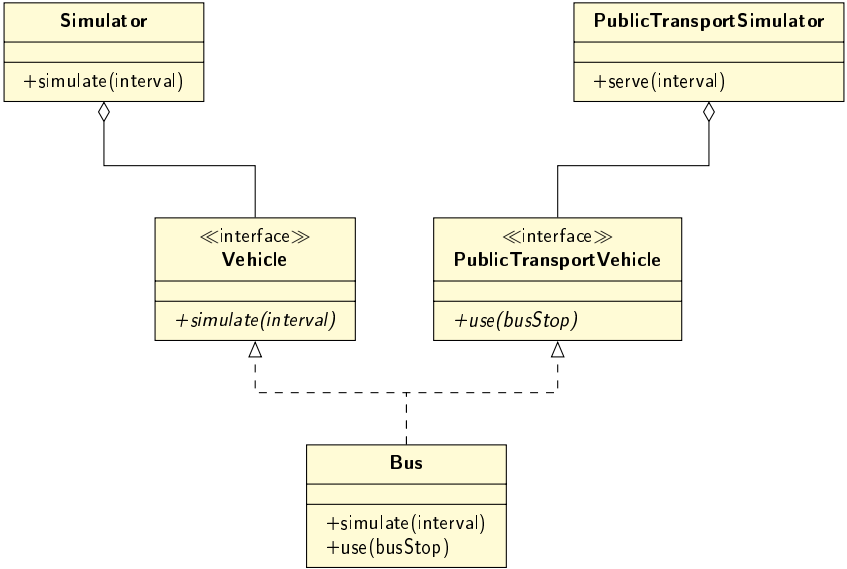
W symulatorze zależy dodać nowy moduł symulujący obciążenie transportu publicznego.

Rozwiązanie (studenckie)

Dodanie nowej klasy **PublicTransportSimulator** oraz poszerzenie interfejsu klasy **Vehicle**.



Zasada segregacji interfejsów



Zasada segregacji interfejsów

- Klasa symulująca ruch uliczny nie musi nic wiedzieć o obsłudze transportu publicznego.
- Różne obiekty pojazdów, niezwiązanych z transportem publicznym nie muszą implementować metod związanych z obsługą przystanków.
- Obiekty pojazdów transportu publicznego, w czasie ruchu mogą być obsługiwane wraz z innymi obiektami uczestniczącymi w zwykłym ruchu drogowym.

Zasada odwrócenia zależności

Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. I jedne, i drugie powinny zależeć od abstrakcji.
(ang. *"High-level modules should not depend on low-level modules. Both should depend on abstractions."*)
Abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji
(ang. *"Abstractions should not depend upon details. Details should depend upon abstractions."*)

Projektuj jasno określone warstwy w architekturze systemu.
Używaj interfejsów i klas abstrakcyjnych wszędzie tam, gdzie wydaje się, że może to być użyteczne w przyszłości.

Zasada odwrócenia zależności

Zadanie

W symulatorze zależy dodać mechanizm zapisu stanu symulacji, np. w celu późniejszej prezentacji w przyspieszonym czasie.

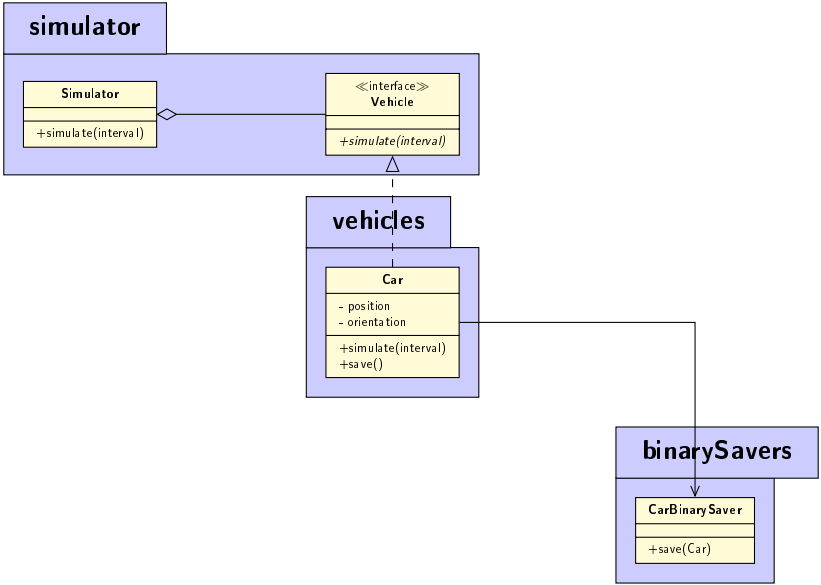
Rozwiązanie (studenckie)

