Reducing complexity of state

Mario Alviano

Introduction

- Managing the mutable state of entities is difficult
- State transitions may be complex
- We need secure patterns for state changes
  - Partially immutable entities
  - Entity state objects (single-thread)
  - Entity snapshot (multi-thread)
  - Entity relay (decomposition)

```
void withdraw(Money amount) {
    if(this.balance.moreThan(amount)) {      ①
        Money newBalance =
this.balance.subtract(amount);    ②
        this.balance = newBalance;    ③
    } else {
        throw new InsufficientFundsException();
    }
}
```

OK for single-thread

Race condition for multi-thread

Balance is inadvertently below 0
for a TOCTOU vulnerability

1. ATM withdrawal checks balance ($100 > $75): OK, proceed.
2. Automatic transfer checks balance ($100 > $50): OK, proceed.
3. ATM withdrawal calculates new balance: $100 - $75 = $25.
4. ATM withdrawal updates balance: $25.
5. Automatic transfer calculates new balance: $25 - $50 = -$25.
6. Automatic transfer updates balance: -$25.

It may be worse:
1, 2, 3, 5, 4, 6
gives a wrong final balance

Goodbye bank status!

Partially immutable entities

- Anything not expected to change should be immutable
- In class **Order**, field **custid** must not change
  - Does it make sense to transfer the basket of a customer to another customer?
  - If it doesn’t, why leaving such a possibility?
- Security by design
  - Make the entity partially immutable
  - Field **custid** must be immutable

```
class Order {
    private final CustomerID custid;
    Order(CustomerID custid) {
        Validate.notNull(custid);
        this.custid = custid;    ②
    }
    public CustomerID getCustid() {
        return custid;
    }
}
class SomeOtherPartOfFlow {
    void processPayment(Order order) {
        registerDebt(order.getCustid(), order.value());
        ...
    }
}
```

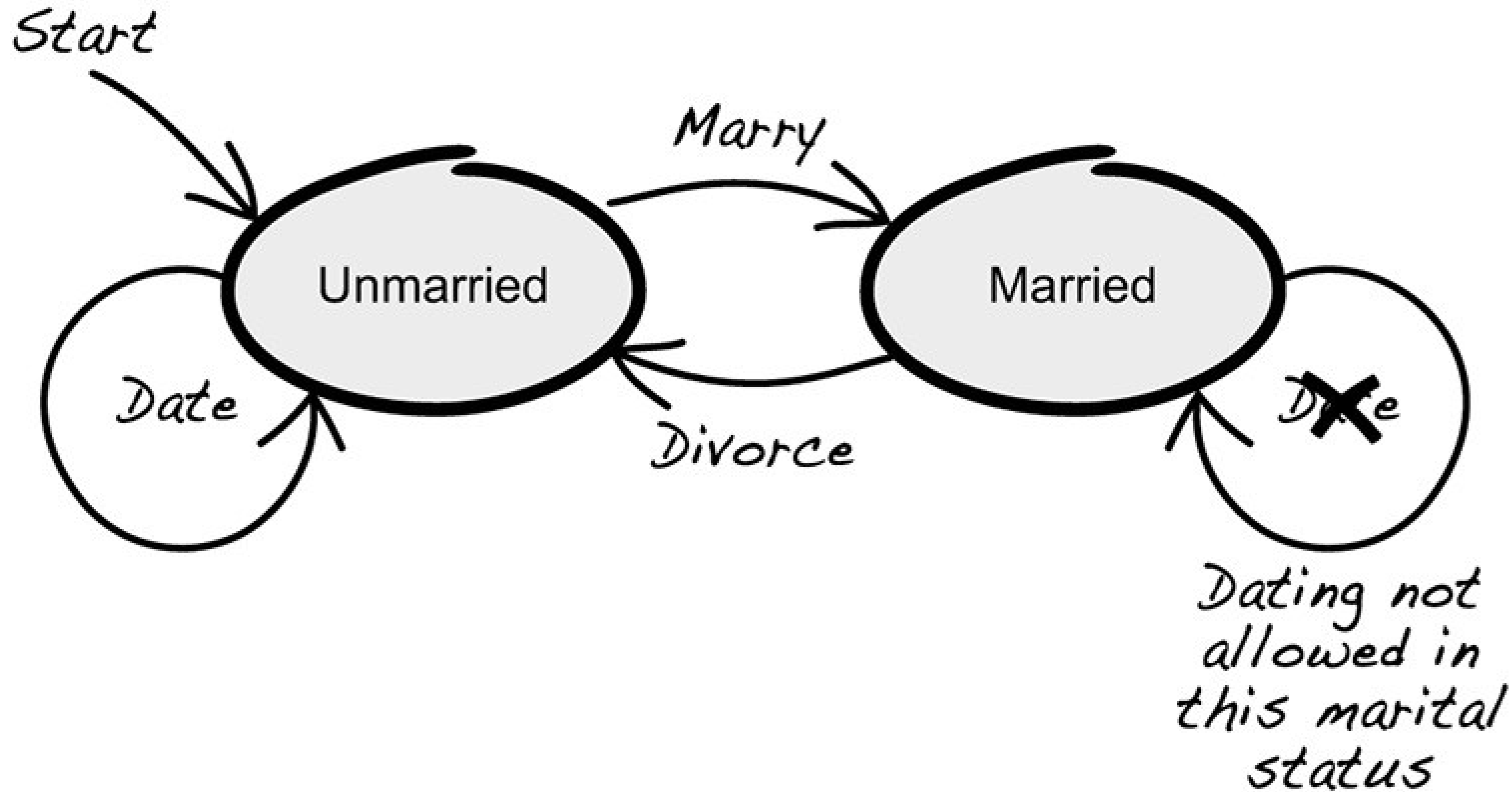
Field custid is final, it cannot change
(if CustomerID is immutabile)

