Behavior of objects

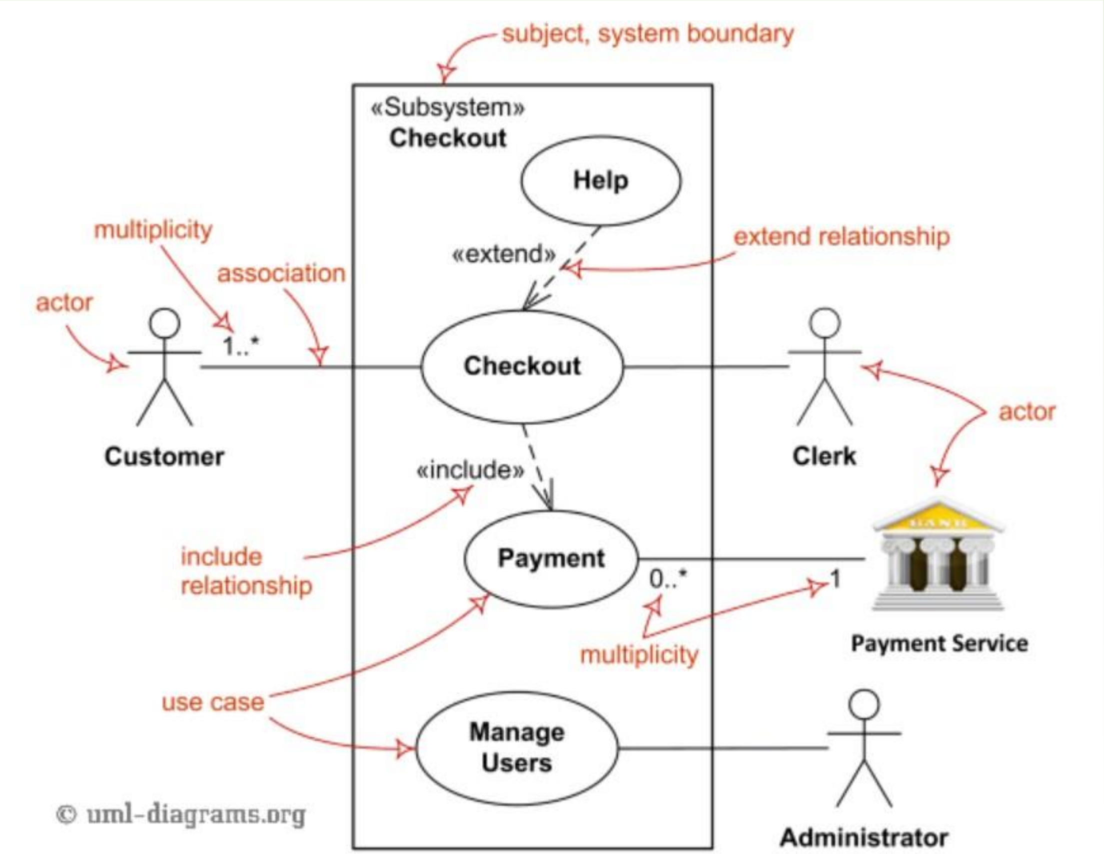
Behavioral views of UML

Behavioral views

- To describe the dynamical behavior of objects, UML has introduced several views. In this lecture, the followings are presented:
  - Use case diagram
  - Communication diagram
  - Sequence diagram
  - State machine diagram

Use case diagram

- For a planned system, it shows
  - its goal,
  - its functionality (what it is capable of),
  - who it serves (actors)
  - what requirements it has for the environment



Specificators of the relationships of the use cases

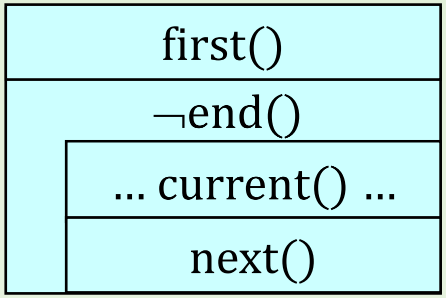
- Precedence of the use cases
  - precede: the order of the activities a user might trigger
  - invoke: an activity which follows a user activity, but cannot be triggered directly
- Extensions of a use case
  - include: independently triggerable, divided part of a user activity without which the container is incomplete (abstract).
  - extend: a complete activity which might extend optionally a user activity and which is never abstract
- Inheritance between activities or actors
- Multiplicity might be denoted

User story

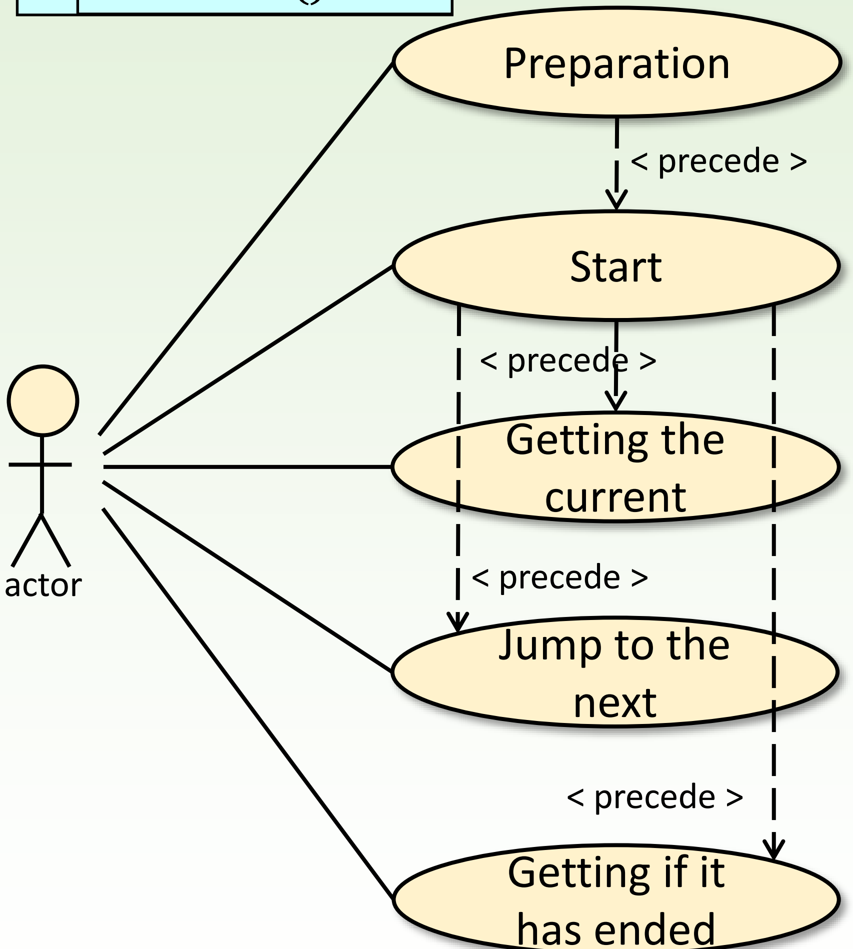
- A use case diagram does not provide an acceptable image of the system to be implemented.
- In a tabular description (called user story) which goes by user groups ("AS a …"), every user activity has to be explained in detail:
  - name of the activity,
  - what prerequisites it assumes (GIVEN)
  - what event triggers it (WHEN)
  - its effects and result (THEN).

AS a …

case		description
activity	GIVEN	assumed precondition
	WHEN	triggering event
	THEN	effects



