Observer

Linda Marshall

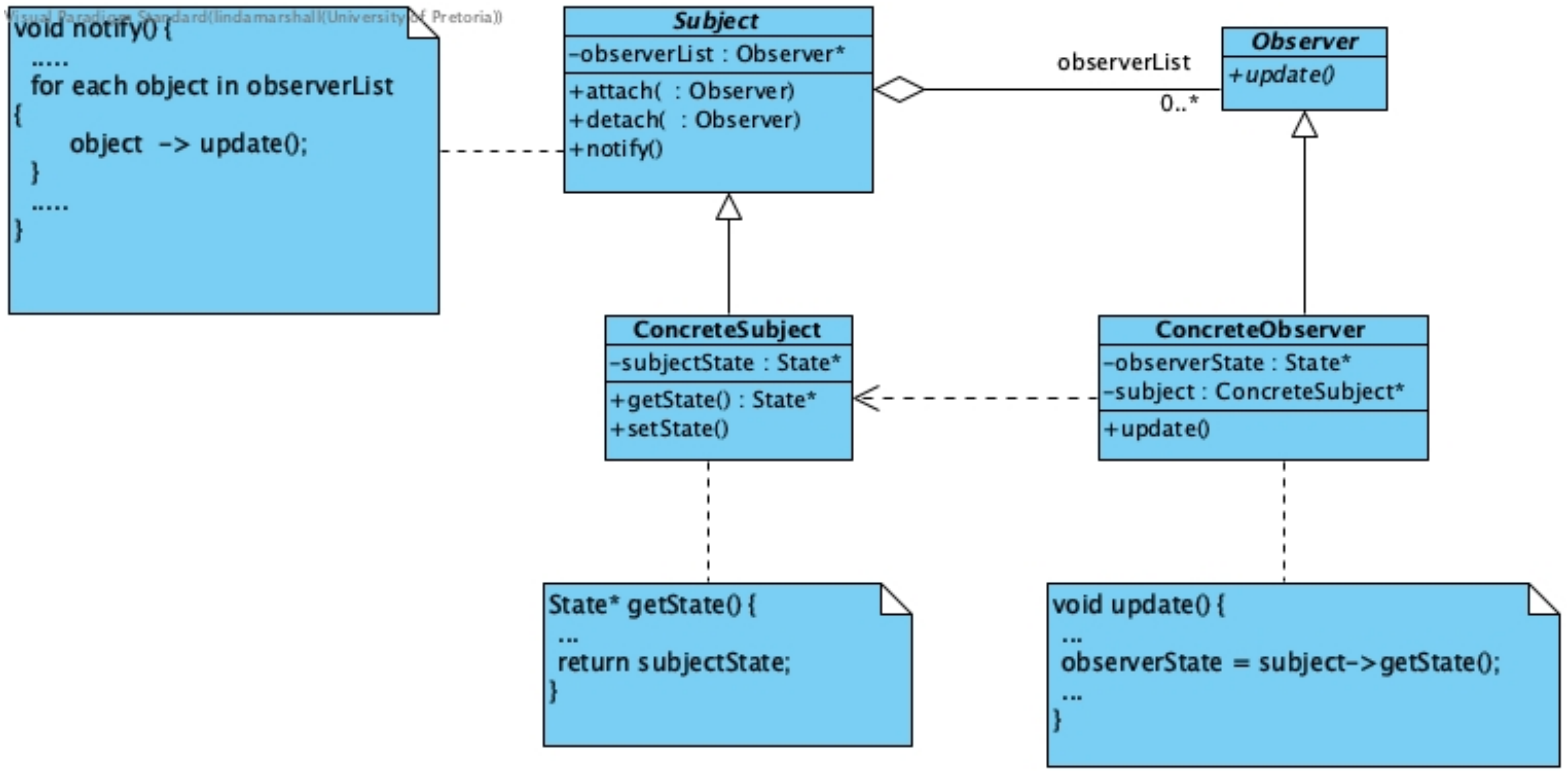
Department of Computer Science
University of Pretoria

21 September 2021

Name and Classification: Observer (Object Behavioural)

Intent: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" GoF(293)