In this case we can also remove the getter and make custid public

The following code results into a compilation error

```
Order order = ...
order.custid = new CustomerID(...);
```

Entity state objects



How to represent such state changes?

Naive solution:
use many **if**

```java
public class Person {
    private final boolean married;
    public Person(boolean married) {
        this.married = married;
    }
    public boolean isMarried() {
        return married;
    }
    public void date(Person datee) {}
}

public class Work {
    private Person boss;
    private Person employee;

    void afterwork() {
        // boss attempts to date
        if (!boss.isMarried()) {        ②
            boss.date(employee);
        } else {     ③
            logger.warn("bad egg");
        }
    }
```

Incorrect encoding

The state is not checked
in the entity

Very likely, some checks will be
forgotten in some usage of
the entity

```java
public class Person {
    private boolean married;

    public Person(boolean married) {
        this.married = married;
    }

    public boolean isMarried() {
        return married;
    }

    public void date(Person datee) {      ①
        if (!isMarried()) {     ②
            dinnerAndDrinks();
        } else {     ③
            logger.warn("bad egg");
        }
    }

    private void dinnerAndDrinks() {}
}

public class Work {
    Person boss = new Person(true);
    Person employee = new Person(false);

    void afterwork() {
        // boss attempts to date
        boss.date(employee);
    }
}
```

Incorrect encoding

The state is implicit

Likely, if statements were added on the base of specific cases

The state of the entity is very important: it must be carefully designed

In code, this fact translates into an ad-hoc class devoted to the state of the entity

```java
public class MaritalStatus {

    private boolean married = false;    ①

    public void date() {    ②
        validState(!married,    ③
                "Not appropriate to date when married");
    }

    public void marry() {
        validState(!married);    ③
        married = true;
    }

    public void divorce() {
        validState(married);    ④
        married = false;
    }
}
```