Dodanie nowej metody zapisującej stan do każdej z klas implementującej interfejs **Vehicle**.

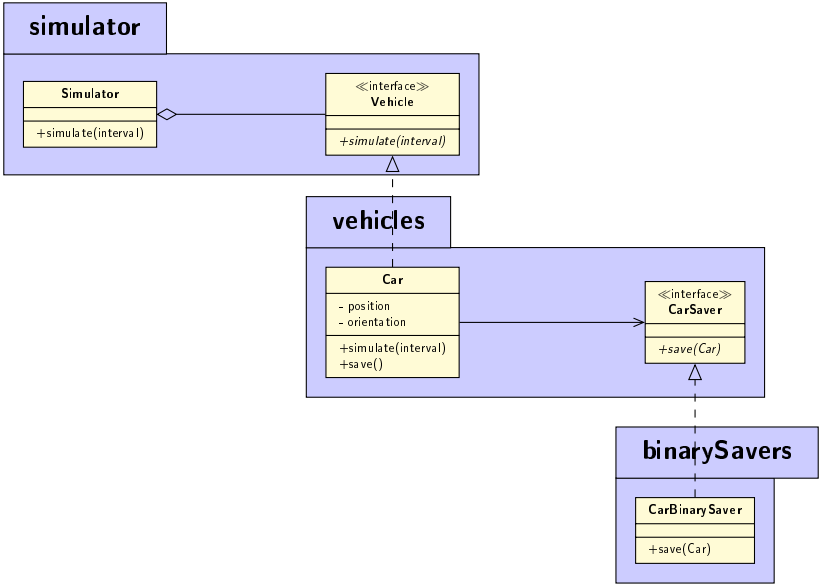
Rozwiązanie (trochę studenckie)

Dodanie nowej klasy zapisującej stan dla każdej z klas implementującej interfejs **Vehicle**.

Zasada odwrócenia zależności



Zasada odwrócenia zależności



Zasada odwrócenia zależności

- Moduły wyższego rzędu nie zależą od modułów niższego rzędu, dzięki wprowadzeniu dodatkowej abstrakcji.
- Dodawanie nowych mechanizmów utrwalania obiektów (np. w postaci klatek filmu) jest mozliwe poprzed dodanie nowego modułu, zgodnie z zasadą otwarte - zamknięte.
- Odwrócenie zależności sprzyja powstawaniu obiektowej struktury oprogramowania.

Zależność od abstrakcji (podejście naiwne)

- Zmienne nie zawierają referencji do obiektów klas konkrentych.
- Klasy nie są klasami pochodnymi klas konkretnych.
- Metody nie przesłaniają metod implementowanych w klasach bazowych.

Diagram sekwencji

Diagram klas

- Jest grafem złożonym z wierzchołków w postaci klas i interfejsów oraz krawędzi w postaci relacji między poszczególnymi wierzchołkami.
- Silnie reprezentuje strukturę systemu.
- Służy do zobrazowania statycznych aspektów projektowanego systemu.

Diagram sekwencji

- Służy do prezentowania interakcji pomiędzy obiektami wraz z uwzględnieniem w czasie komunikatów, jakie są przesyłane pomiędzy nimi.
- Pozwala uzyskać odpowiedź na pytanie, jak w czasie przebiega komunikacja pomiędzy obiektami.
- Stanowi technikę modelowania zachowania systemu.