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" GoF(293)


void notify() {
.....
for each object in observerList
{
object -> update();
}
.....
}
Subject
-observerList : Observer*
+attach(: Observer)
+detach(: Observer)
+notify()
observerList
0..*
Observer
+update()
ConcreteSubject
-subjectState : State*
+getState() : State*
+setState()
ConcreteObserver
-observerState : State*
-subject : ConcreteSubject*
+update()
State* getState() {
...
return subjectState;
}
void update() {
...
observerState = subject->getState();
...
}

Comprises of two hierarchies:

- *Subject hierarchy* - the objects that are to be observed
- *Observer hierarchy* - the objects that are to do the observation

Subject

- Provides an interface for observers to attach and detach to the concrete subject.

ConcreteSubject

- Implementation of the subject being observed.
- Implements the functionality to store objects that are observing it and sends update notifications to these objects.

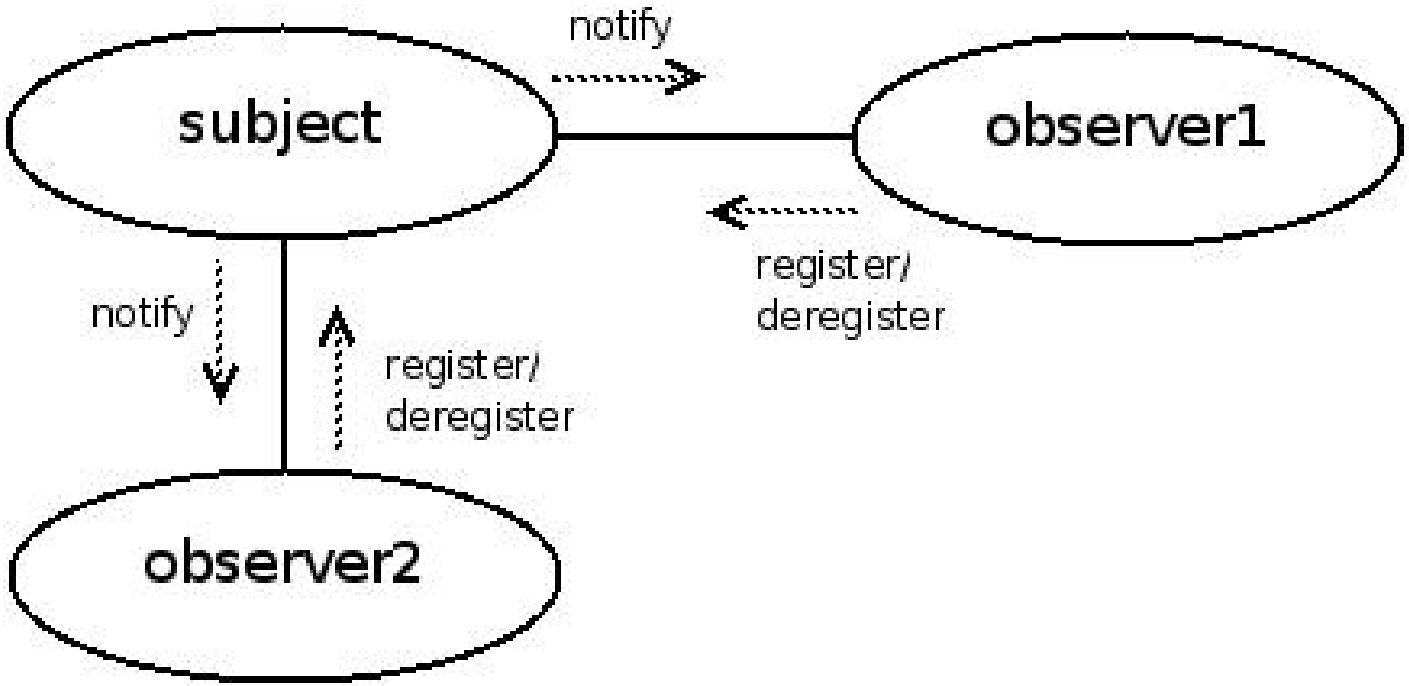
Observer

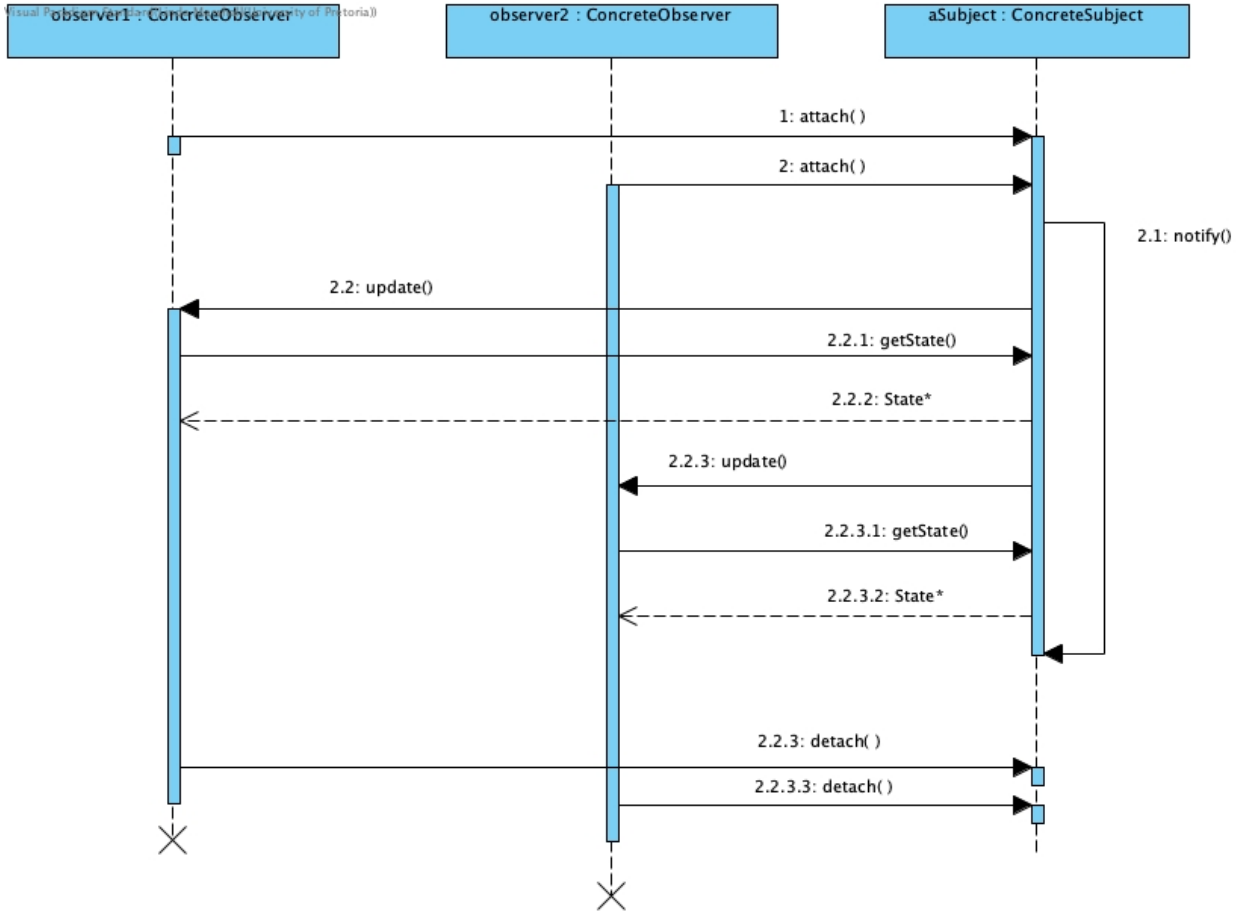
- Defines the interface of objects that may observe the subject.
- Provides the means by which the observers are notified regarding change to the subject.

ConcreteObserver

- Maintains a reference to the subject it observes.
- Updates and stores relevant state information of the subject in order to keep consistent with the state of the subject.

- Enables the attaching and detaching of observers from the code and leaving the subject intact.
- Observers of a subject register with the subject and will be notified when a change occurs in the subject.