The state is explicitly represented

We can now also define unit tests for the state of the entity

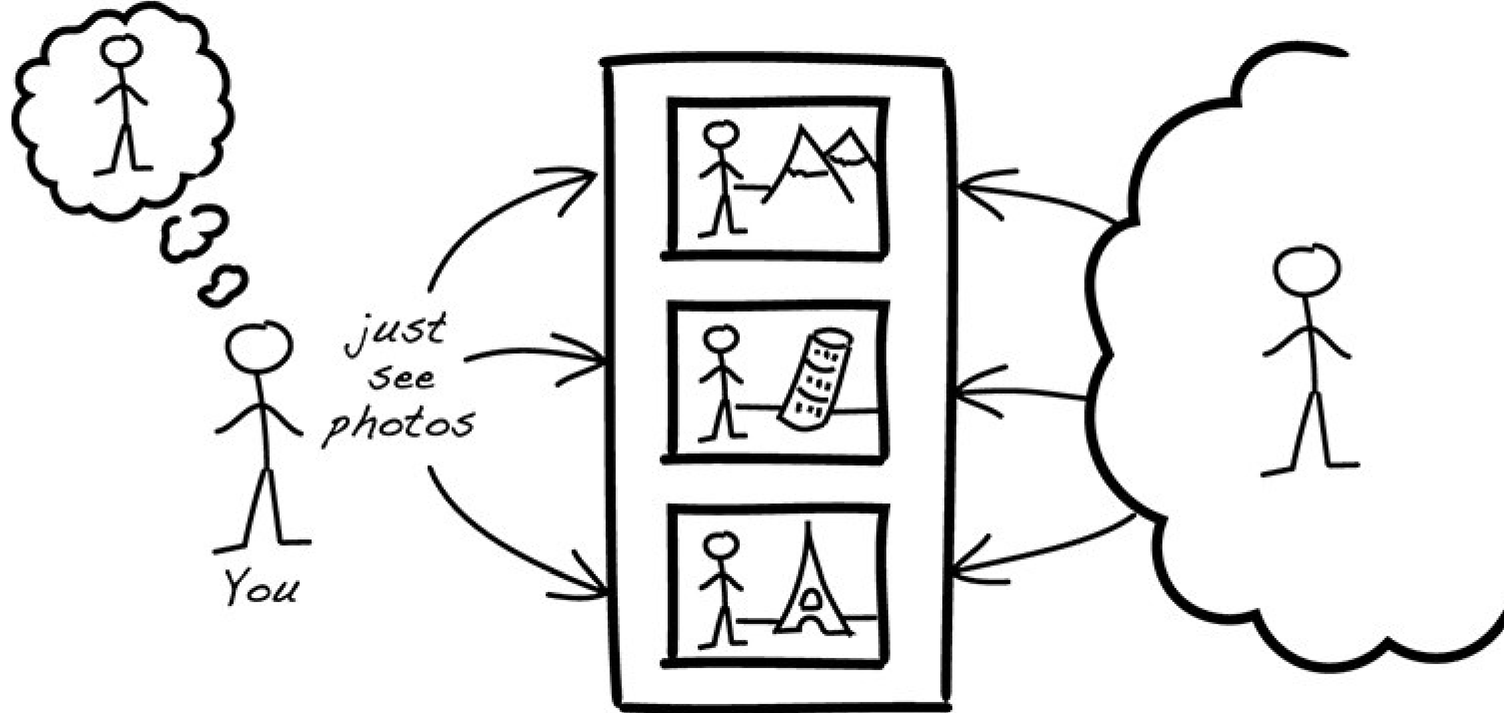
```java
public class Person {

    private MaritalStatus maritalStatus =
            new MaritalStatus();

    public void date(Person datee) {
        maritalStatus.date();    ①
        buydrinks();
        offerCompliments();
    }

    public void divorce() {
        maritalStatus.divorce();    ②
        ...
    }
    ...
}  ②
```

The entity calls methods of the state class

Illegal calls are identified (and logged)

Multi-thread environments

- Quite common (for example, web services)
- OK to share immutable objects (privitive domains)
- Sharing mutable objects is more subtle
  - Several methods have to be synchronized
  - Deadlock problems may arise
- Entity snapshots
  - The entity is not “represented” by mutable classes
  - We actually use (immutable) snapshots of the entity



Snapshot idea

A friend that you don’t see since a long time

You see their pictures on FB

Your friend is an entity

Each picture is a representation of your friend in a given instant in time

```
public class OrderSnapshot {    ①
    public final OrderID orderid;
    public final CustomerID custid;
    private final List<OrderItem> orderItemList;

    public OrderSnapshot(OrderID orderid;
                         CustomerID custid,
                         List<OrderItem> orderItemList)
{
        this.orderid = notNull(orderid);
        this.custid = notNull(custid);
        this.orderItemList =
            Collections
                .unmodifiableList(
                    notNull(orderItemList));    ②
        checkBusinessRuleInvariants();
    }

    public List<OrderItem> orderItems() {
        return orderItemList;    ②
    }

    public int nrItems() {    ③
        ...
    }

    private void checkBusinessRuleInvariants() {
        validState(nrItems() <= 38, "Too large for ordinary
shipping");
    }

}    ④

public class OrderService {
    public OrderSnapshot findOrder(OrderID orderid) ...
    public List<OrderSnapshot>
findOrdersByCustomer(CustomerID custid) ...
}
```

