Documentation
The Unified Modeling Language

Paolo Burgio
paolo.burgio@unimore.it

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

High Performance
Real Time Lab

	DEVELOPERS	DESIGNERS	PROJECT MANAGERS	QA	SYSADMINS
SEEN BY DEVELOPERS					
SEEN BY DESIGNERS					
SEEN BY PROJECT MANAGERS					
SEEN BY QA					
SEEN BY SYSADMINS					

facebook.com/vitadainformatici

What is UML?

Born in 1994, standardized in 1998, official version (2.0) in 2005

A *de-facto* design language

› A semi-standard notation based on a meta-description of entities in a SW system

› Graphical notations

› Supports *divide-et-impera*

Useful because

› Can model different level of abstractions and dev phases: from specs to single classes

› Works both for top-down and bottom-up

› Language-independant

UML taxonomy

Three macro-areas

<u>Entities</u> (a.k.a.: the structure)

› Classes, interfaces

› Behavior (FSMs, interaction w/users)

› Grouping and packaging

› Notations and general information

<u>Relations</u>

› Association

› Dependency

› Generalization

› Implementation

<u>Diagrams</u>

› Same object/entity, different perspectives

› Partial representation, to "see things" under a different light

UML (standard) diagrams

Structural diagrams

› Use-cases/scenarios   } now

› Notations for classes/objects/packages/components – From OOP   } later

› Deployment/components   } won't see these

Behavioral diagrams

› Sequence diagrams

› State diagrams

› Activity diagrams

Use-case diagrams

Use case modeling

Describe the interaction between the system and other actors

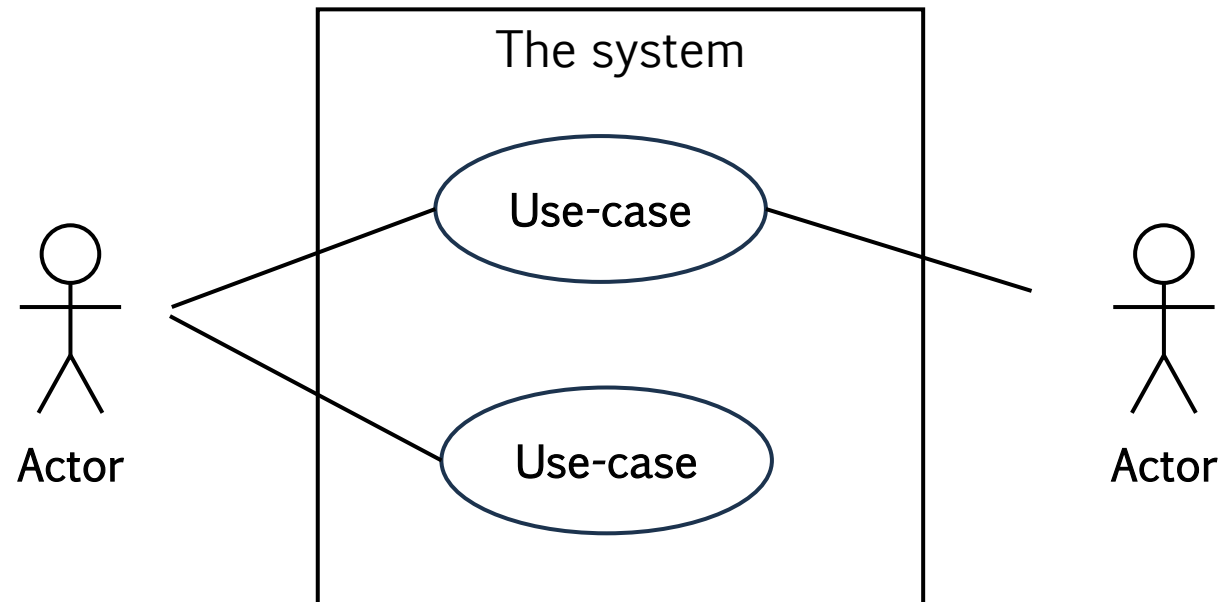› Users, other external systems,….

› Can be either a picture, or a table (or both)

These are **not** system requirements!

› We model the expected behavior

› Also, useful for functional tests/verification

We want to clearly identify

› The boundaries of the system (behavioral and "physical")

› Actors/macro-entities

› Use-case scenarios

Graphical notation

The system

Use-case

Use-case

Actor

Actor

› For every element, can/shall add a <u>brief</u> description

The right level of abstraction

First, identify the <u>Actors</u>, and model their interaction with the system

› Remember: actors are **external** to our system!

› ..even if we might model them internally, they're not under our control

Then, define multiple <u>Scenarios</u>

› They are a single instance of use-case
  – Define the (sequence of) events that happen in a specific ….scenario ☺
  – Scenario (and their events) define functional, and user assessment tests

› Distinguish between <u>main scenarios </u>(everything works correctly)

› …and <u>secondary scenarios </u>(extensions, or in case of errors)
  – «Optional parts» of the scenario
  – Can become a lot!

Scenario/UC diagrams

Define how the system interacts with the extern

› Operational conditions (also called "Operational Design Domain"), might also be standardized (see automotive)

› "Doesn't work"…but under which conditions? Which scenario?

› "It works <u>on my laptop</u>" – I <u>do not want</u> to hear this sentence anymore, ok?

We typically specify

› Pre-conditions, and post-conditions (i.e., states change)

› Guarantees both to be given, and assumed

   – Reliability, both ours and others', QoS, …

› Triggers to events

Example: Paolo's Casino

Use-case: users throwing die

› User shall be able to bet

› User shall be able to throw die
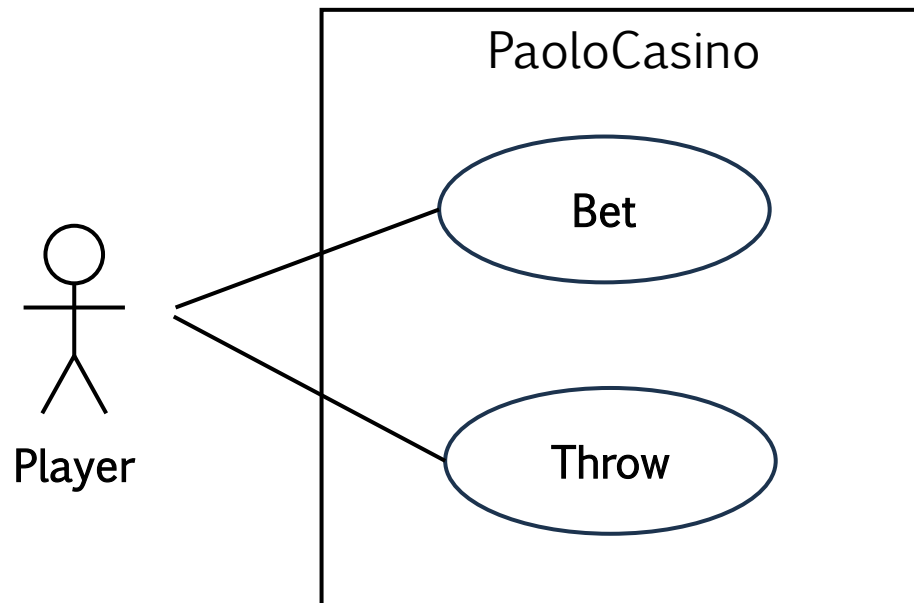
Table description

More structured, accompanies the image (or the image accompanies it)

› Unique ID

› Title

› Actors

› Pre-conditions

› Sequence of events
  – Triggers by actors
  – System responses

› Postconditions, exceptions
  – Also, state changes (e.g.., data storage in DB)

Table description (cont'd)

› Use case/scenario: UC1

› Actors: Actors 1 (starter), Actor 2

› Type: Primary, secondary, essential

› Description: ...