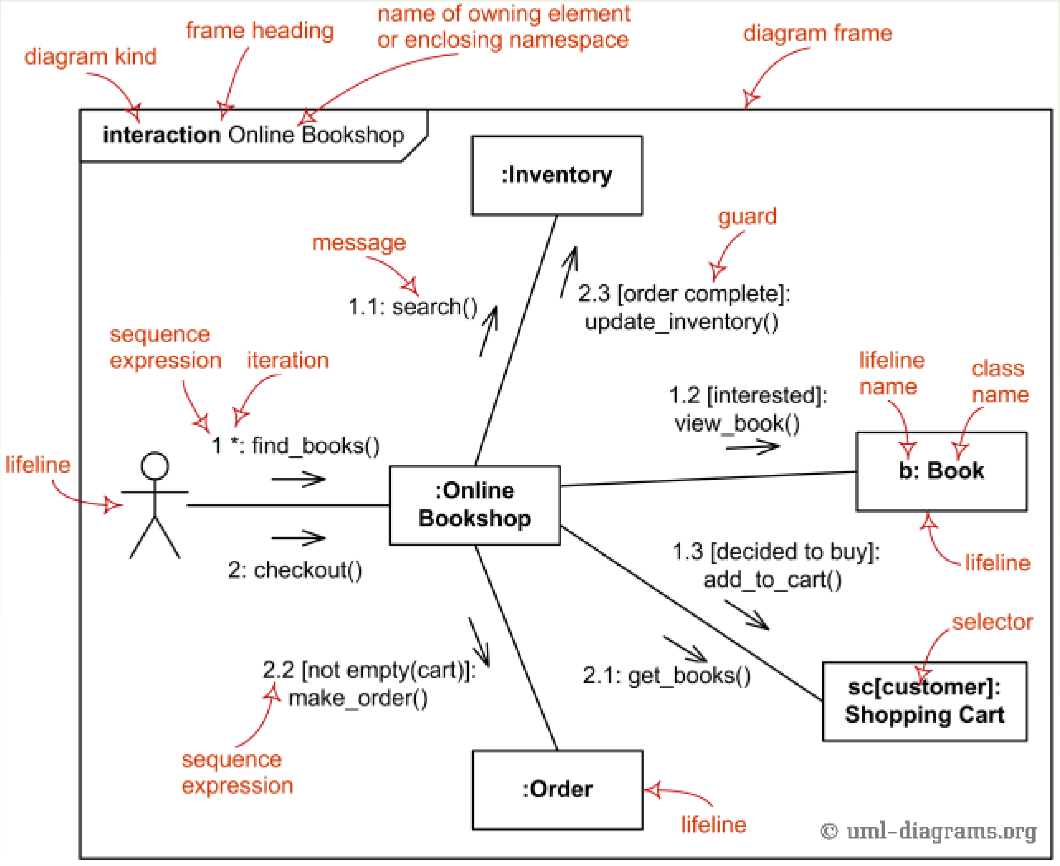
Example: Enumeration



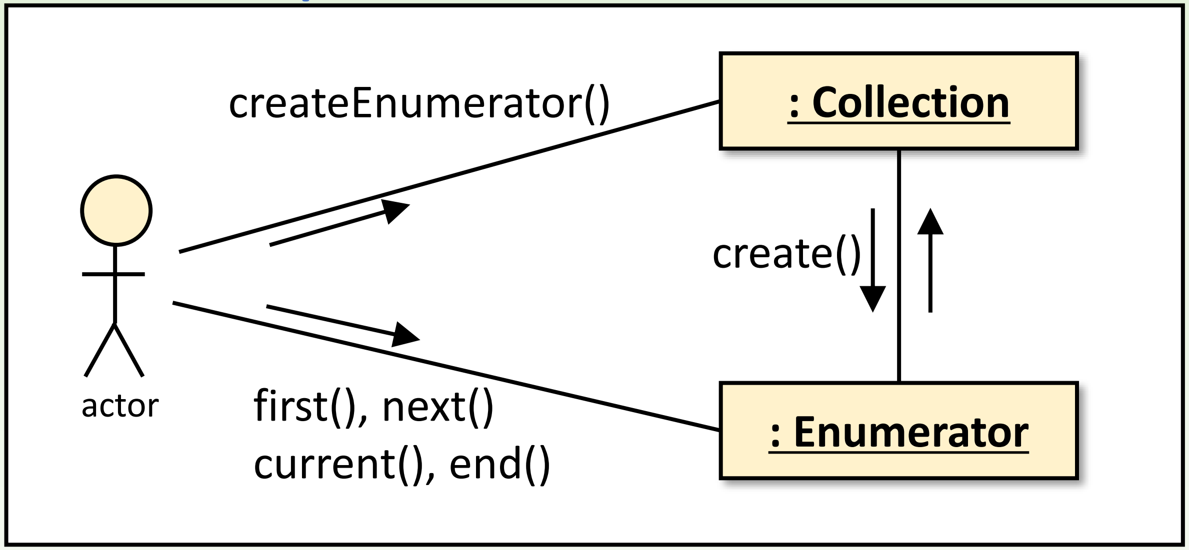
case		description
Preparation in normal case	GIVEN	Given a collection to be enumerated.
	WHEN	Instantiation of the enumerator.
	THEN	Enumerator is created.
Preparation in abnormal case	GIVEN	There is no collection to be enumerated.
	WHEN	Instantiation of the enumerator.
	THEN	Error, enumerator object is not created.
Start in normal case	GIVEN	Given an enumerator object in state *pre-start*.
	WHEN	Starting the enumeration with operation **first()**.
	THEN	The enumerator gets to state *in-process*.
Start in abnormal case	GIVEN	Given an enumerator object in state *in-process* or *finished*.
	WHEN	Starting the enumeration with operation **first()**.
	THEN	Error, and the enumerator preserves its state.
...		

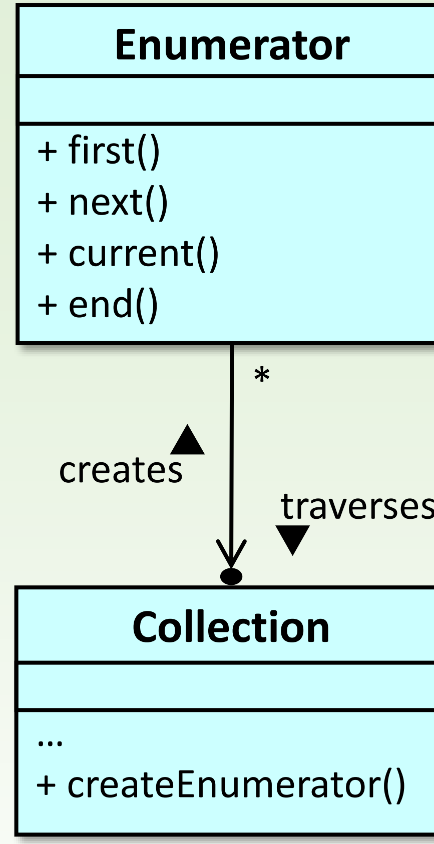
Communication diagram

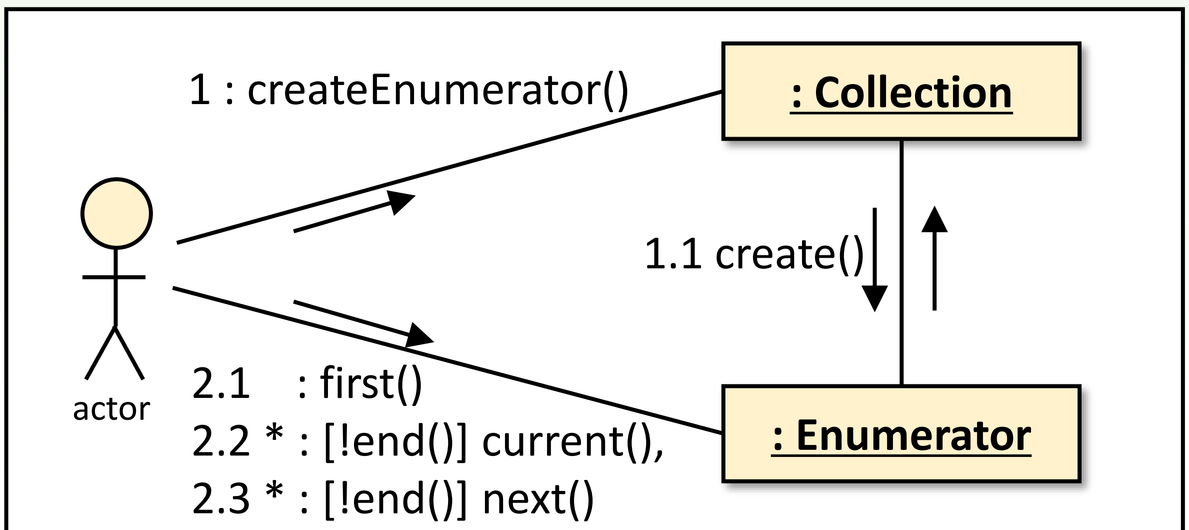
- A communication diagram shows with what kind of messages (method calls, signal sends) the objects communicate with each other.
- It is possible to indicate the order of the messages with numbers and to give guards (condition that allows the message) between square brackets.

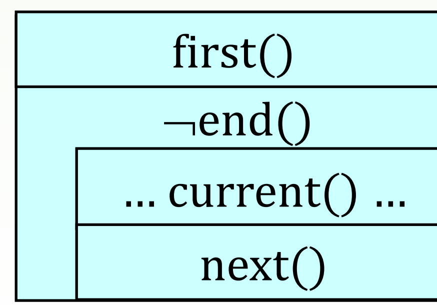


Example: Enumeration



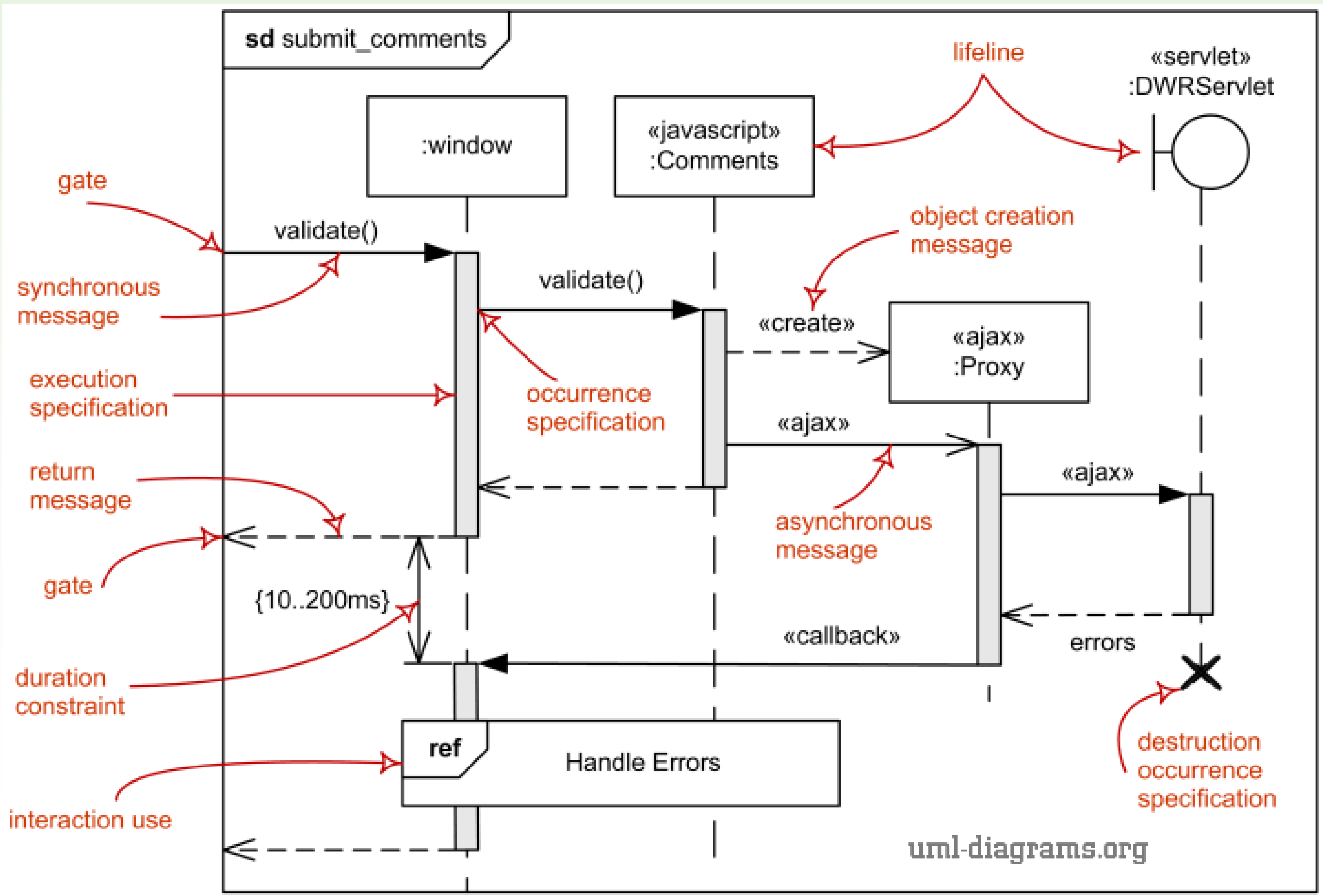






Sequence diagram

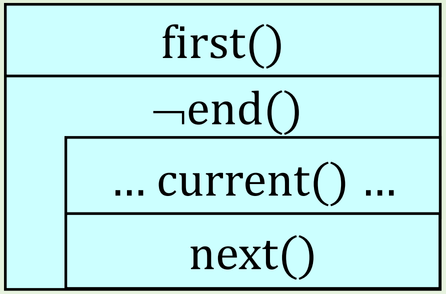
- It shows the order of the messages in the communication.