An entity snapshot is encoded by an immutable object

Usually, the snapshot is built from data stored in a database

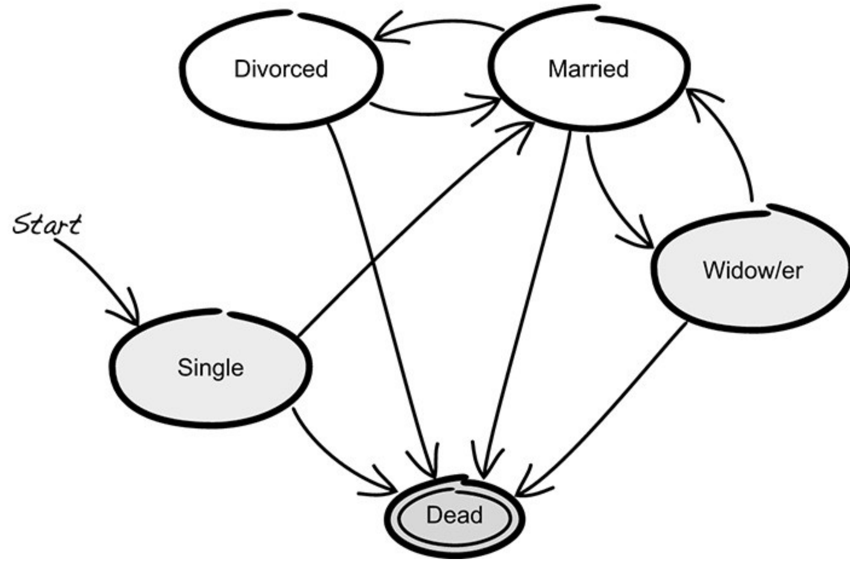
Many business logic is in the snapshot

State changes must be managed by another class (it violetes encapsulation)

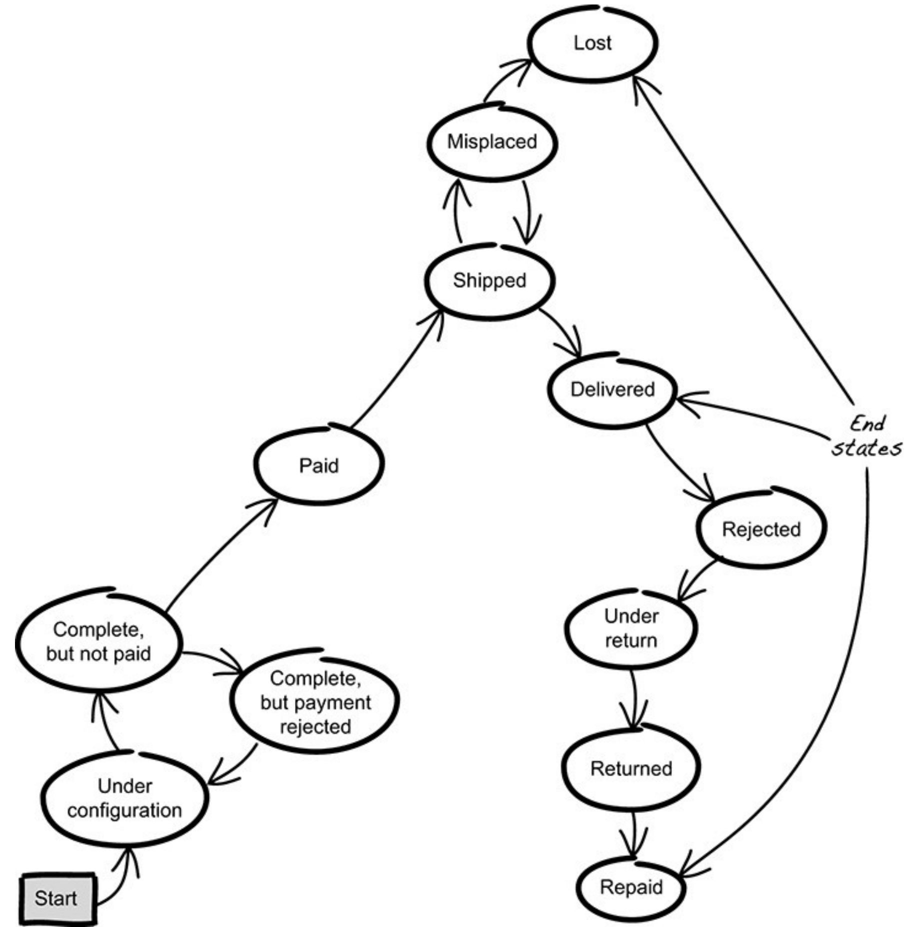
Synchronization is required only for “taking” the snapshot