Diagram sekwencji

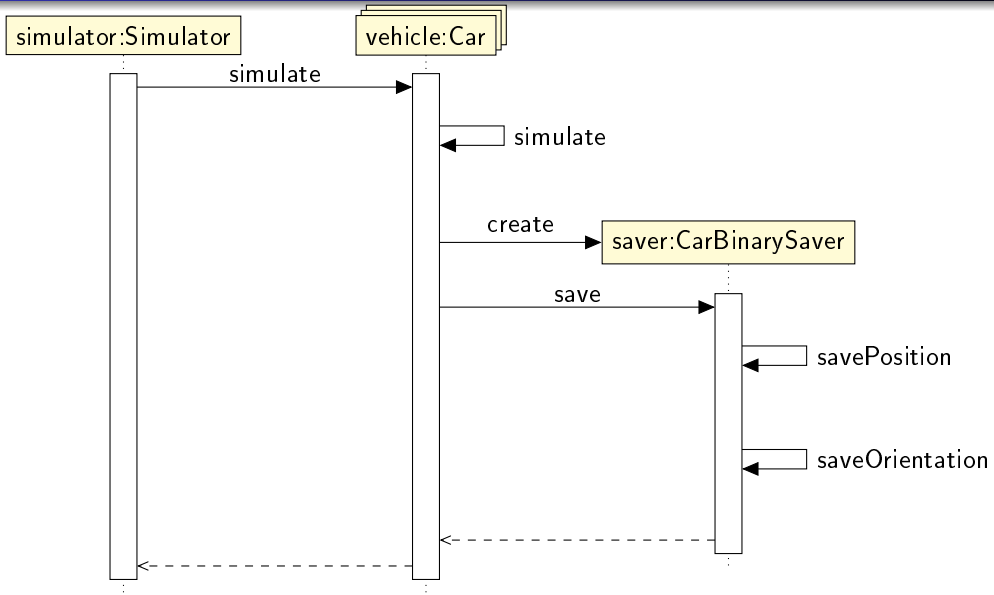


Diagram sekwencji

Linia życia

Linie życia reprezentują konkretne byty – obiekty lub systemy. Linia życia to rola uczestnika interakcji, jaką pełni w czasie jej trwania. Reprezentuje współuczestnika interakcji i czas jego istnienia podczas realizacji scenariusza.

Linie życia mogą przyjmować różne stereotypy. Na przykład stereotyp aktora informuje, że obiekt pełni funkcję zewnętrzną w stosunku do modelowanego systemu

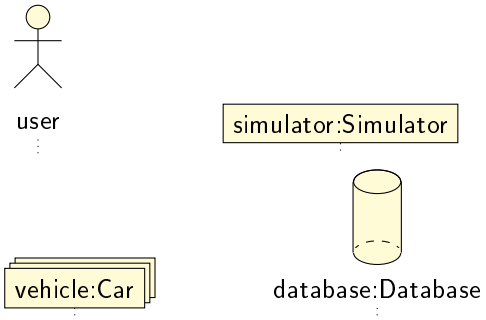


Diagram sekwencji

Komunikat

Kominukat jest informacją przesyłaną pomiędzy obiektami.

Komunikat synchroniczny oznacza, że obiekt go wysyłający oczekuje na odpowiedź.

Komunikak asynchroniczny nie wymaga oczekiwania na odpowiedź.

Komunikat może być wysłany do tego samego obiektu, który go wysyła.

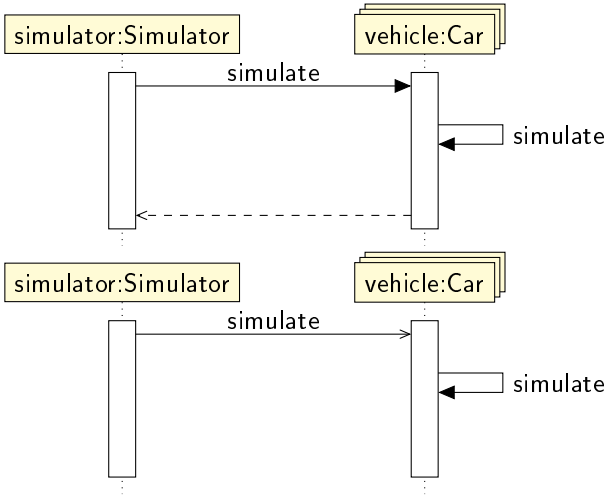


Diagram sekwencji

Komunikat

Specjalnym rodzajem komunikatu jest utworzenie nowego obiektu.