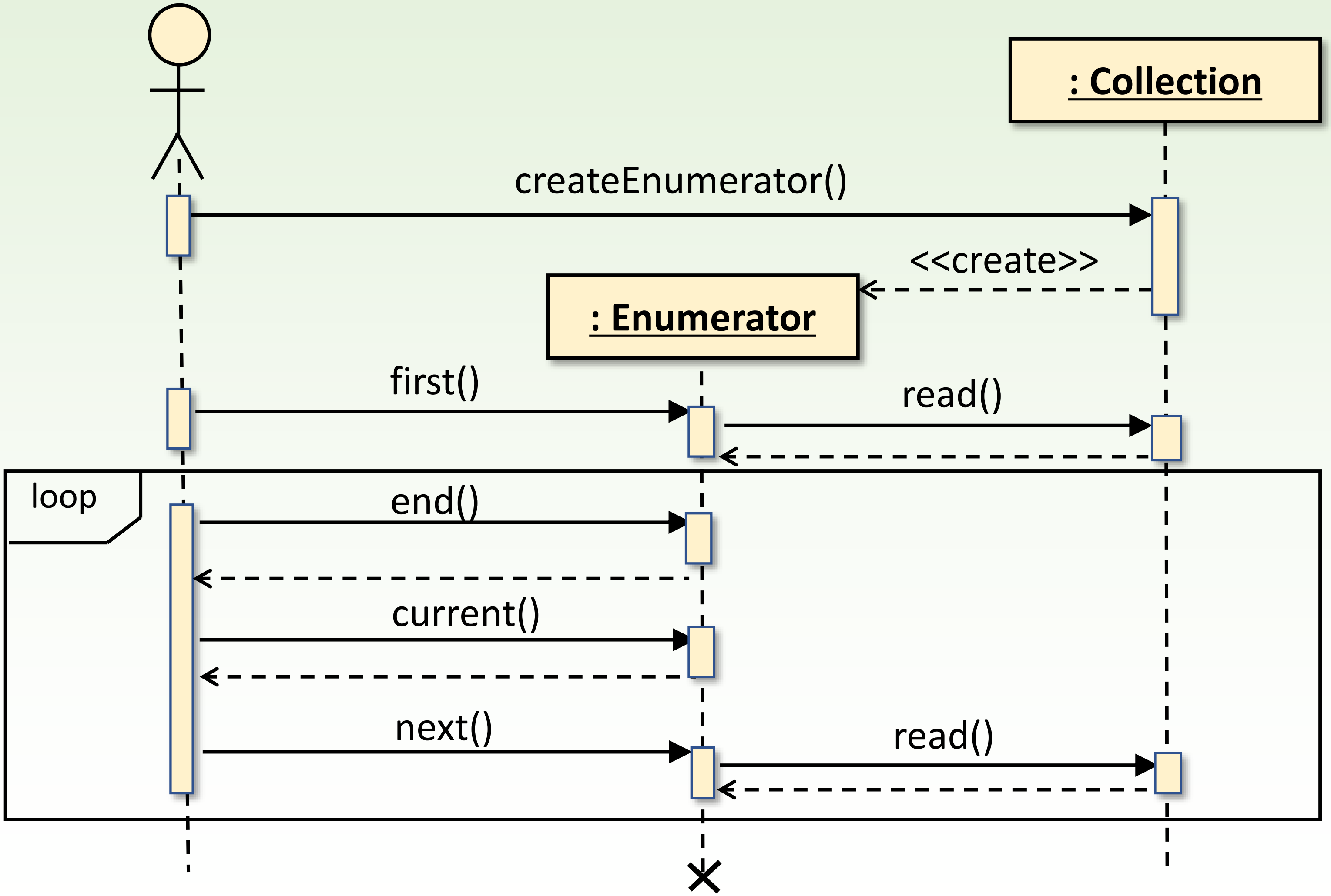


Messages

- Synchronous message: the sender gives the control to the receiver. It means a synchronous method call. ⟶
- Reply message: the receiver of a previous message sends it back to the sender when it has finished and wants to give back the control. Many times it is not represented in the diagram as it can be seen easily based on the environment. <- - - - - - - -
- The following messages count only in case of concurrent messages:
  ⟶ , previously ⇀
  - Asynchronous message: activity of the sender object does not stop, it is not important when the receiver receives the message.
  - Synchronization message: it blocks the activity of the sender as long as the receiver has not received the message.
  - Timeout waiting message: the sender waits for the receiver to receive message up to some fixed period of time.
  - Rendezvous message: the receiver waits for the sender to get a message from it.

Example: Enumeration

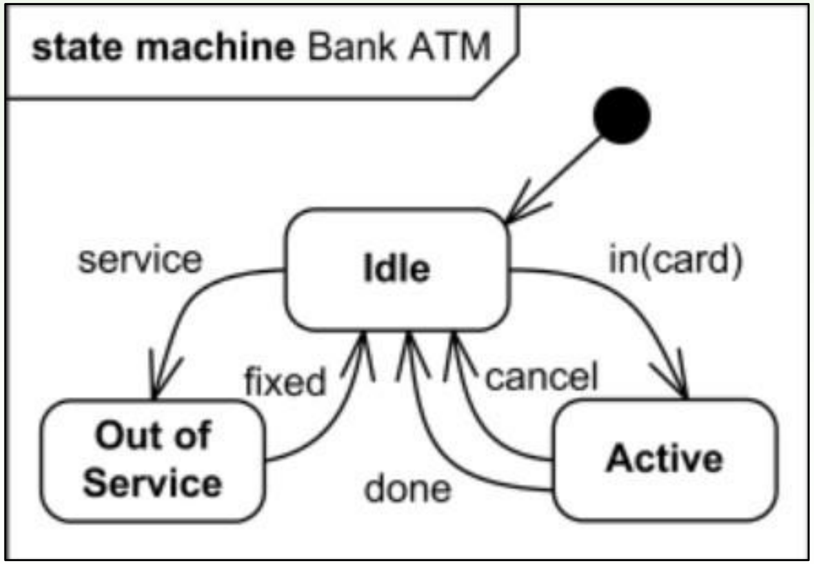




State machine

- A state machine diagram illustrates the lifecycle and the behavior of an object. It shows how the logical state changes of an objects when it gets a message (method call or signal).

- A state machine is a directed graph the nodes of which are the logical states, the edges of which are the transitions between them.
- Executable actions may belong to both the states and the transitions.



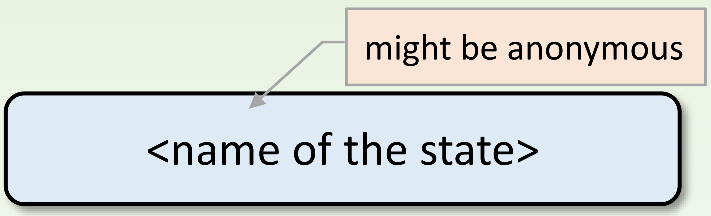
Lifecycle of an object

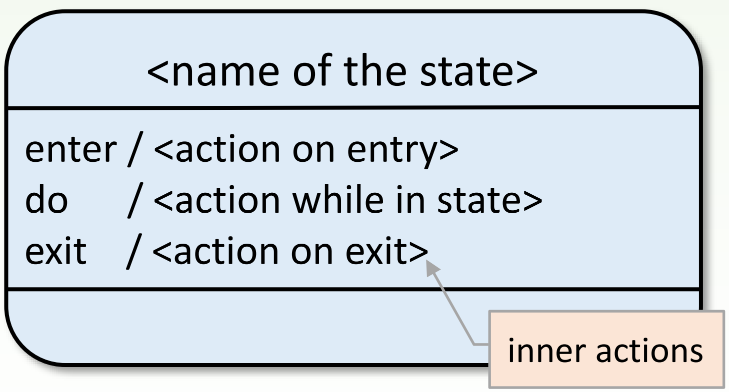
- During the lifecycle of an object:
  - it is created: by its constructor,
  - it works: communicates with other objects, which means they call each other's methods synchronously or asynchronously, or they react to the other's signal asynchronously and during that their properties may change, and
  - it is destroyed by its destructor.
- An object may have different physical states: a physical state means the current values of each of its attributes that may change during its lifetime.
- Often, one logical state includes several physical states with similar or common properties.