Actions	Responses
1. Actor 1, Action X	
2. Actor 2, Action Y	
	3. Response to Action 2
4. Actor 1, Action Z	
	5. Response to Action 4
...	...

Paolo's Casino

› Use case/scenario: [Throwing die] Bet

› Actors: Player (starter)

› Type: Essential

› Description: The player guesses a number, and places a bet with X money

Actions	Responses (Primary)
1. Player sets X money, and places the bet	
	2. The system accepts the bet
Exceptions (secondary)	
1. The number is not between 0 and 6	
2. The player does not have X money	
...	

Relations

› Association between actors and use cases

› Generalization between actors and use cases
  – Add feature/characteristics to the parent

› Inclusion of use-cases
  – <<includes>>

› Extension of use-cases
  – <<extends>>

15
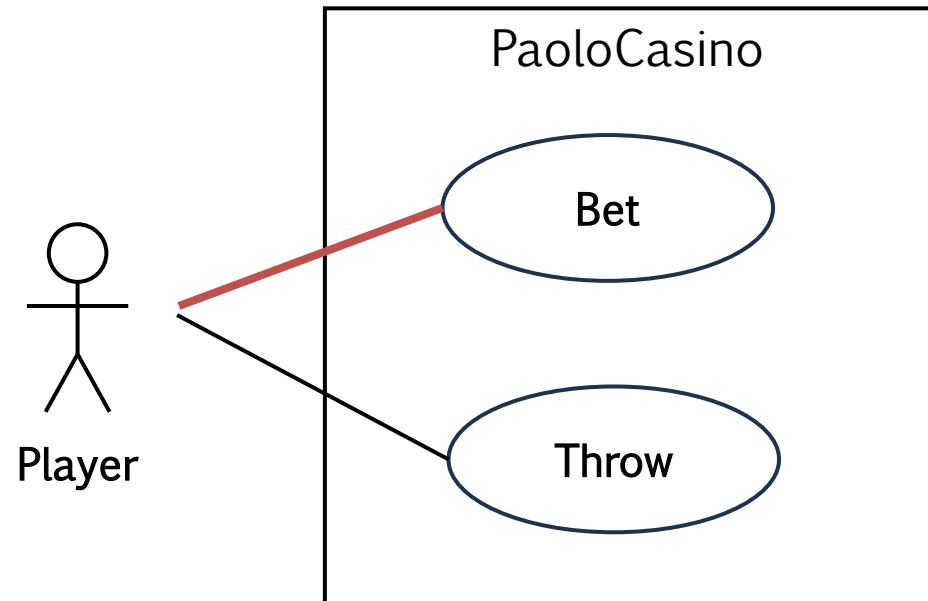
Generalization between actors

› Children can participate to all use-cases

› And add more

PaoloCasino

Bet

Throw

Edit Bet

Player
(free subscription)

Player
(paid subscription)
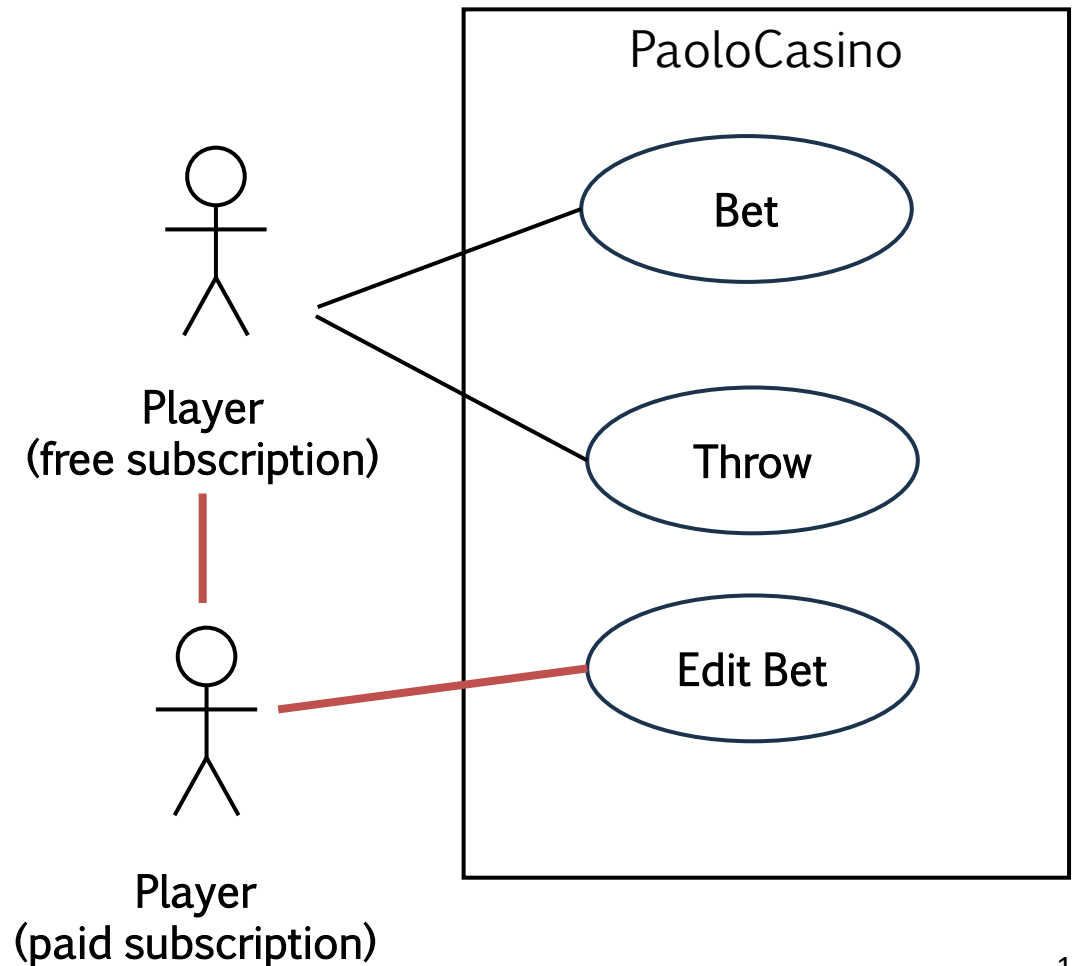
Generalization between UCs