Entity relay

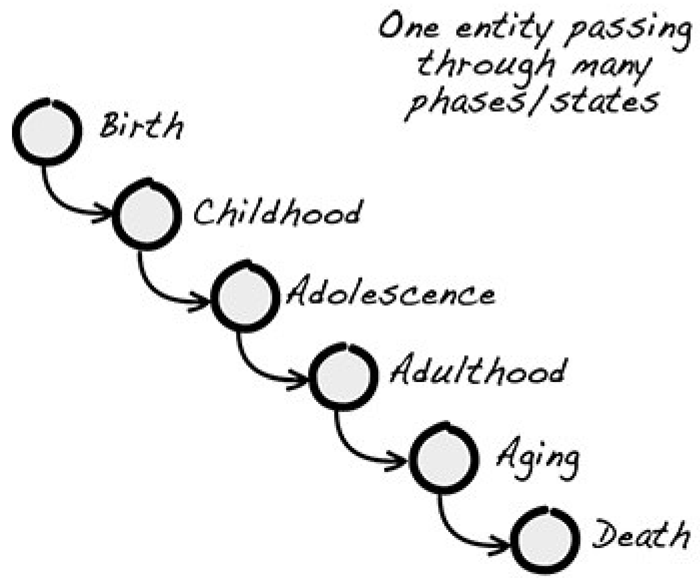
- Pattern to handle entities with many states
- The idea is to identify life phases of the domain entity
- Every phase is then represented in code by a new entity
- Phase changes imply a change of the entity