Notations of the states

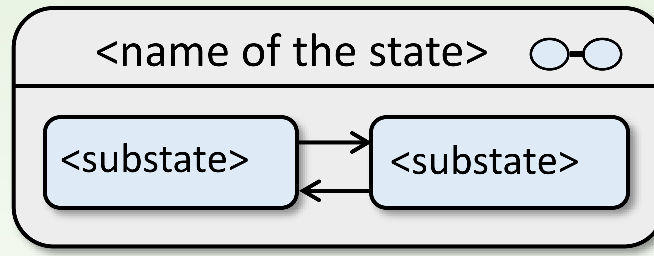
hierarchical

- Simple state
- Complex state

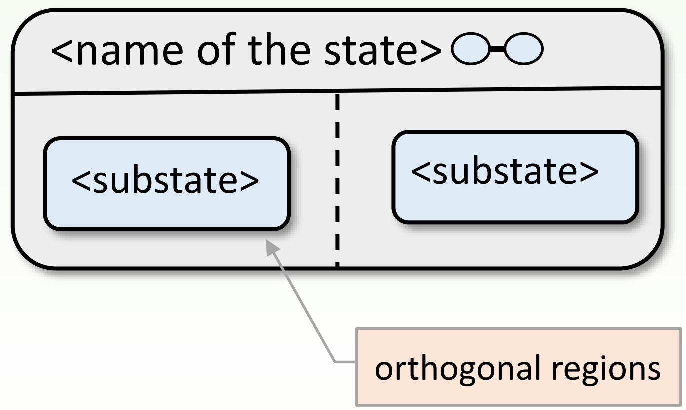




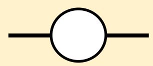
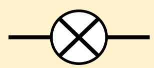
Sequential:



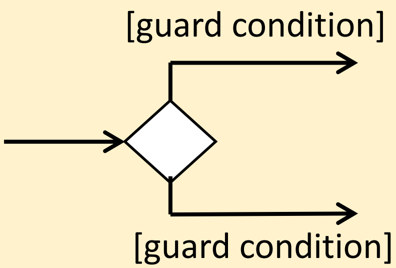
Concurrent:

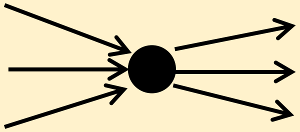
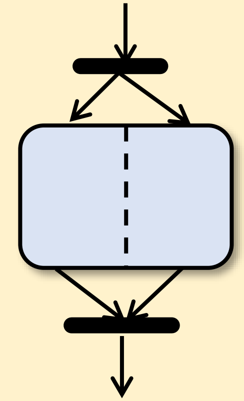


Pseudo states

- Start state 
- Final state 
- Entry point 
- Exit point 
- Terminate state 
- Shallow history 

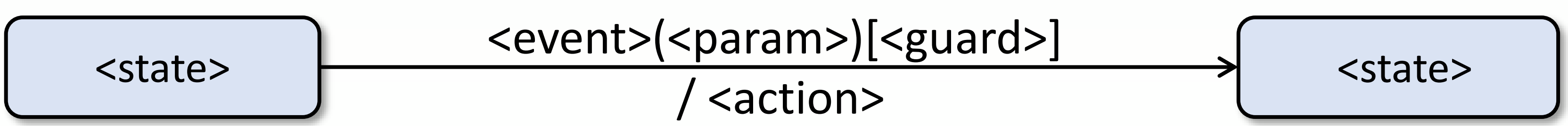
- Deep history 

- Choice 

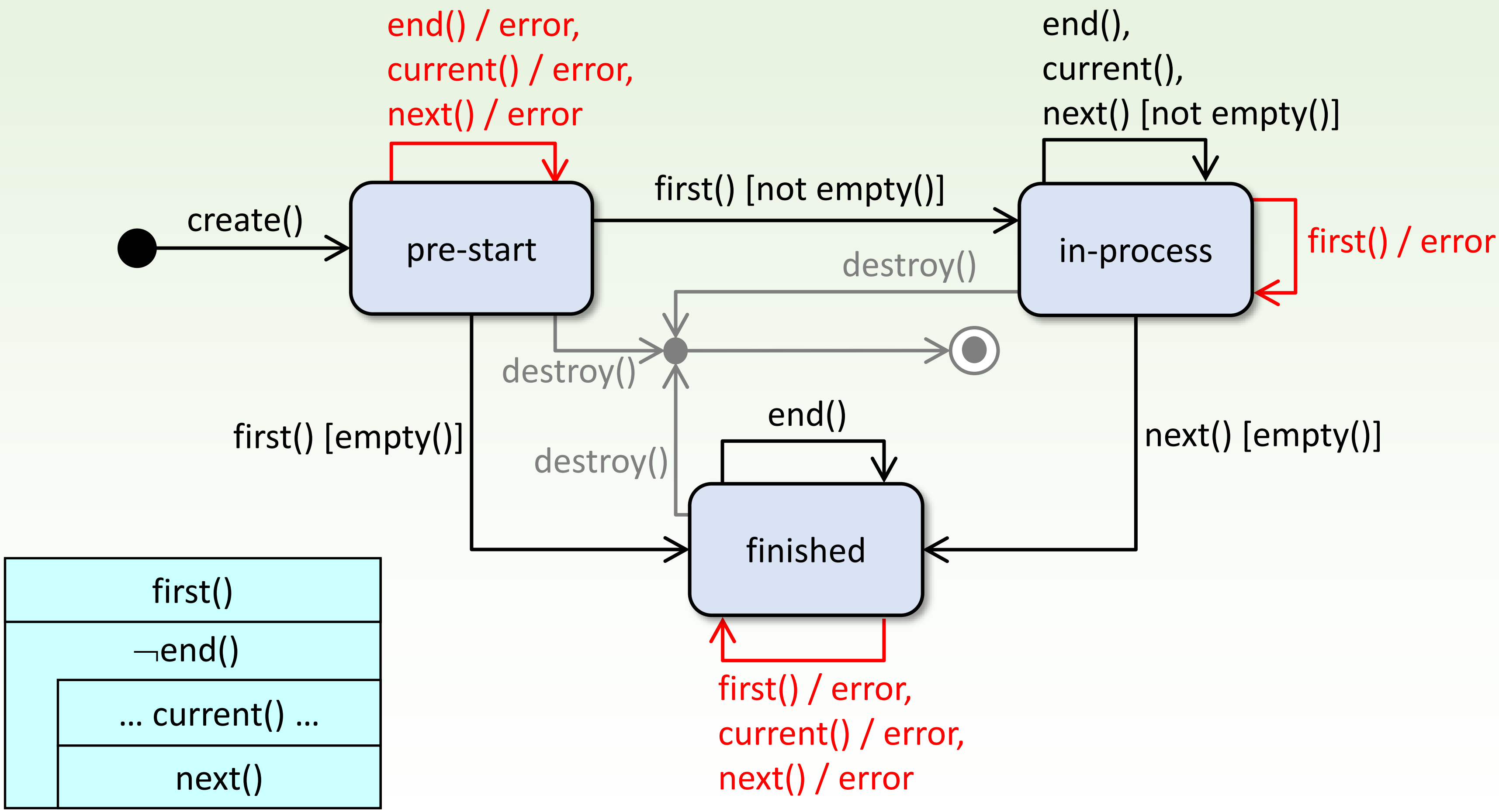
- Junction 
- Fork
- Join 

Notations of the transitions

- Properties of the transitions (any of them may be skipped):
  - trigger (*event, trigger*) of the transition with parameters
    - either a synchronous method call of the object
    - or an asynchronously processed signal which is sent to it
  - a guard condition, which necessarily precedes it
    - either a logical statement which depends on the parameters (*when*)
    - or a time-bound waiting condition (*after*)
  - an action assigned to the transition (a program operating with the attributes of the object and the parameters of the triggering event)
  - short explanatory description (often missing)
- A transition may be reflexive (inner) where the state does not change and enter and exit actions are not executed.