Can re-define steps and events

› ..or specialize the existing ones
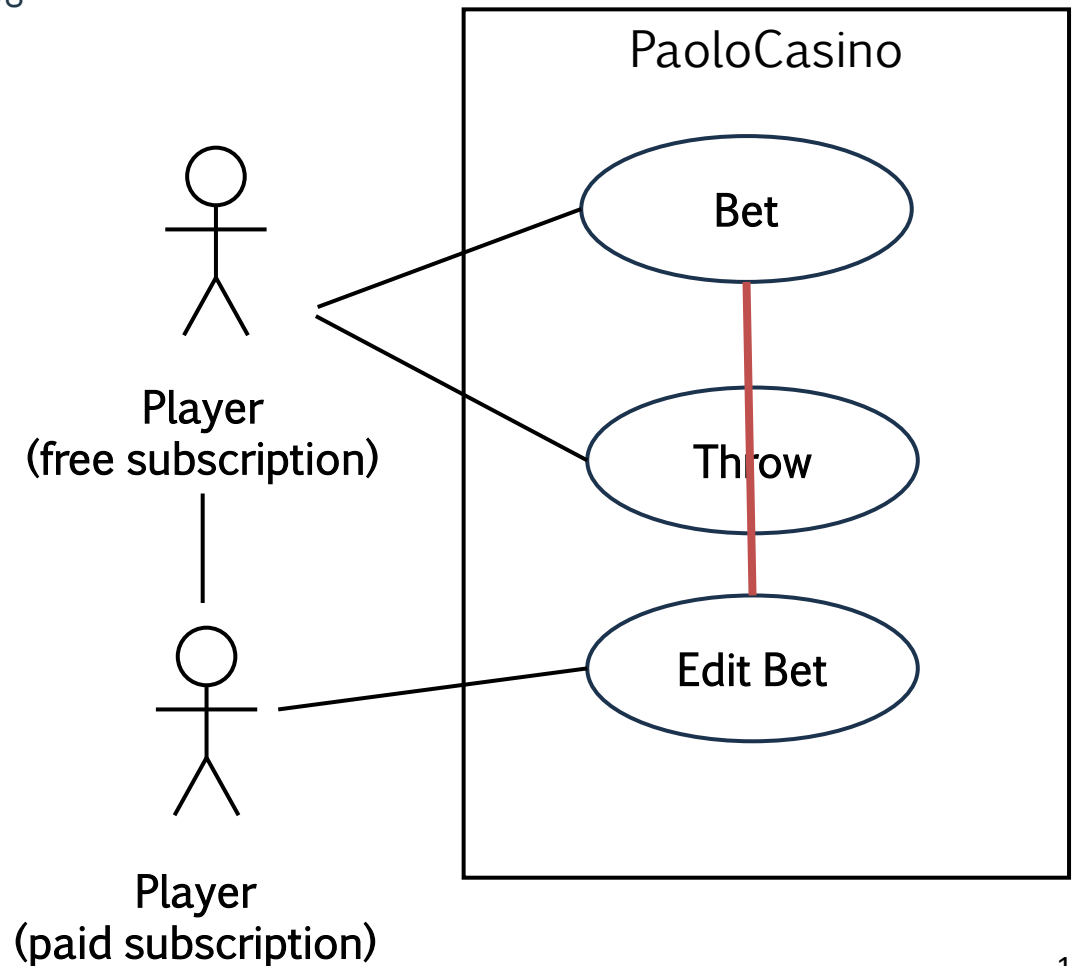
Extensions

Used to express dependencies

› First, place the bet, then throw the die

› Theorically, Player only "sees" the "throw the die" functionality (button)
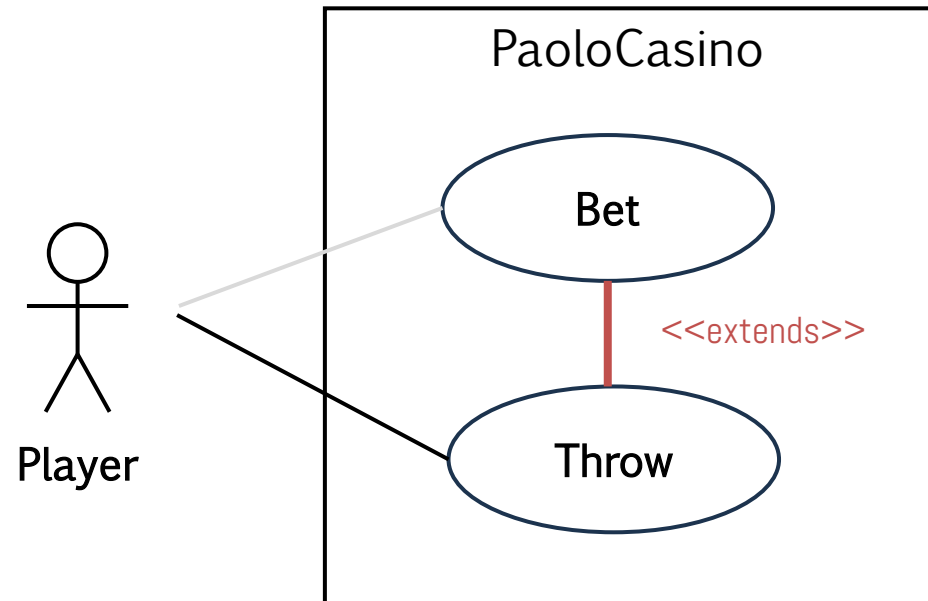
Inclusion

Used to express grouping reusage

› We can also play/bet with cards, not only with dice
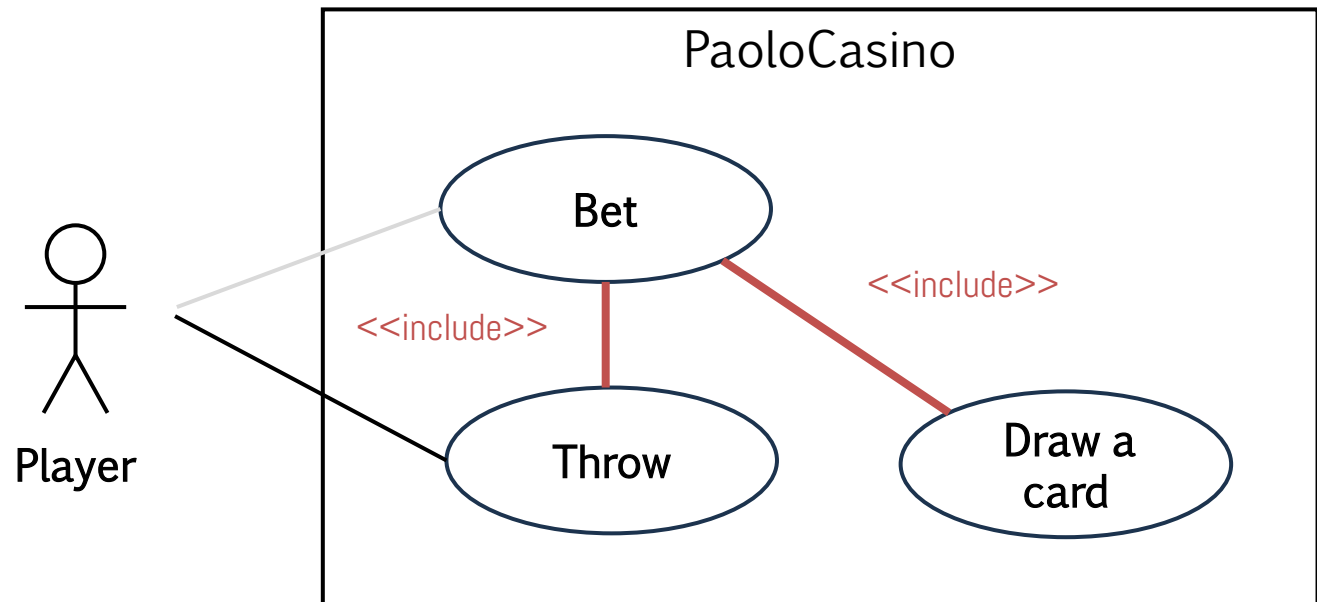
How do I identify UCs (and scenarios)?

Actor-based vs process-based identification

› For every actor, model the interaction

    vs

› For every interaction, identify the actors

Use case diagrams

› Are the starting point for system designers

› Gives a good approximation of the dimension and complexity of the system

› We can write user guides out of them!