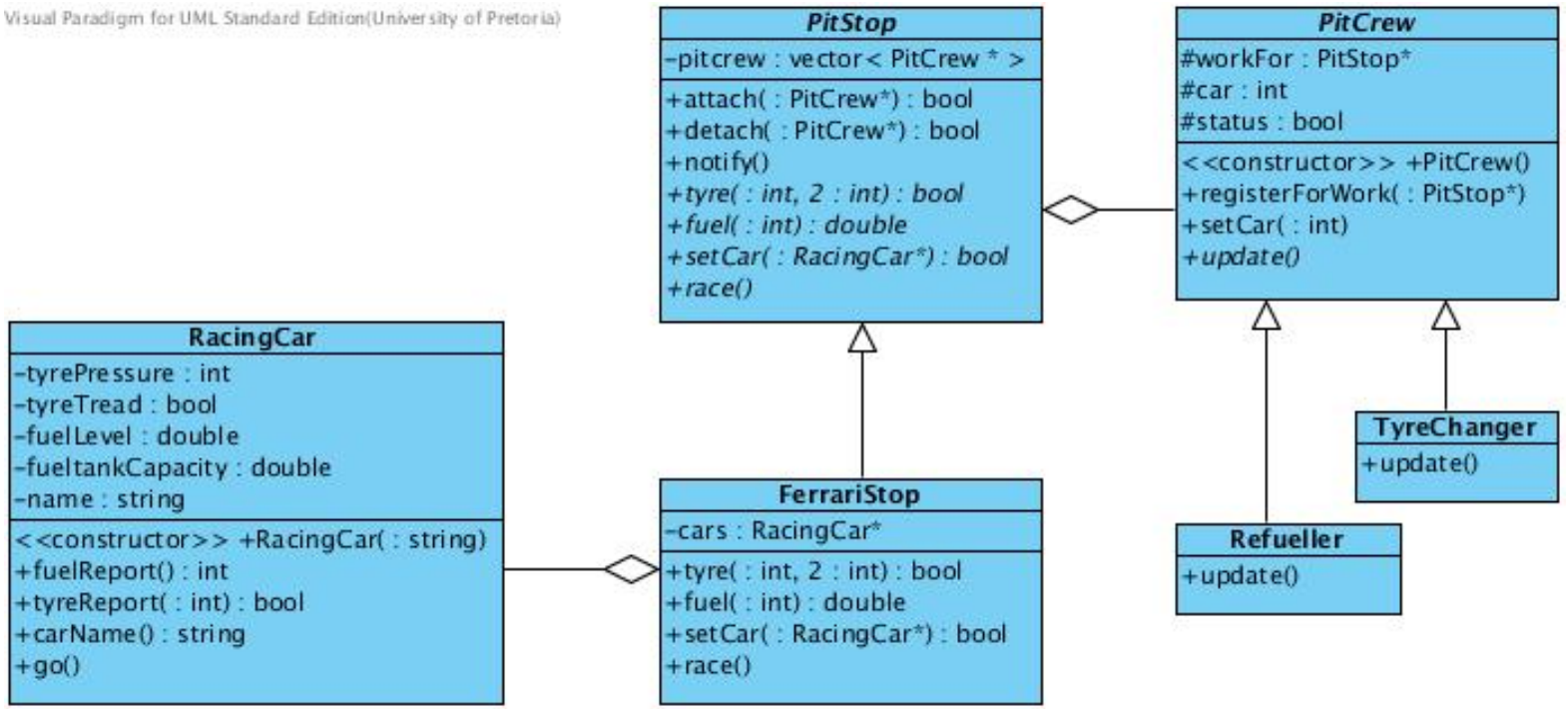

notify
subject
observer1
register/
deregister
notify
register/
deregister
observer2


observer1 : ConcreteObserver
observer2 : ConcreteObserver
aSubject : ConcreteSubject
1: attach( )
2: attach( )
2.1: notify()
2.2: update()
2.2.1: getState()
2.2.2: State*
2.2.3: update()
2.2.3.1: getState()
2.2.3.2: State*
2.2.3: detach( )
2.2.3.3: detach( )

- *Detaching the observer from the subject when is goes out of scope*
- *Transferring state to the observer*

- **Mediator** The mediator defines how objects interact and thereby promotes loose coupling between the objects. The objects can therefore interact independently. Mediators are often used to ensure the independent transfer of state between the subject and its observers.
- **Singleton** By making the subject a singleton, it ensures that the subject has only one access point to it.