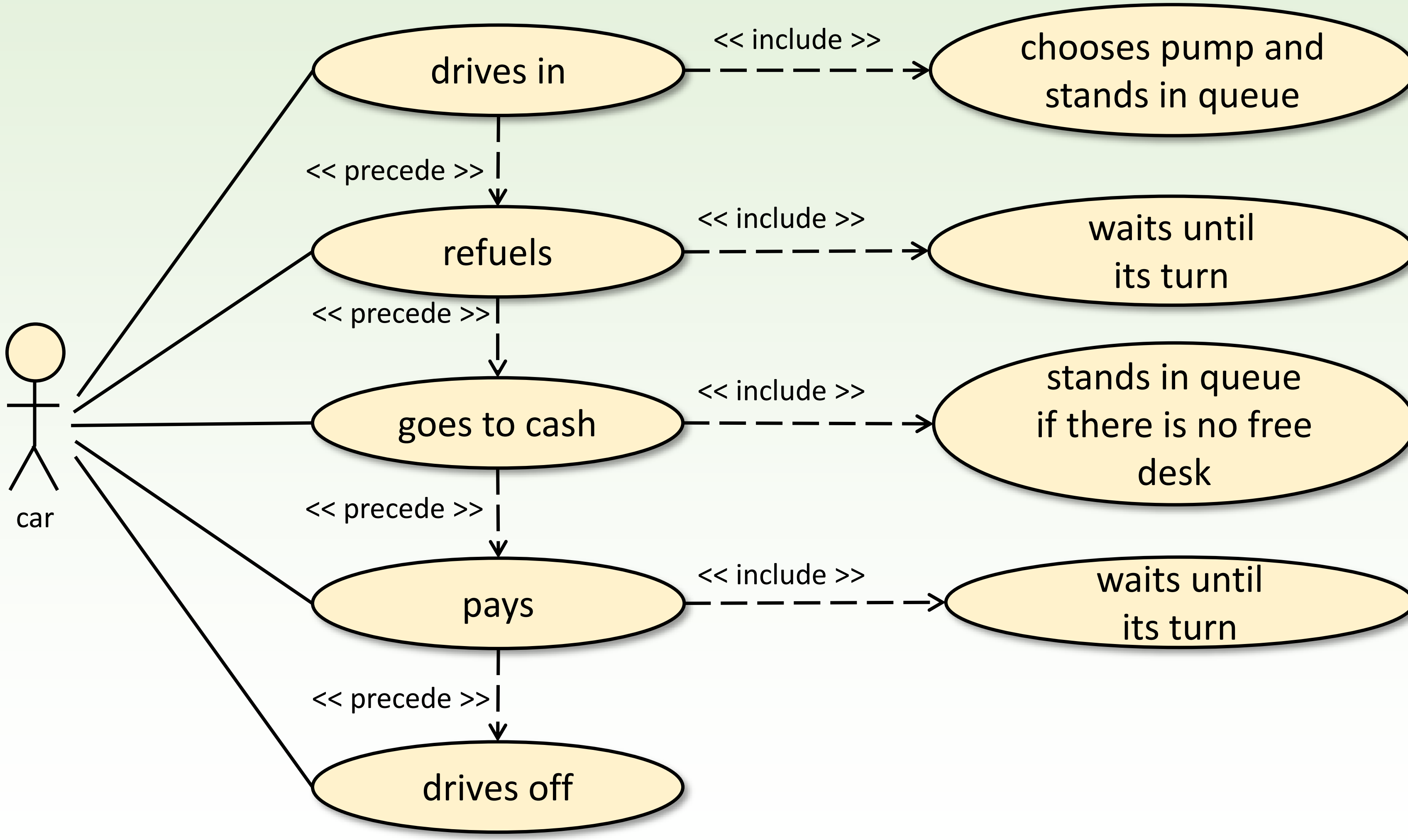
Example: Enumeration



Task

- On a petrol station, there are some pumps and some cash desks. The cars drive in and queue for a concrete pump. When it is their turn, they fill by a previously decided amount of fuel. After that, they go to pay. There is one queue for all the cash desks. When its their turn, the cash desk calculates the money to pay based on the amount of fuel. After paying, the cars drive off.
- Model this process for arbitrary number of cars acting concurrently.

Use case diagram

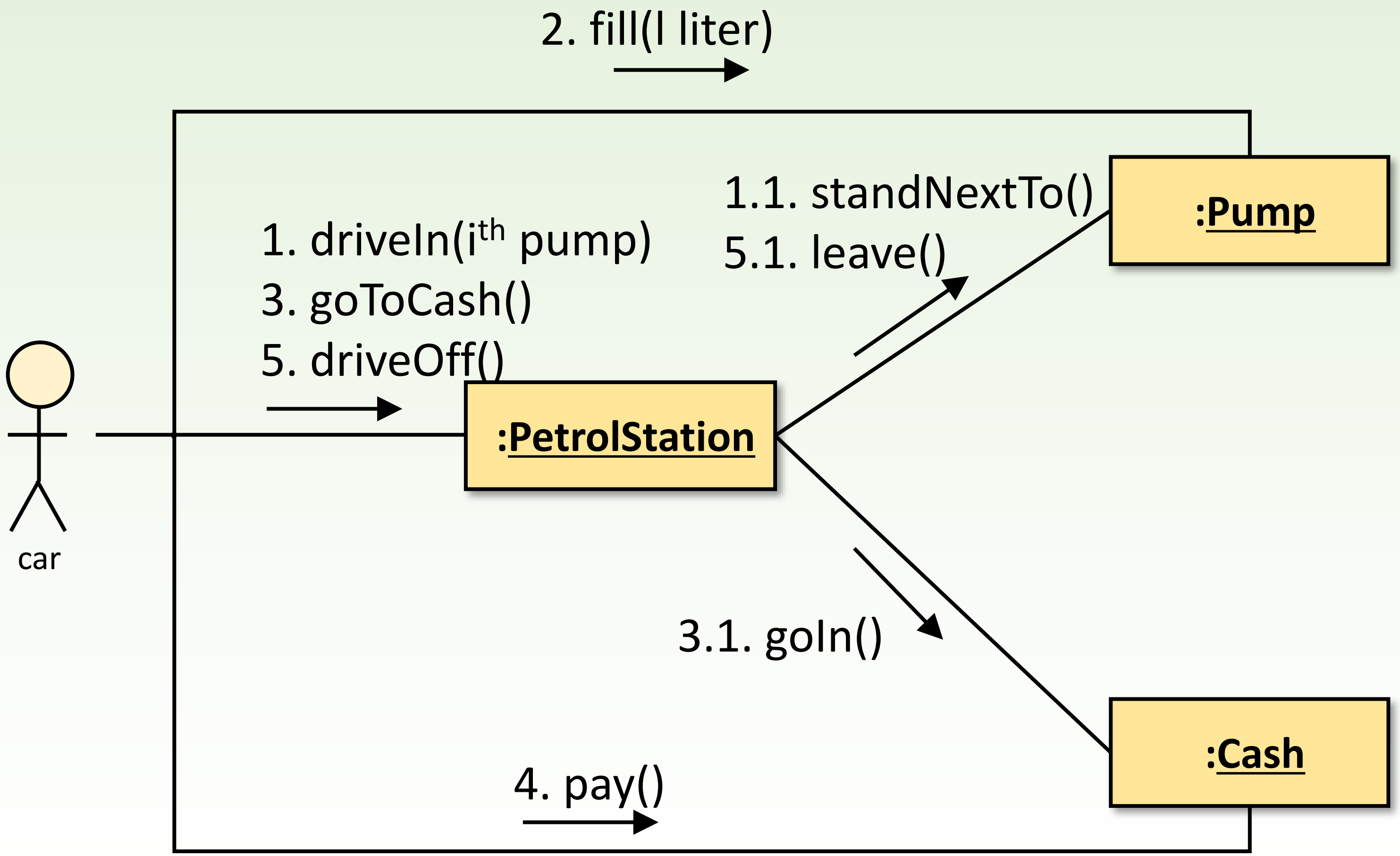


User story

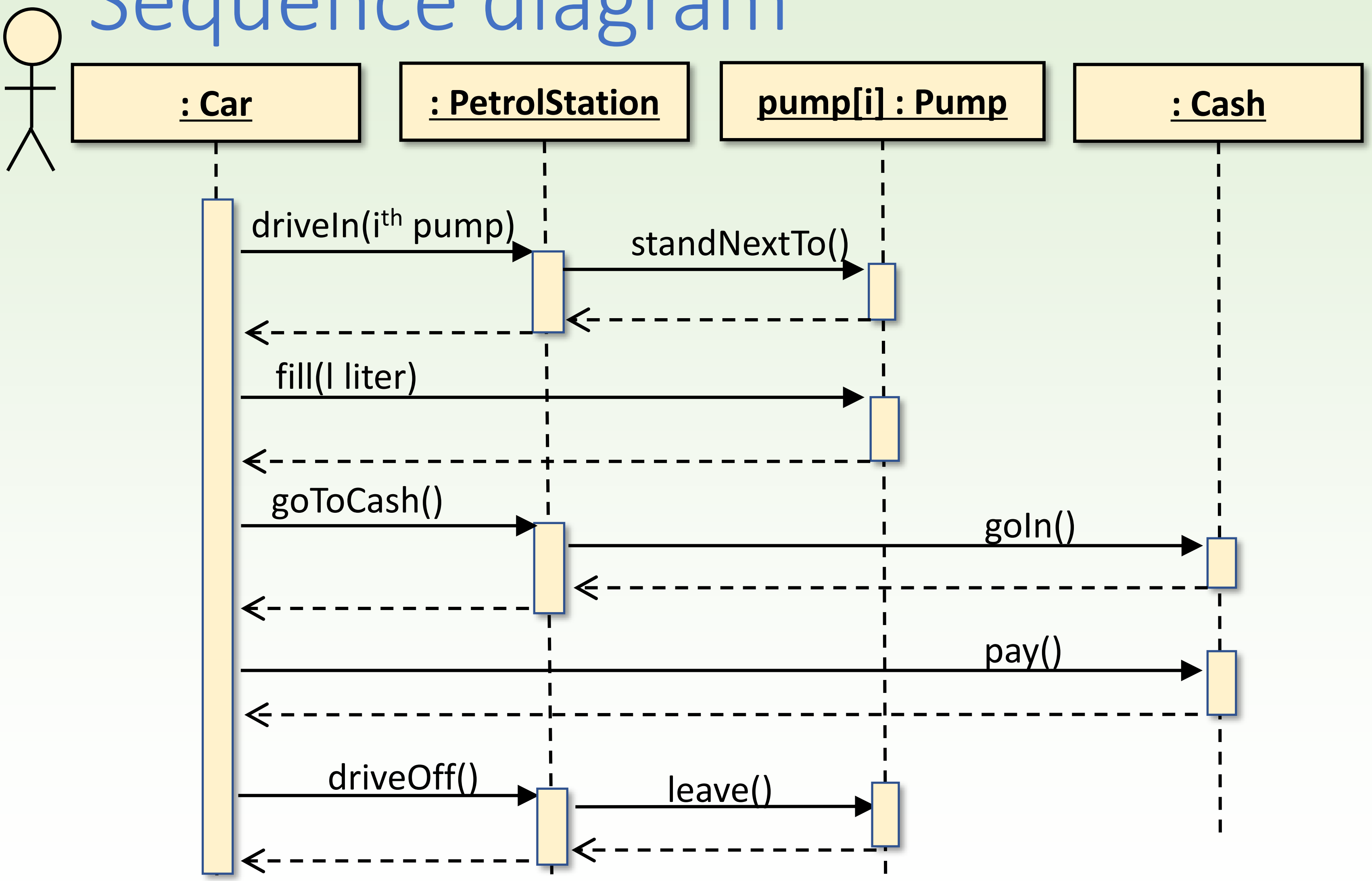
case		description
drives in	GIVEN	there is a petrol station with pumps
	WHEN	drives to an existing pump
	THEN	stands in the queue
refuels	GIVEN	in the queue of a pump
	WHEN	wants to get a given amount of fuels
	THEN	waits for its turns, then fills
goes to pay	GIVEN	there is a petrol station with cash desks
	WHEN	goes to the cash desks
	THEN	goes to a free cash desk or stands in the queue
pays	GIVEN	there is a petrol station with a cash desk, it stands next to a pump and it is at a free cash desk or is waiting in a queue to pay
	WHEN	pays
	THEN	if it is in a queue waiting for its turn, steps out of the queue, the cash computes the money to be paid and the display of the pump is reset
leaves	GIVEN	It stands next to a pump
	WHEN	drives off
	THEN	the queue at the pump becomes shorter

case		description
drives in	GIVEN	there is no petrol station
	WHEN	drives in
	THEN	error
drives in	GIVEN	there is petrol station
	WHEN	drives in to a nonexisting pump
	THEN	error
refuels	GIVEN	it is not at the given pump
	WHEN	fills
	THEN	error
goes to cash desk	GIVEN	there is no petrol station
	WHEN	goes in to the cash desk
	THEN	error
pays	GIVEN	there is no petrol station, or it is not at a pump
	WHEN	pays
	THEN	error
leaves	GIVEN	it is not at a pump
	WHEN	drives off
	THEN	error