This is an iterative process

Sequence diagrams

Sequence diagrams

› Focus on Actors, and the data they exchange with the system

› Describe the interaction, by means of Messages, objects, etc

› They describe the timeline of a scenario

Player	Bet module	Play module	Usr module	Storage

TIME

<<Place bet>>

<<Decrease credit>>

<<Update credit>>

<<Ok>>

<<Ok>>

<<Ok>>

<<Throw die >>

<<Result>>
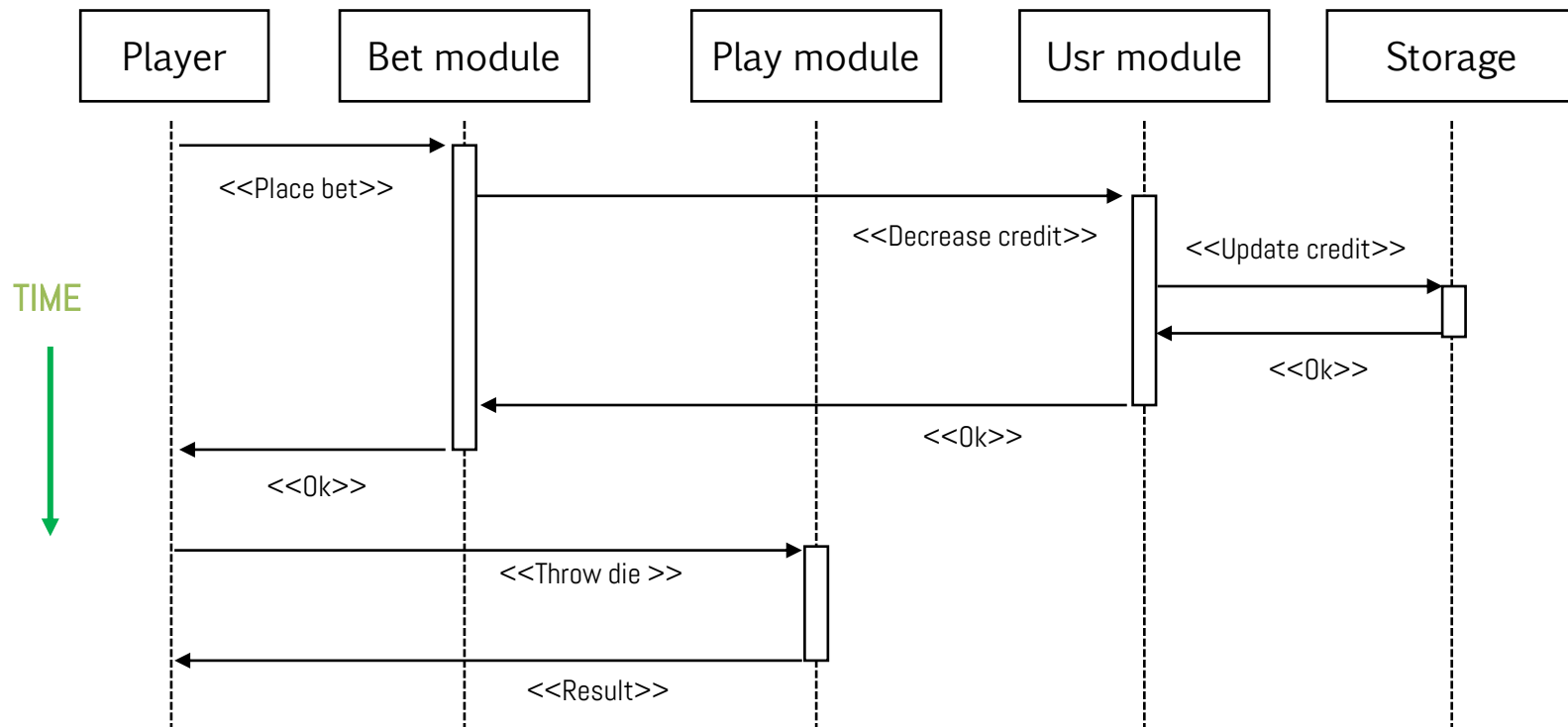
Sequence diagrams (cont'd)

Objects are entities, in our system

› Represented as a rectangle

› Can be Actors, Modules, Classes, DBs….
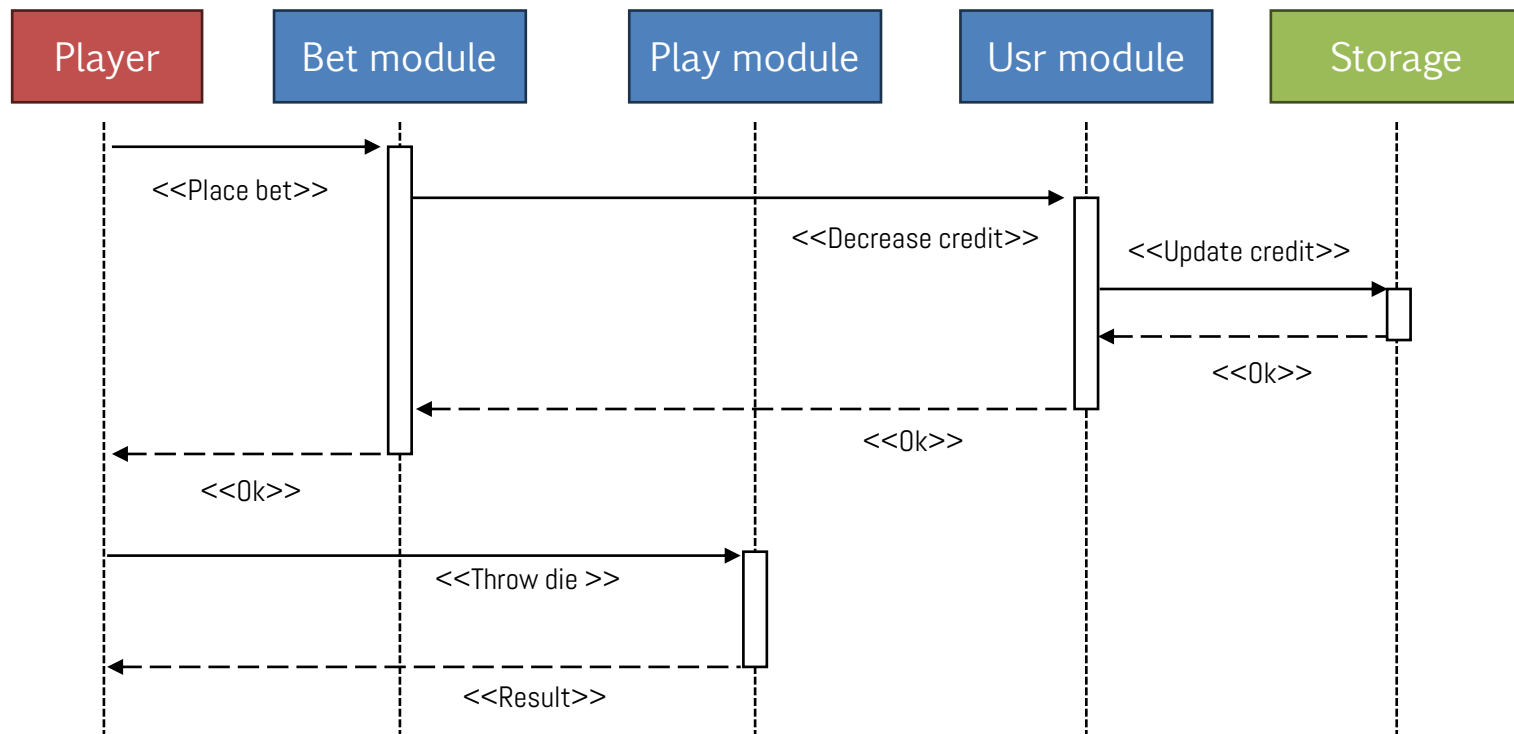
Sequence diagrams (cont'd)