Consider the F1 Grand Prix, or any other motor racing scenario where a pit crew needs to make decisions for refuelling and changing the tyres of the car that is racing throughout the race. The car in this case is the subject and the pit crew are the observers.



Note: This diagram is incomplete.

The PitStop hierarchy represents the subject and the PitCrew hierarchy the observers with two concrete observers, the Refueller and the TyreChanger.

```
bool PitStop::attach(PitCrew* person) {
  pitcrew.push_back(person);
  person->registerForWork(this);
  return true;
};
bool PitStop::detach(PitCrew* person) {
  bool found = false;
  vector<PitCrew*>::iterator it = pitcrew.begin();
  while ((it != pitcrew.end()) && (! found)) {
    if (*it == person) {
      found = true;
      pitcrew.erase(it);
    }
    ++it;
  }
  return found;
}
void PitStop::notify(){
  vector<PitCrew*>::iterator it = pitcrew.begin();
  for (it = pitcrew.begin(); it != pitcrew.end(); ++it){
    (*it)->update();
  }
}
```

```
bool FerrariStop::tyre(int car, int tyre) {
  return cars[car−1]−>tyreReport(tyre−1);
}
double FerrariStop::fuel(int car) {
  return cars[car−1]−>fuelReport();
}
bool FerrariStop::setCar(RacingCar* car) {
  static int carId = 0;
  if (carId < 2) {
    cars[carId] = car;
    carId++;
    return true;
  }
  return false;
}
```

```
void FerrariStop::race() {
  int input;
  cout << "Type in a number [0 stops]";
  cin >> input;
  while (input != 0) {
    if ((input % 2) == 0) {
      cars[0]->go();
    } else {
      cars[1]->go();
    }
    printWorkshopStatus(this);
    notify();
    cout << "Type in a number [0 stops]";
    cin >> input;
  }
}
```

```
void PitCrew::registerForWork(PitStop* employer) {
  workFor = employer;
}

void PitCrew::setCar(int c) {
  car = c;
}
```

```
void TyreChanger::update() {
  if (status == 0) {
    cout << "Check tyre status" << endl;
    bool tyreStatus = false;
    for (int i = 1; i <= 4; i++)
      tyreStatus = tyreStatus && workFor->tyre(car,i);
    if (tyreStatus) {
      status = 1;
      cout << "Need to change all tyres" << endl;
    }
  } else
    status = 0;
}
```

```
void Refueller::update() {
  cout << "Refeuller for car " << car << " status is " << status << endl;
  if (status == 0) {
    cout << "Check fuel status" << endl;
    double fuelStatus = workFor->fuel(car);
    cout << "  fuel status is: " << fuelStatus << endl;
    if (fuelStatus < 20) {
      status = 1;
      cout << "Need to add fuel" << endl;
    }
  } else
    status = 0;
}
```

```
RacingCar::RacingCar(string n) : name(n) {
  for (int i = 0; i < 4; i++) {
    tyrePressure[i] = 4;
    tyreTread[i] = true;
  }
  fueltankCapacity = 100;
  fuelLevel = 100;
}

int RacingCar::fuelReport() {
  return fuelLevel / fueltankCapacity * 100;
}

bool RacingCar::tyreReport(int tyre) {
  return tyrePressure[tyre] && tyreTread[tyre];
}

string RacingCar::carName() {
  return name;
}
```

```
void RacingCar::go() {
  int input;
  cout << "Type in any value: " << endl;
  cin >> input;
  if ((input % 2) == 0) {
    // Do the tyres
    if ((input % 3) == 0) {
      tyreTread[input%4] = false;
    } else {
      tyrePressure[input%4] = false;
    }
  } else {
    // Do the fuel
    fuelLevel -= 5;
  }

}
```

```
void printWorkshopStatus(PitStop* p) {
  cout << "Fuel for car 1 = " << p->fuel(1) << endl;
  cout << "Fuel for car 2 = " << p->fuel(2) << endl;

  for (int i = 1; i <= 4; i++) {
    cout << "Tyre for car 1, tyre " << i << " = " << p->tyre(1,i) << endl;
    cout << "Tyre for car 2, tyre " << i << " = " << p->tyre(2,i) << endl;
  }

}
```