Communication diagram



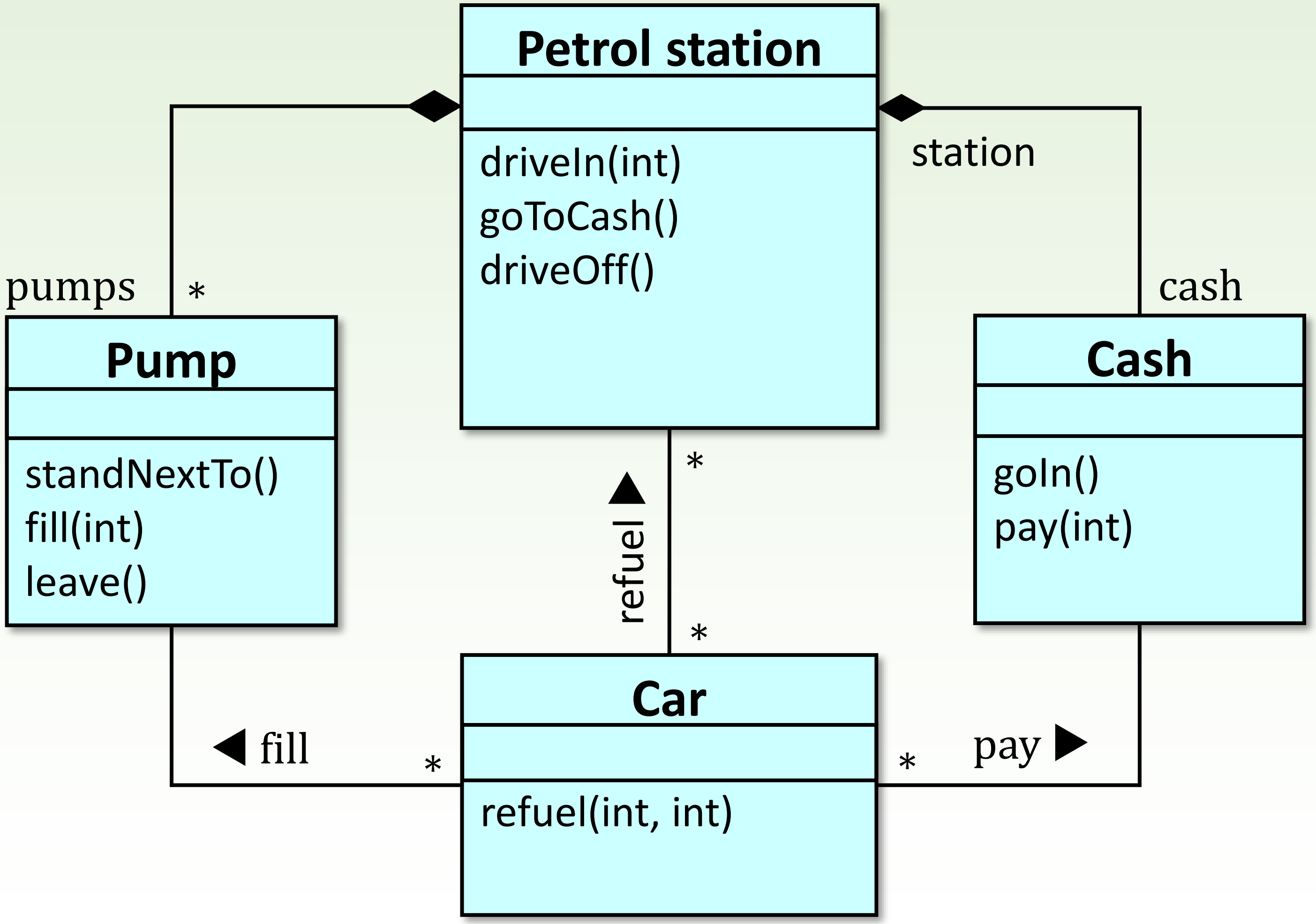
Sequence diagram



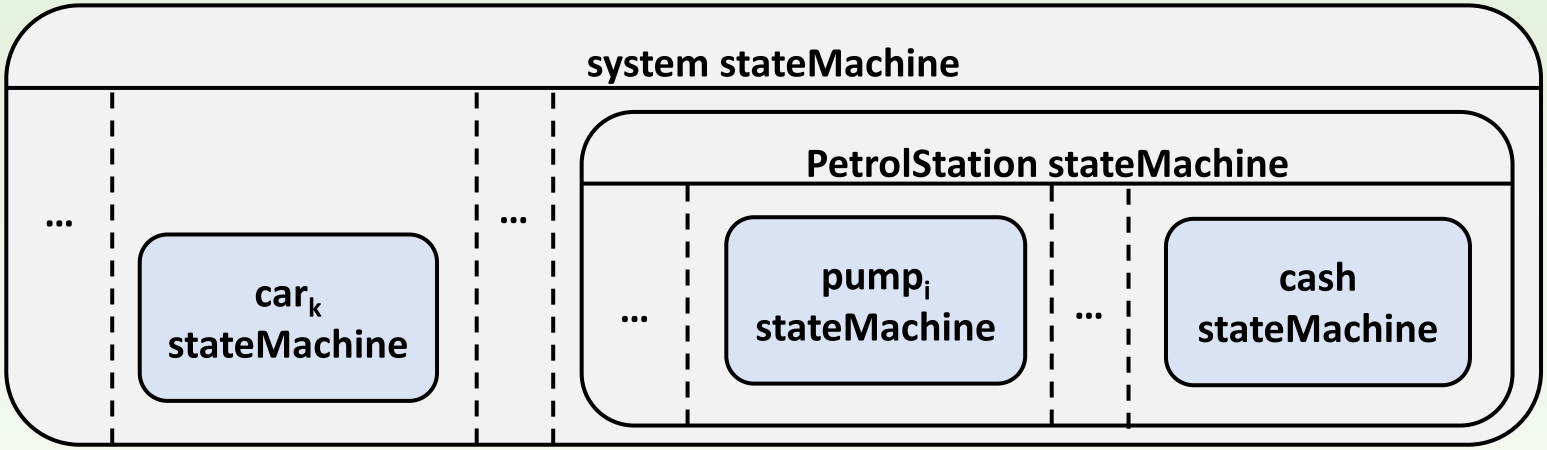
Result of the analysis

- Objects and their activities:
  - cars (they refuel)
  - petrol station (where the cars drive in, refuel, go to cash, and pay and from where they drive off)
  - pumps (where the car stands next to and fills and from where it leaves)
  - cash with more desks (where the driver goes in and pays)
- Relationships between objects:
  - parts of the petrol station are the pumps and the cash
  - a car temporarily gets in touch with a pump and a cash

Class diagram



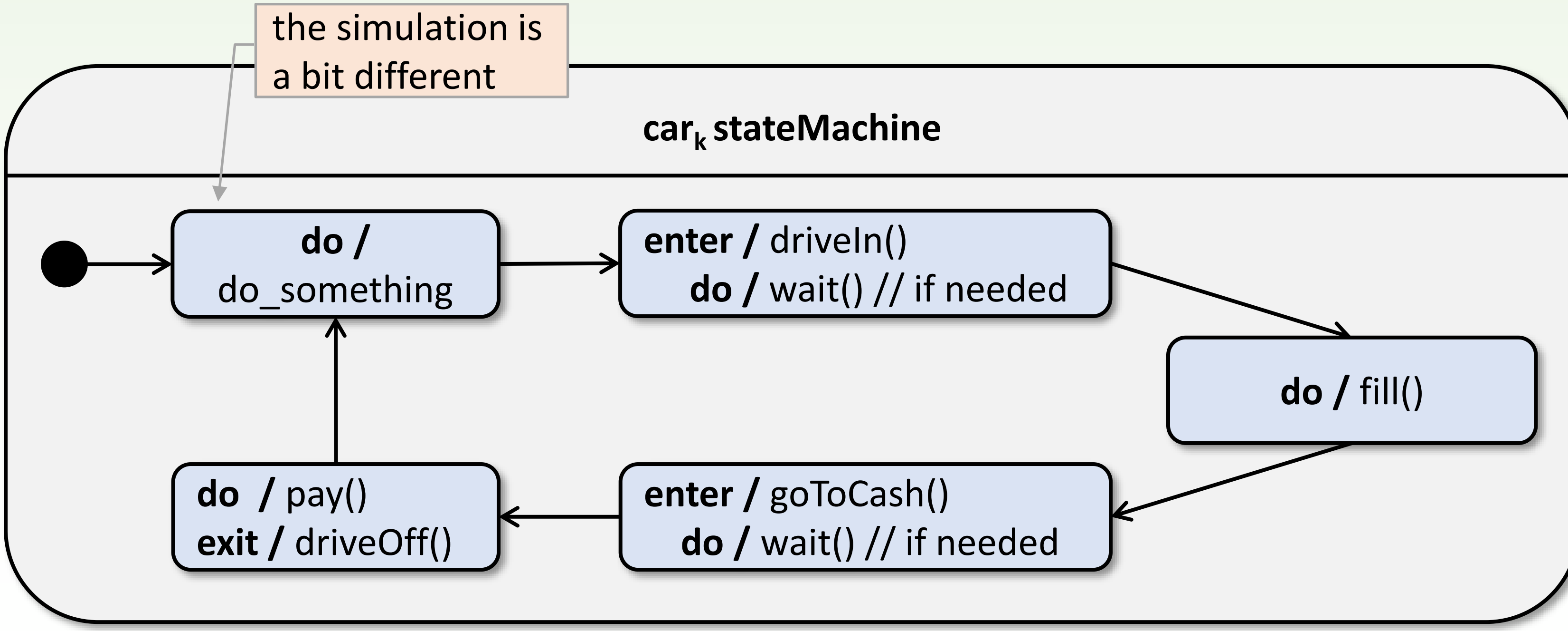
State machine of the system



- State of the system is determined by the state of the cars and the petrol station. State of the petrol station is determined by the state of the pumps and the cash.
- Cars are the so-called active objects: they perform actions concurrently, their state machines run on different threads.
- Petrol station is a passive object: its state machine runs synchronously (by calling its methods) with the state machine of other objects. It does not need other thread.