Lifeline represents when an entity exists

› For classes and objects, it is extremely meaningful

› Especially, if you don't have implicit memory mgmt, e.g., non-OO languages

Goodbye!
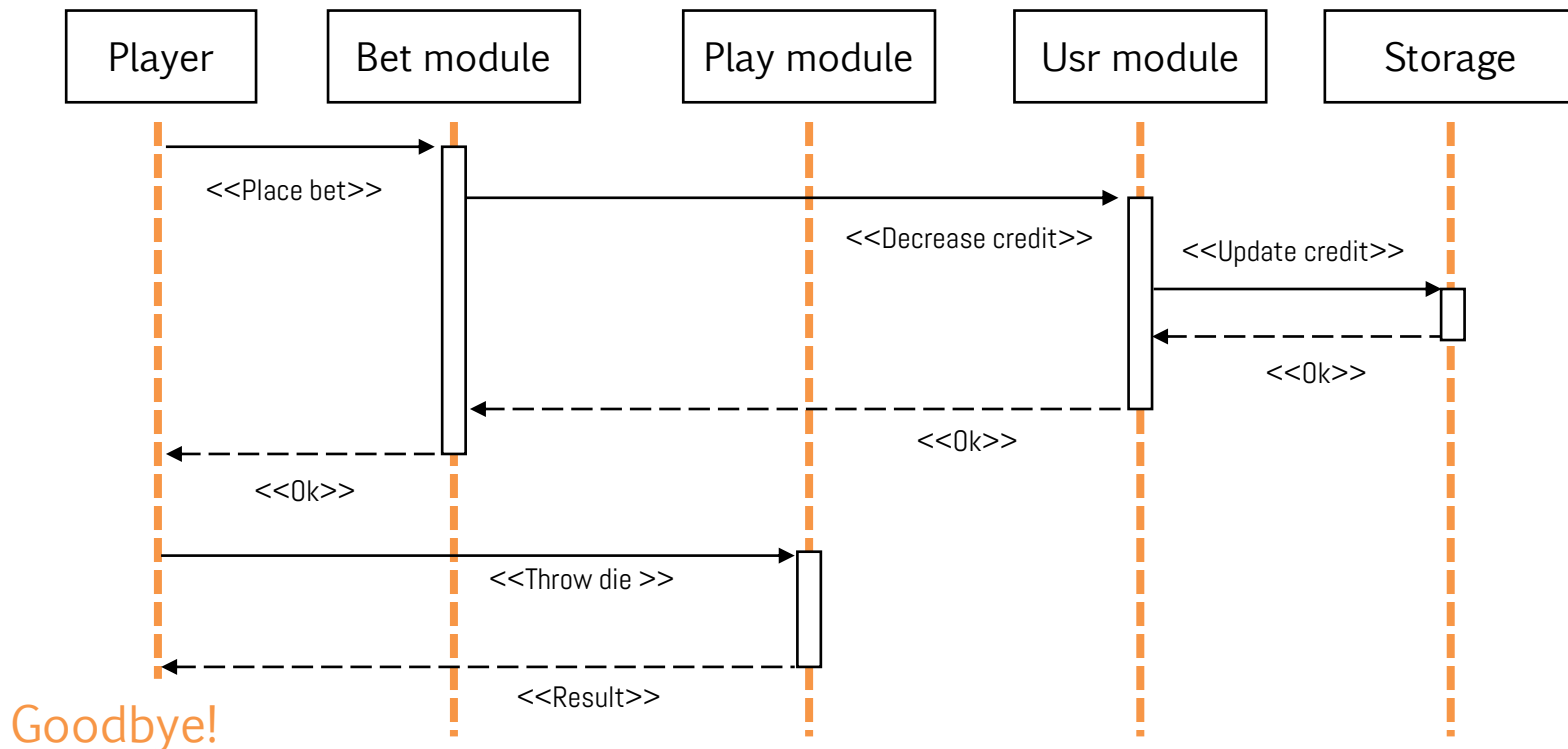
Sequence diagrams (cont'd)