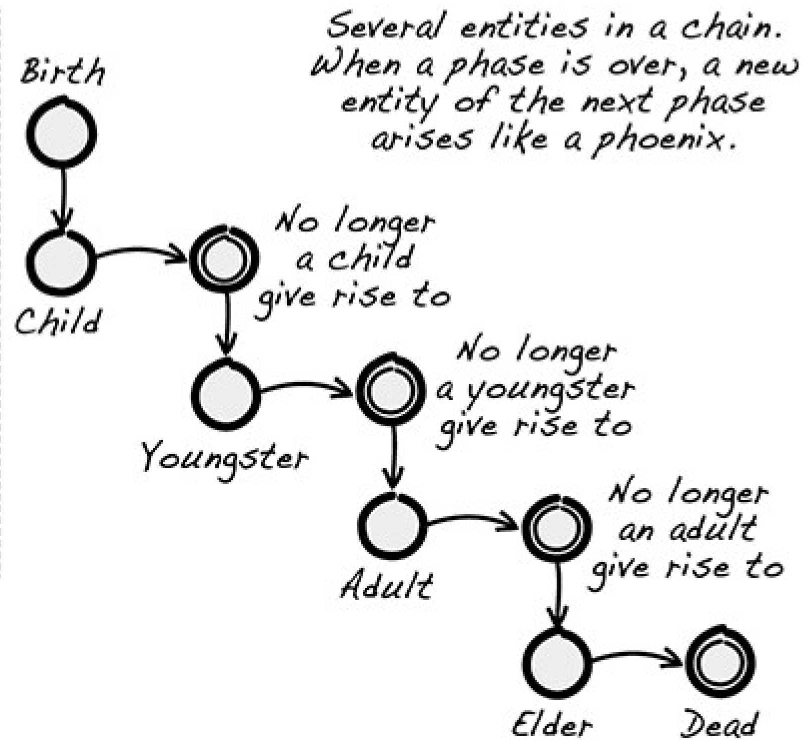


Entity with low number of states: directly manageable



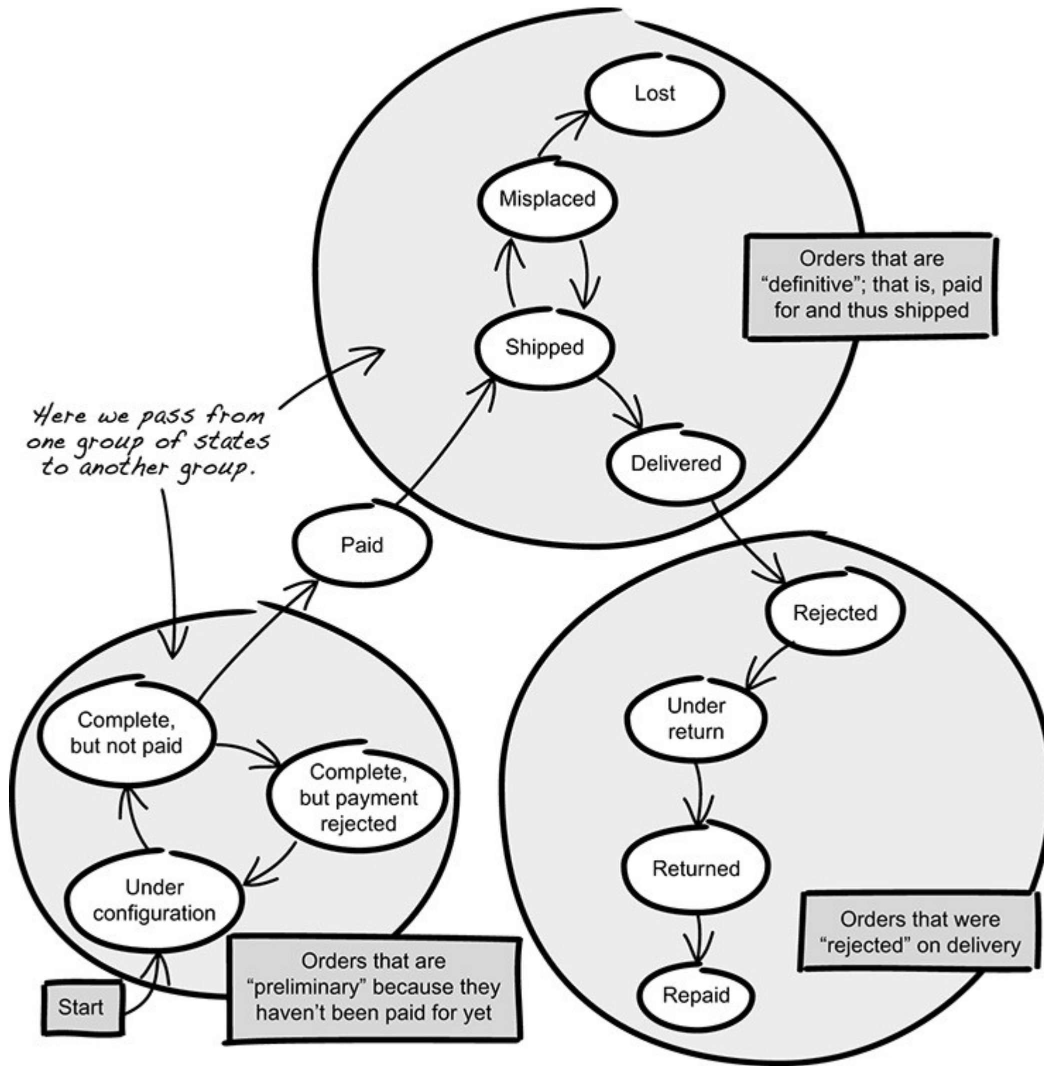
Entity with high number of states: better to group states by phases