State machine of the cars

- A car may be in five states that may change cyclically due to the methods of the petrol station:
  - does something else; drives in (to a pump) and waits; fills; goes to the cash and waits; pays and drives off
- Transitions are triggered by the end of the actions of the states.



Class Car

Car
- name : string
+ Car(str : string) + refuel (petrol : PetrolStation, i : int, l : int) : void + getName() : string { query }

p: Pump
p := petrol.driveIn(this, i)
p.fill(this, l)
c : Cash
c := petrol.goToCash(this)
n : int
n := c.pay(this)
petrol.driveOff(this)

Class Car

car.h

```
class PetrolStation;
class Car {
public:
    Car(const std::string &str) : _name(str) {}
    ~Car() { _fuel.join(); }
    std::string getName() const { return _name; }
    void refuel(PetrolStation* petrol, unsigned int i, int l) {
        _fuel = new std::thread(activity, this, petrol, i, l);
    }
private:
    std::string _name;
    std::thread _fuel;
    void activity(PetrolStation* petrol, unsigned int i, int l);
};
```

waits for the end of the extra thread

it runs on an extra thread
#include <thread>

car.cpp

```
void Car::activity(PetrolStation* petrol, unsigned int i, int l) {
    if ( nullptr == petrol ) return;          // if there is no petrol station
    Pump *pump = petrol ->driveIn(this, i);   // goes to the ith pump
    if ( nullptr == pump ) return;            // if there is no ith pump
    pump->fill(this, l);                      // fills l liter fuel
    Cash *cash = petrol->goToCash(this);
    if ( nullptr == cash ) return;            // if there is no cash
    int n = cash->pay(this);
    petrol->driveOff(this);
}
```

Class PetrolStation

PetrolStation

- pumps : Pump[*]
- cash : Cash
- unit : int

+ PetrolStation(n : int, m : int)
+ driveIn(car : Car, i:int) : void
+ goToCash(car : Car) : void
+ driveOff(car : Car) : void
- search(car : Car, ind : int) : bool { query }
+ getUnit() : int { query }
+ setUnit() : int
+ getQuantity(i:int) : int { query }
+ resetQuantity(i:int) : void

pumps[i].standNextTo(car)

cash.goIn(car)

l,i := search(car)
pumps[i].leave(car)

return LinSearch(car in pumps)

return pumps[i].getQuantity()

pumps[i].resetQuantity()

Class PetrolStation

```
class PetrolStation {
public:
    PetrolStation(int n, int m) {
        for(int i = 0; i < n; ++i) _pumps.push_back( new Pump() );
         _cash = new Cash(this, m);
    }
    ~PetrolStation() { for( Pump* p : _pumps ) delete p;  delete _cash; }

    bool driveIn(Car* car, unsigned int i);
    void goToCash(Car* car);
    bool driveOff(Car* car);

    int getUnit() const { return _unit; }
    void setUnit(int u) { _unit = u; }
    void resetQuantity(unsigned int i) { _pumps[i]->resetQuantity(); }
    int getQuantity(unsigned int i) const { return _pumps[i]->getQuantity(); }
private:
    std::vector<Pump*> _pumps;
    Cash* _cash;
    int _unit;

    bool search(Car* car, unsigned int &ind) const;
};
```

petrol.h

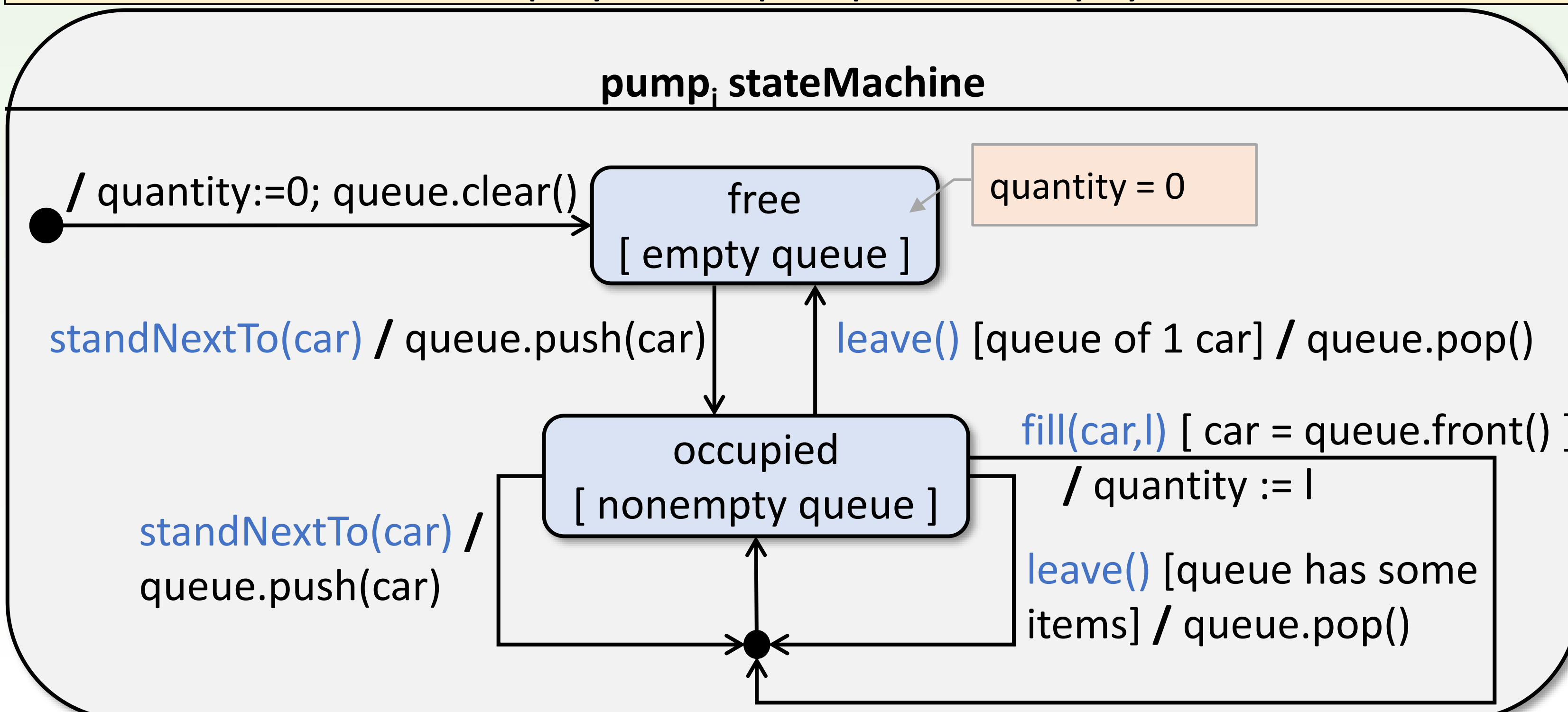
Methods of PetrolStation

```
Pump* PetrolStation::driveIn(Car* car, unsigned int i){
    if ( i >= _pumps.size() ) return nullptr;
    _pumps[i]->standNextTo(car);
    return _pumps[i];
}
Cash* PetrolStation::goToCash(Car* car){
    if (nullptr == _cash ) return nullptr;
    _cash->goIn(car);
    return _cash;
}
bool PetrolStation::driveOff(Car* car){
    unsigned int i;
    if ( !search(car, i) ) return false;
    _pumps[i]->leave();
    return true;
}
bool PetrolStation::search(Car* car, unsigned int &i) const {
    bool l = false;
    for ( i = 0; i < _pumps.size(); ++i) {
        if ( (l = _pumps[i]->getCurrent() == car) ) break;
    }
    return l;
}
```