Focus of control

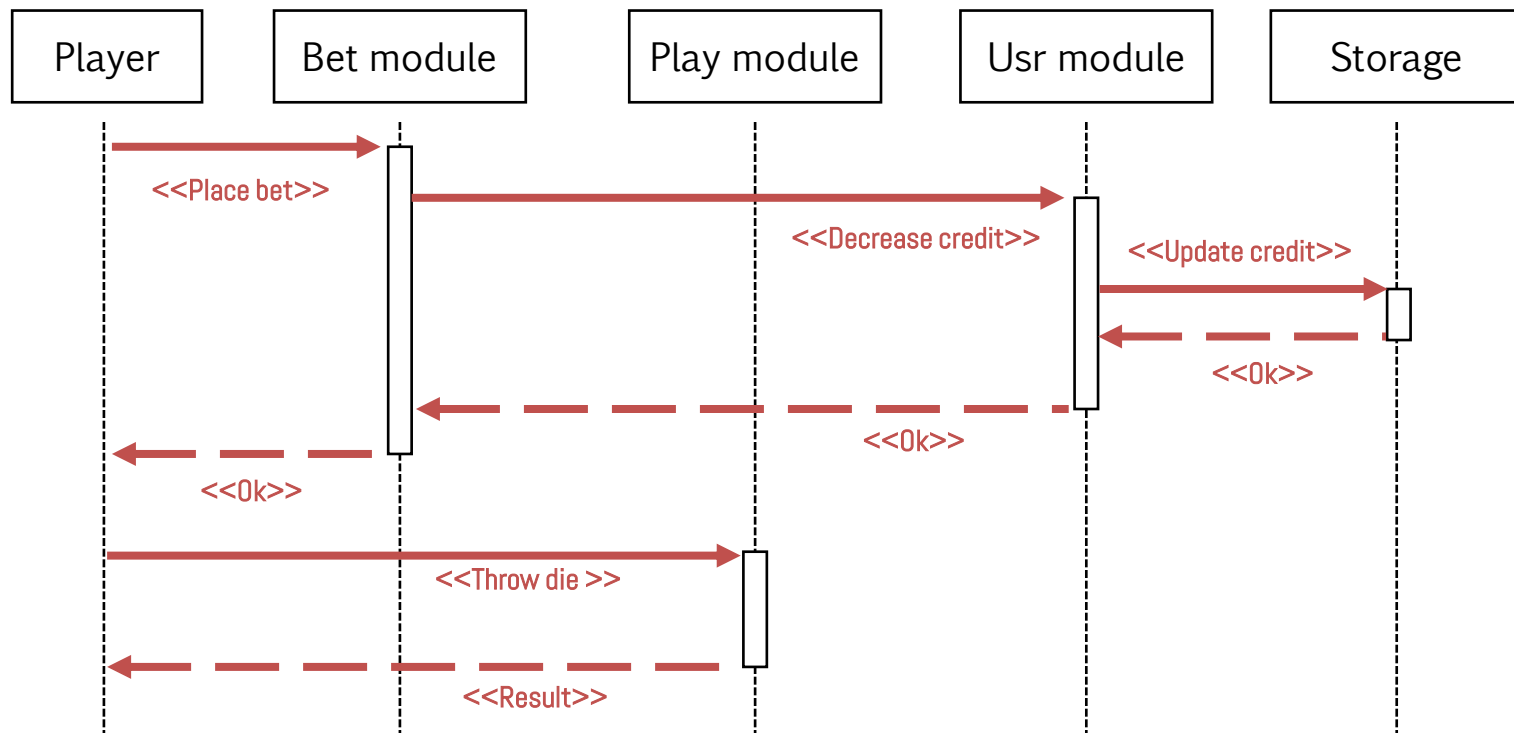› Rectangle on the lifeline

› The objects **synchronously locks** the interaction
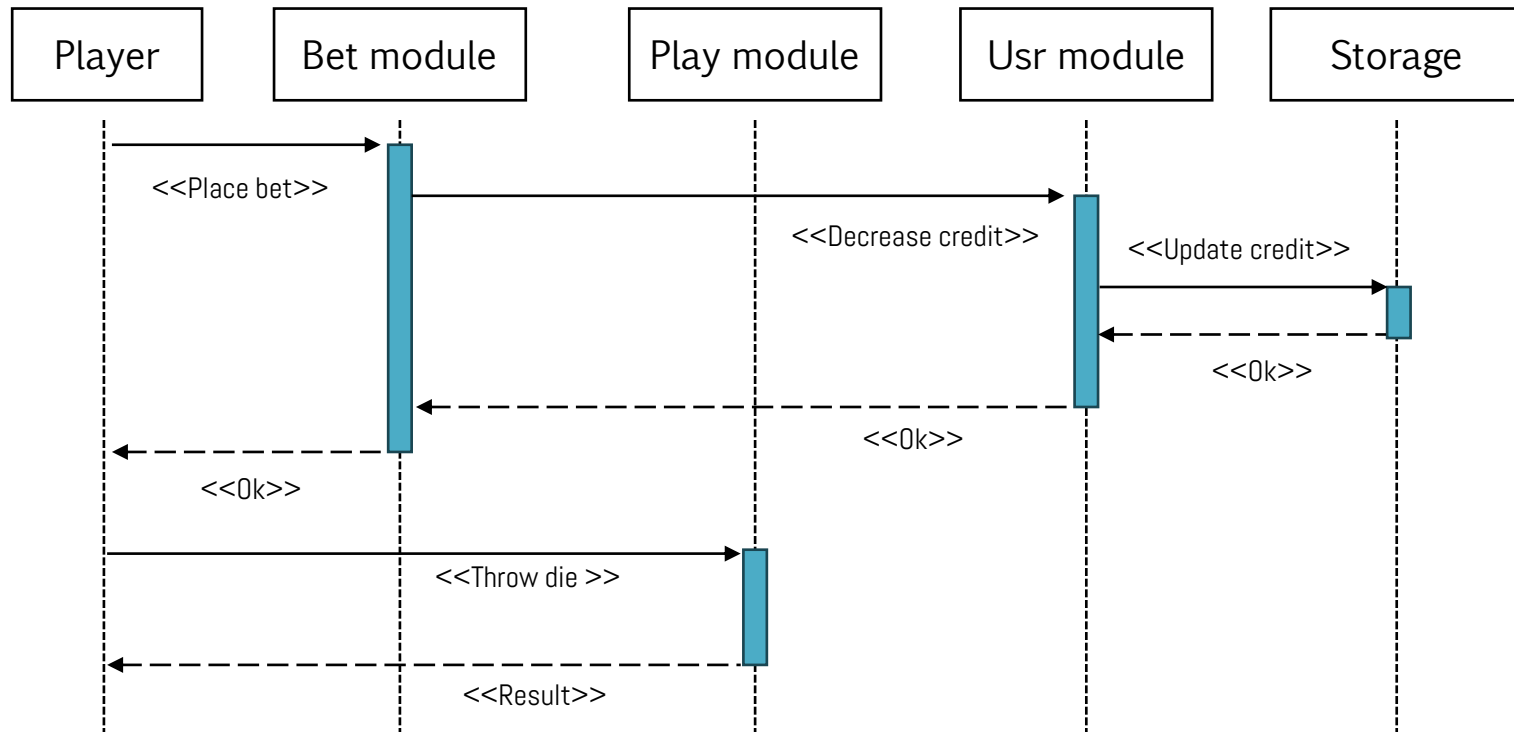
Sequence diagrams (cont'd)

Stimula

› Represent calls, invokations (with brief description)

› For synchronous calls, dotted arrows are the return/response

› Not necessary to model also the transferred data

Sequence diagrams (cont'd)

Asynchronous focus of control

› Fire and forget semantics

› Ex: update user age
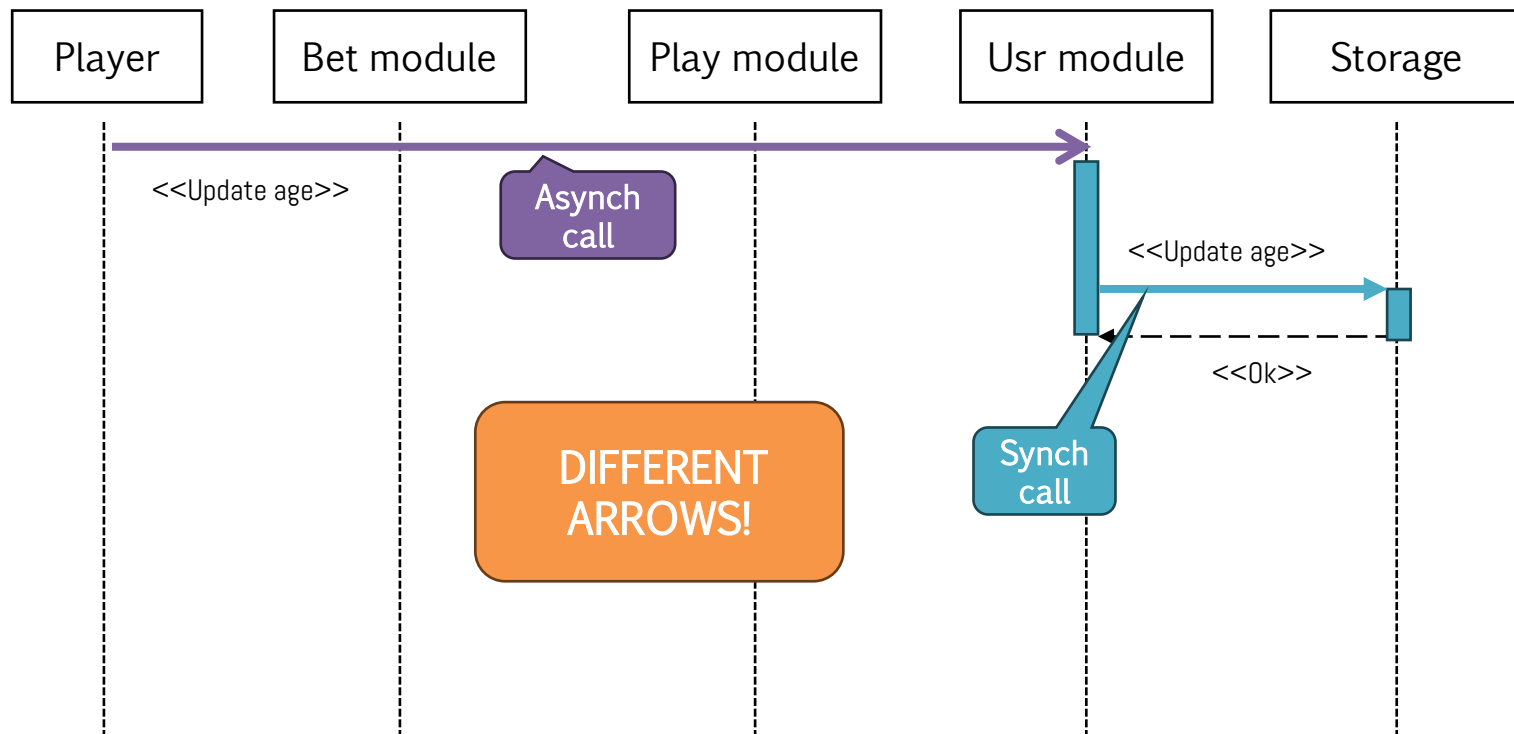
Stimula and messages