A person seen as an entity,
with several states from birth to death

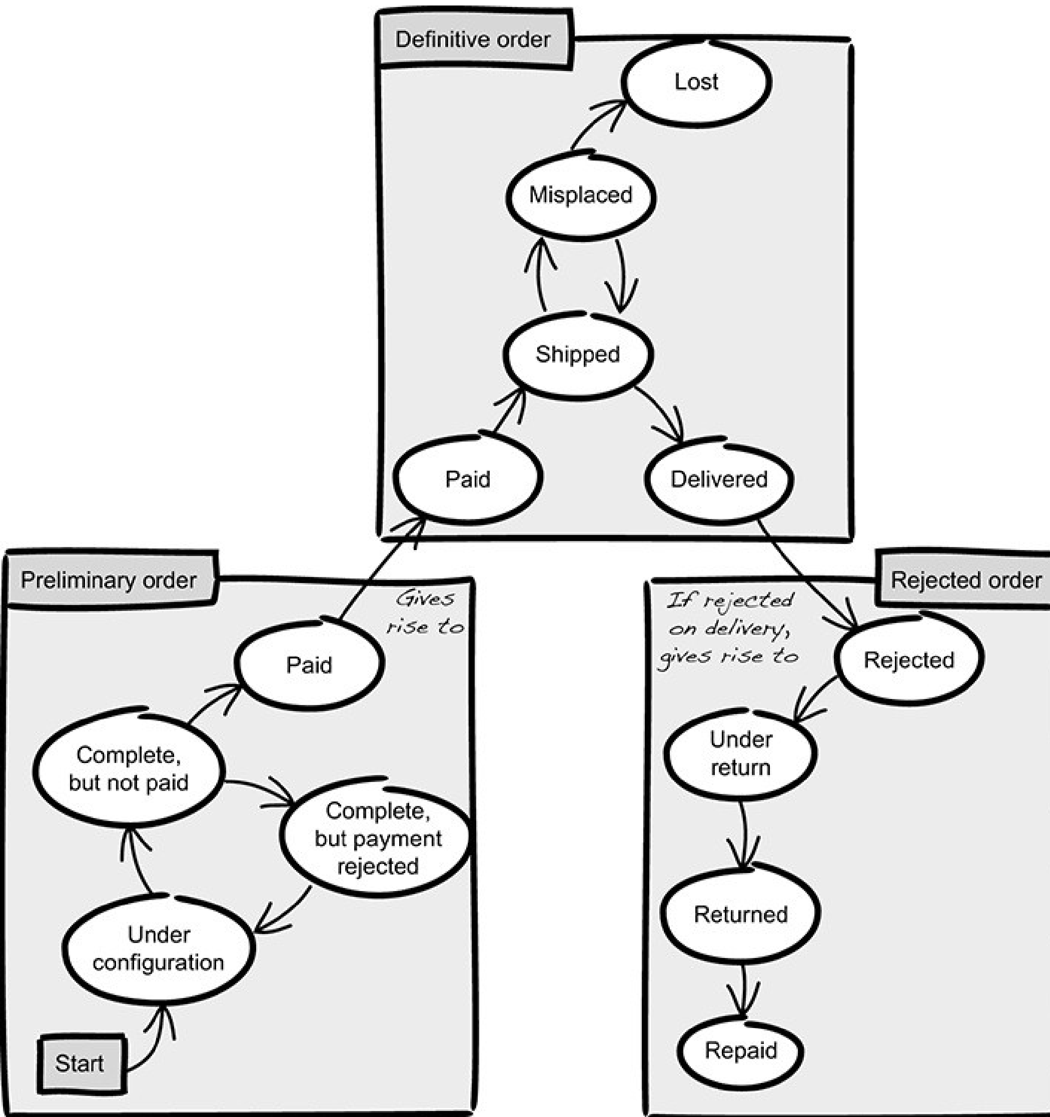
A person seen as a chain of entities,
every entity corresponding to one life phase,
with its own states



If there are points of no return,
there likely is a life phase of the entity

An order is preliminary until it is paid,
at that point it is definitive

If the shipment is rejected
by the customer,
the order enters
a third phase of its life



Let’s represent the order with 3 entities

Every entity has a manageable number of states

The 3 phases are sorted somehow

Only one transition point from one phase to the next phase (it’s also OK with 2 or 3 transition points)

Google Form

- https://forms.gle/TfoNLvUQ7QzTYPhq5

Exercise

Riprendiamo il dominio della concessionaria.

Vogliamo gestire i veicoli come delle entità in modo da consentire la modifica di tutti i loro campi. Come identifichiamo i veicoli?

Vogliamo consentire ulteriori operazioni dal menù:

- modifica di veicoli
- rimozione di veicoli
- lista di case produttrici di auto
- lista di case produttrici di moto
- lista di auto di una data casa produttrice
- lista di moto di una data casa produttrice

Fine della lezione