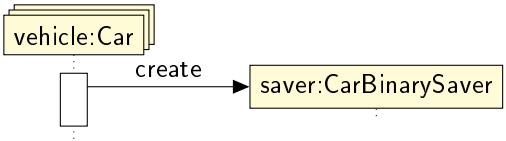


Diagram sekwencji

Komunikat

Poszczególne komunikaty są oddzielone graficznie od siebie, jeżeli nie stanowią większej całości.

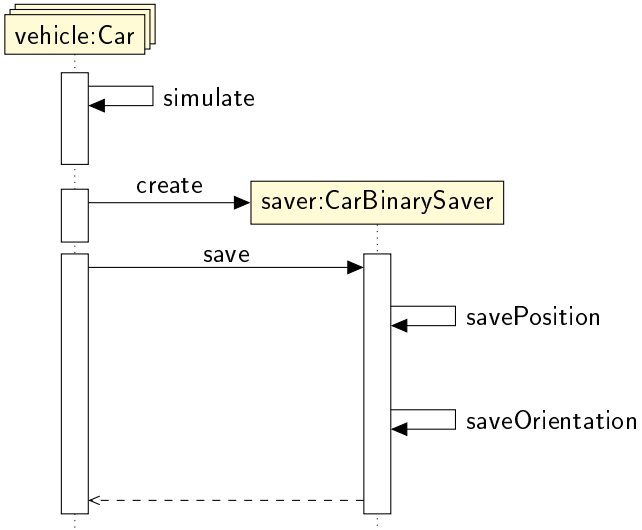
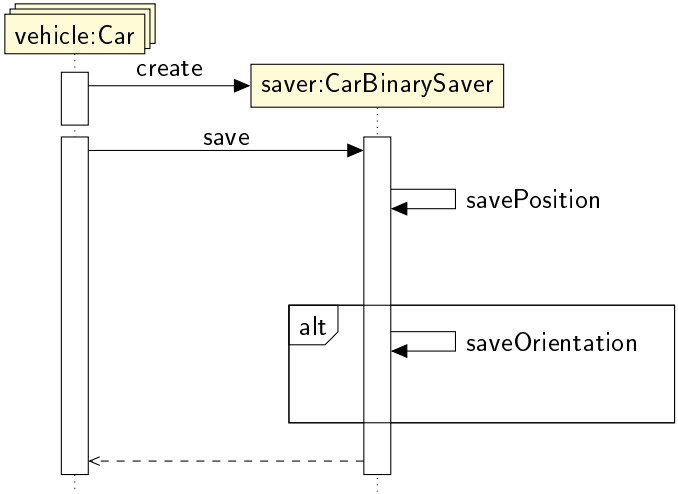


Diagram sekwencji

Fragment - alt

Dzieli fragment interakcji na dwa alternatywne scenariusze.



Fragment - loop

Powtórzenie fragmentu interakcji określoną warunkiem liczbę razy.

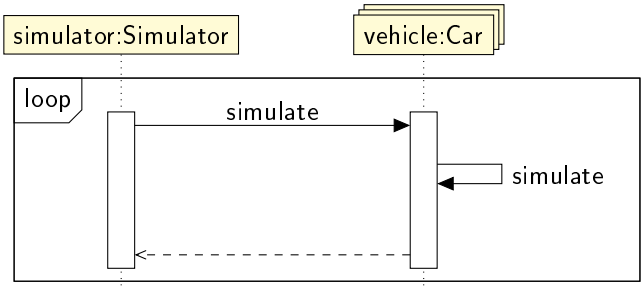


Diagram sekwencji

Fragment - par

Prezentuje równoległe wykonywanie przepływu wiadomości.