Stimula abstract messages, who represent/trigger **Actions**

› **Call/Invokation**, e.g., of a function, an endpoint etc

› **Return**

› **Send** a signal or a message

› **Create** or **destroy** objects

Message can be aggregated onto sequences

› To model complex interation / state changes

› In this case, might be useful to use numbering to explicit sequencing/ordering

<<1. Place bet>>

<<2. Throw die >>

28

Types of messages

› Constructors, destructors

 – For objects

› Read/query

› Update

› Collaborate / trigger an action

› Iterative (e.g., to specify that the entity on which we work is a list of items such as Java Arrays, Dictionaries, Lists)

› Marked with *****

Example: HTTP verbs

› POST, GET, UPDATE/PATCH, DELETE (CRUD)
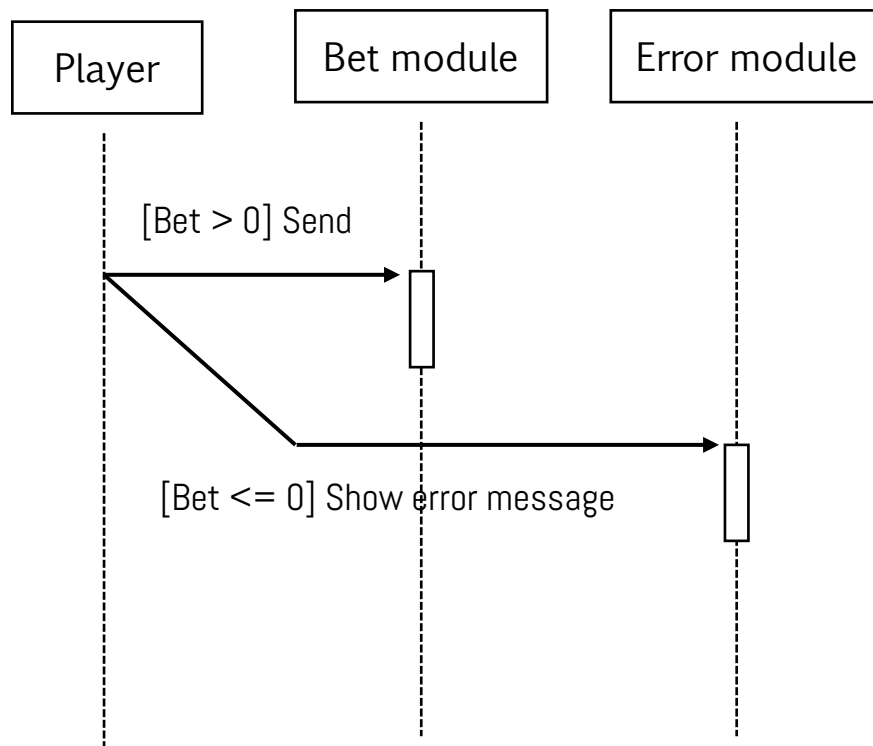
› But also PUT, OPTIONS, …

Homework

Write the sequential diagram of a simple web server

› That supports all HTTP verbs on a sample endpoint

› https://<myserviceurl>/me

› E.g., handle user infos

Conditional execution

› IFs are represented as []
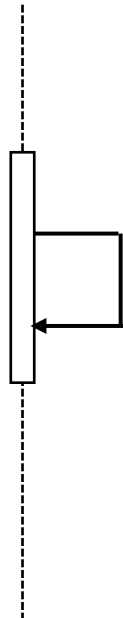
Loops

Loops are represented as *[], and squares

› E.g., Place multiple bets by a list
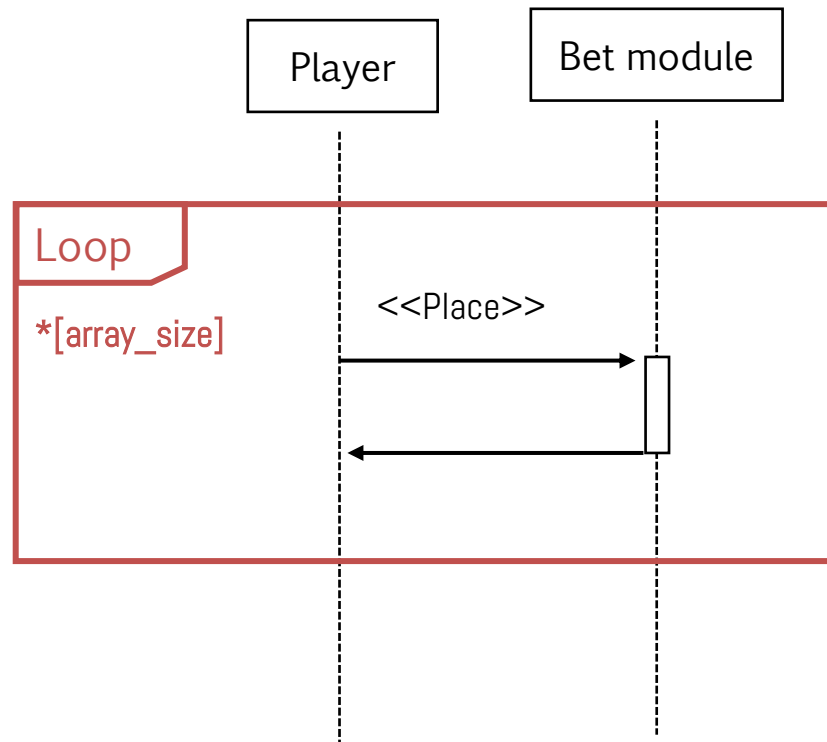
Recursion

State diagrams

Stateful entities

State diagram to represent the lifecycle of an entity, typically classes/objects, by means of

› Events and Actions

› States

› Transitions

› Guards

State defined as:

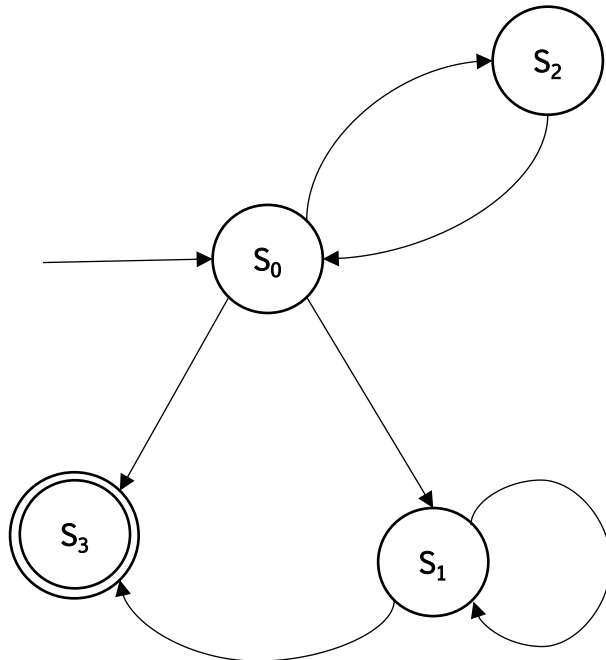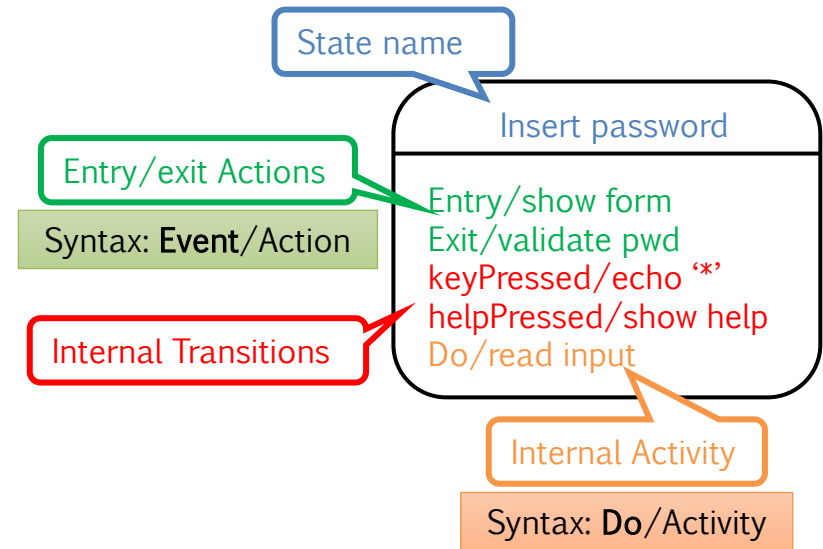*a consistent, meaningful set of attributes of an entity/object, that affects its behavior*