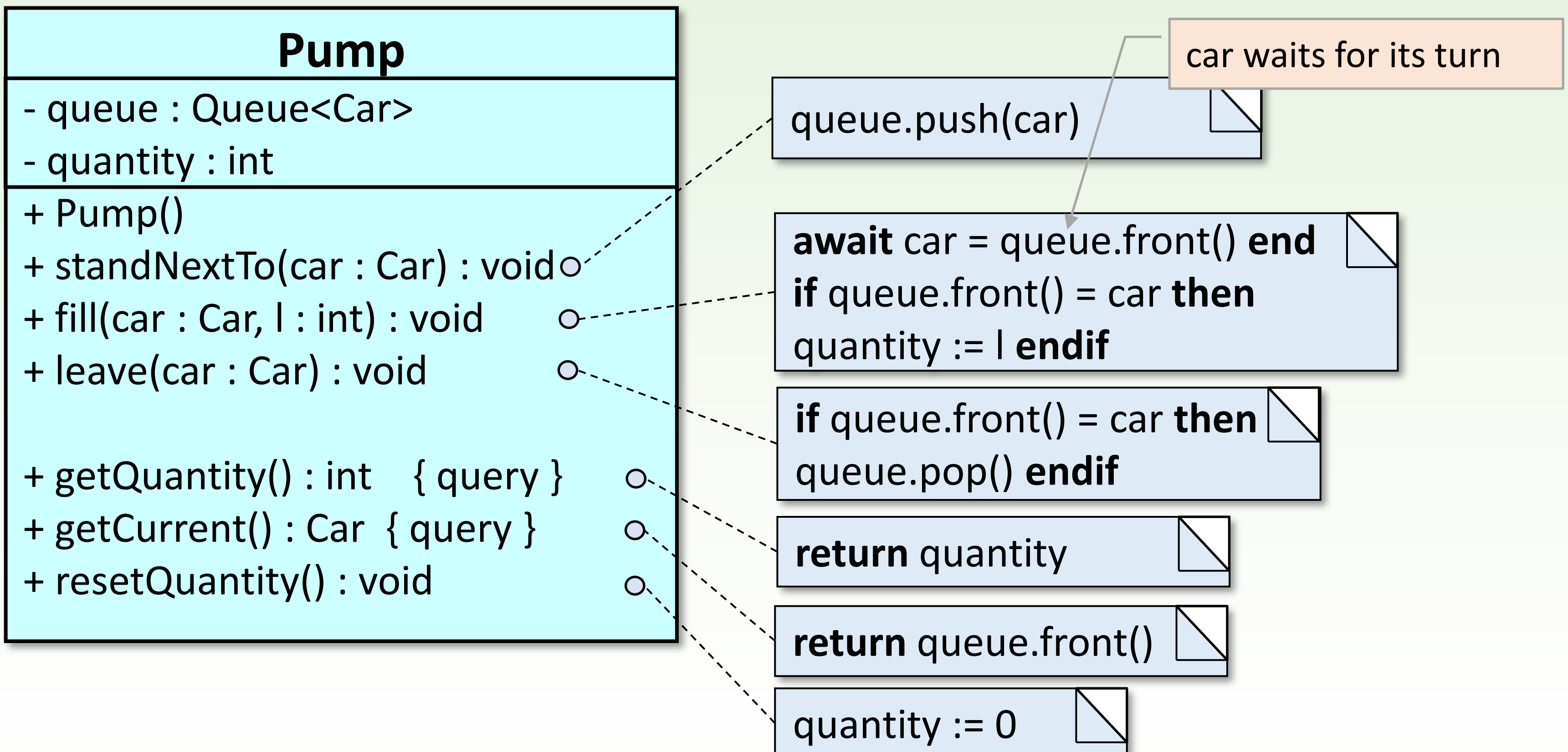
petrol.cpp

State machine of a pump

- A pump may be free or occupied.
- Methods standNextTo() and leave() effect the queue at the pump.
- Method fill() can be executed only in state occupied, when the car is in the front. In this case, the quantity of the fuel to be filled is given which can be seen on the display of the pump until the payment.



Class Pump


Pump
- queue : Queue<Car>
- quantity : int
+ Pump()
+ standNextTo(car : Car) : void
+ fill(car : Car, l : int) : void
+ leave(car : Car) : void
+ getQuantity() : int { query }
+ getCurrent() : Car { query }
+ resetQuantity() : void
car waits for its turn
queue.push(car)
await car = queue.front() **end**
if queue.front() = car **then**
quantity := l **endif**
if queue.front() = car **then**
queue.pop() **endif**
return quantity
return queue.front()
quantity := 0

Class Pump

```
class Car;

class Pump {
public:
    Pump() : _quantity(0) { }

    void standNextTo(Car* car);
    void fill(Car* car, int l);
    void leave(Car* car);

    Car* getCurrent() const { return _queue.front(); }
    int  getQuantity()  const { return _quantity; }
    void resetQuantity()       { _quantity = 0; }
private:
    int _quantity;
    std::queue<Car*> _queue;

    std::mutex _mu;
    std::condition_variable _cond;
};
```

#include <mutex>
#include <condition>

pump.h

Methods of Pump

```
void Pump::standNextTo(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    _queue.push(car);
}

void Pump::fill(Car* car, int l)
{
    std::unique_lock<std::mutex> lock(_mu);
    while( car !=_queue.front() ) _cond.wait(lock);
    _quantity = l;
}

void Pump::leave(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    if( car ==_queue.front() ) _queue.pop();
    _cond.notify_all();
}
```

to avoid simultaneous acces to _queue

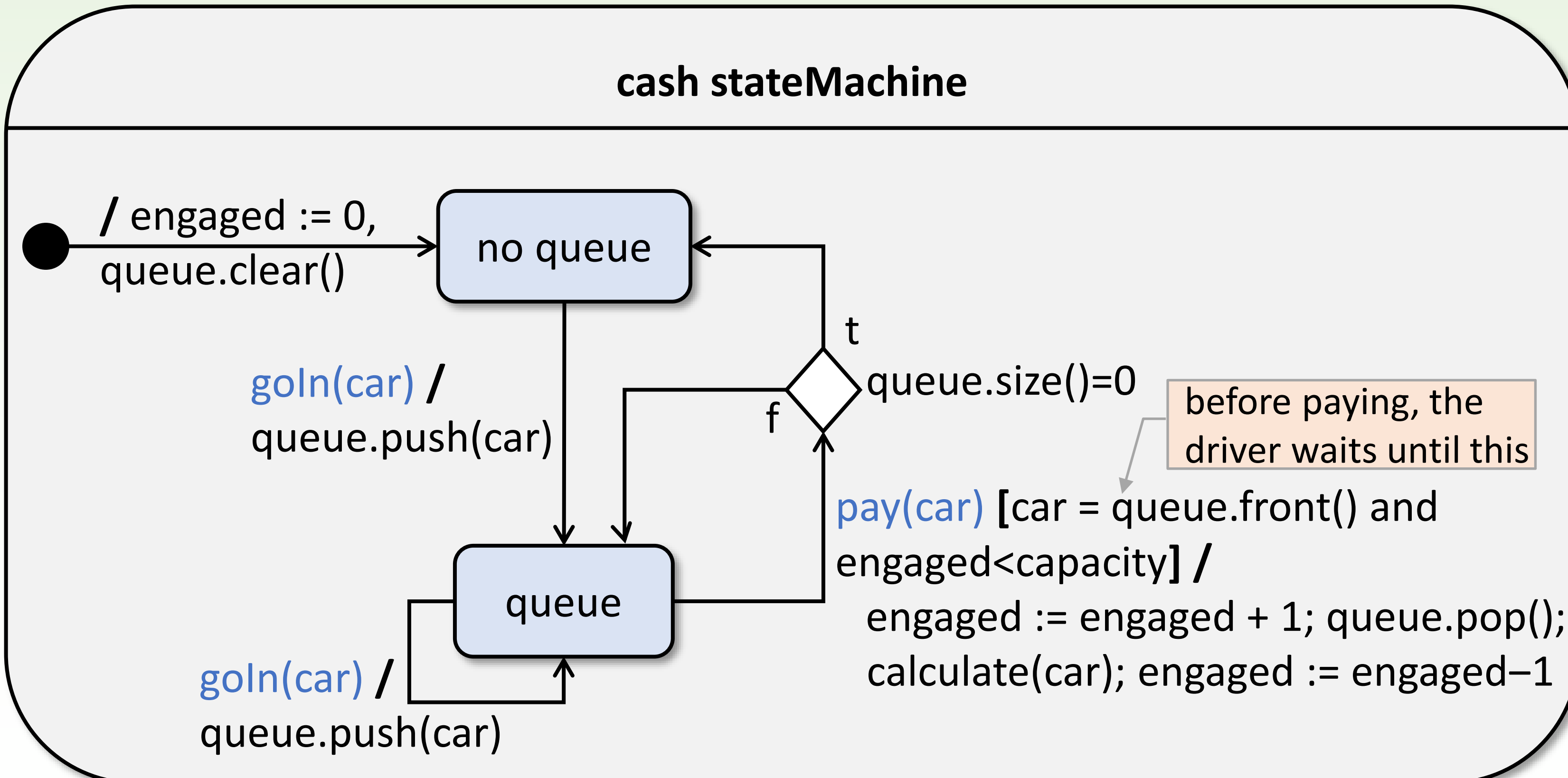
thread is waiting: it "falls asleep"

"wakes up" all those threads that are waiting at cond

pump.cpp

State machine of Cash

- In the cash, the drivers (cars) stand in queue.
- The states are effected by methods goIn() and pay(). Only that driver (car) can pay which is at the front of the queue and there is a free desk.



Cash

- A cash may be "used" by more cars. As many drivers (cars) may pay (engaged) as the number of the cash desks (capacity). The rest is waiting in the queue.

Cash

- queue : Queue<Car>
- engaged : int
- capacity : int
- station : PetrolStation

+ Cash(station : PetrolStation, cp : int)
+ goIn(car : Car) : void
+ pay(car : Car) : int

```
queue.push(car)
```

```
await  queue.front() = car and
       engaged < capacity  end
engaged := engaged +1
queue.pop()
l := station.search(car,i)
if not l then return nil endif
amount := station.getQuantity(i) *
                    station.getUnit()
station.resetQuantity(i)
engaged := engaged – 1
return amount
```

waits for its turn

index of the pump

Class Cash

```
class PetrolStation;
class Car;

class Cash {
public:
    Cash(PetrolStation* station, int cp): _station(station),
    _capacity(cp) {}
    void goIn(Car* car);
    int pay(Car* car);
private:
    PetrolStation* _station;

    std::atomic_int _engaged;
    int _capacity;
    std::queue<Car*> _cashQueue;

    std::mutex _mu;
    std::condition_variable _cond;
};
```

cash.h

Methods of Cash

```
void Cash::goIn(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    _cashQueue.push(car);
}
```

```
int Cash::pay(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    while( _cashQueue.front() != car || _engaged == _capacity ) {
        _cond.wait(lock);
    }
    ++_engaged;
    _cashQueue.pop();
    _cond.notify_all();
    _mu.unlock();

    unsigned int i;
    if ( !_station->search(car, i) ) return nullptr;
    int amount = _station->getQuantity(i) * _station->getUnit();
    _station->resetQuantity(i); // resets the display of the ith pump
    --_engaged;
    _cond.notify_all();
    return amount;
}
```

thread is waiting

starts those threads that are waiting at cond

cash.cpp