› Initial and end pseudo-states

› Initial state after creation/boot, etc

› End state might not exist (for persistent entities)

Graphical representation
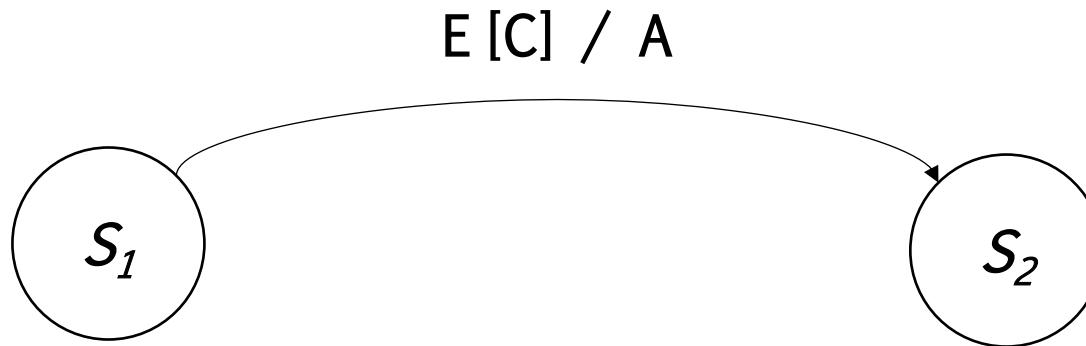
By standard, a rounded square

› fill with useful information/state/behavior

› Example: (simple) password insertion form

State name

Insert password

Entry/exit Actions

Syntax: **Event**/Action

Entry/show form
Exit/validate pwd
keyPressed/echo '*'
helpPressed/show help
Do/read input

Internal Transitions

Internal Activity

Syntax: **Do**/Activity

Or (non-standard) circles/ovals

› Useful when we don't have many details

› System-wide

$$E \, [C] \, / \, A$$

This means:

› If we are in State S1

› And Event/Trigger E happens

› If Condition/Guard C is satisfied

› Then we execute Action A, and we enter State S2

Transitions and guards

event [parameters] [guard] / action

› Events can have <u>parameters</u>

› Guards are conditions that "block" the transition, if not satisfied

› Action is a (typically small) piece of computation task that is executed as soon as transition is triggered

– States can also contain (more complex) activities!

Events

› Trigger state transitions

› In Sequence Diagrams, modeled as Messages
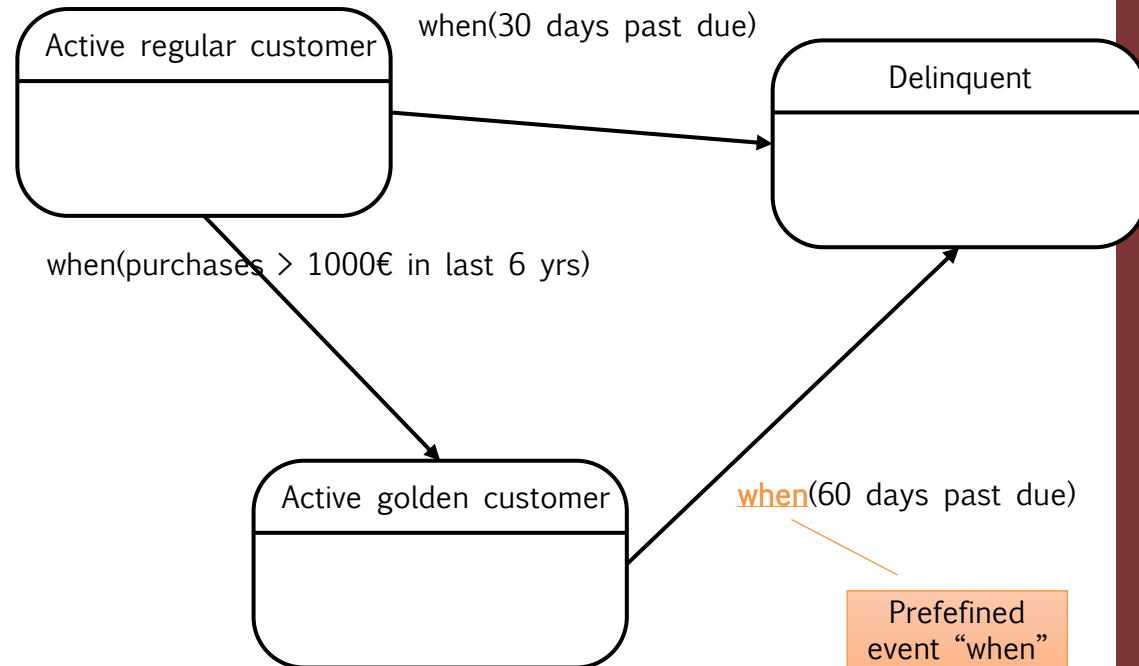
› Together with States, trigger object responses (Mealy vs. Moore...)

```
┌─────────────────────────────────────────────────────────────┐
│                                                               │
│   ┌──────────────┐      placeBet()      ┌──────────────┐      │
│   │ $$ allocated │ ──────────────────▶  │ In progress  │      │
│   │              │                      │              │      │
│   └──────────────┘                      └──────────────┘      │
│                                                               │
└─────────────────────────────────────────────────────────────┘
```
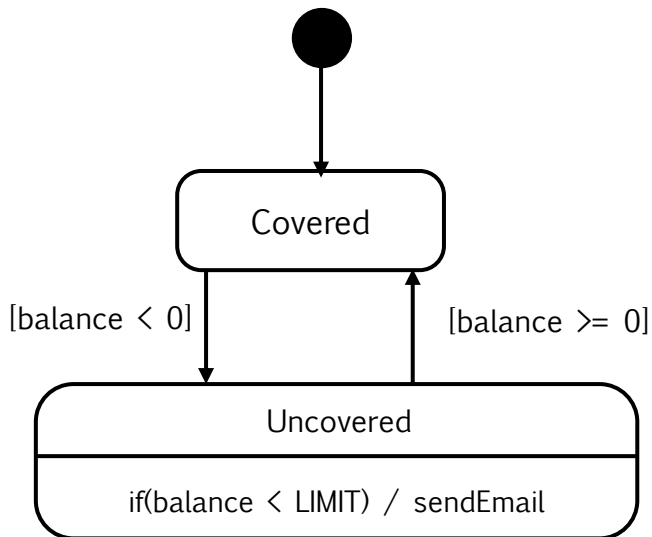
They can be:

› Synchronoous method invokation ("call")

› Asynchronoous method invokation ("signal")
  – ex: throw exception, interrupts

› A condition turning true or false ("change event")

› An expired timer/counter ("elapsed time event")

Change events

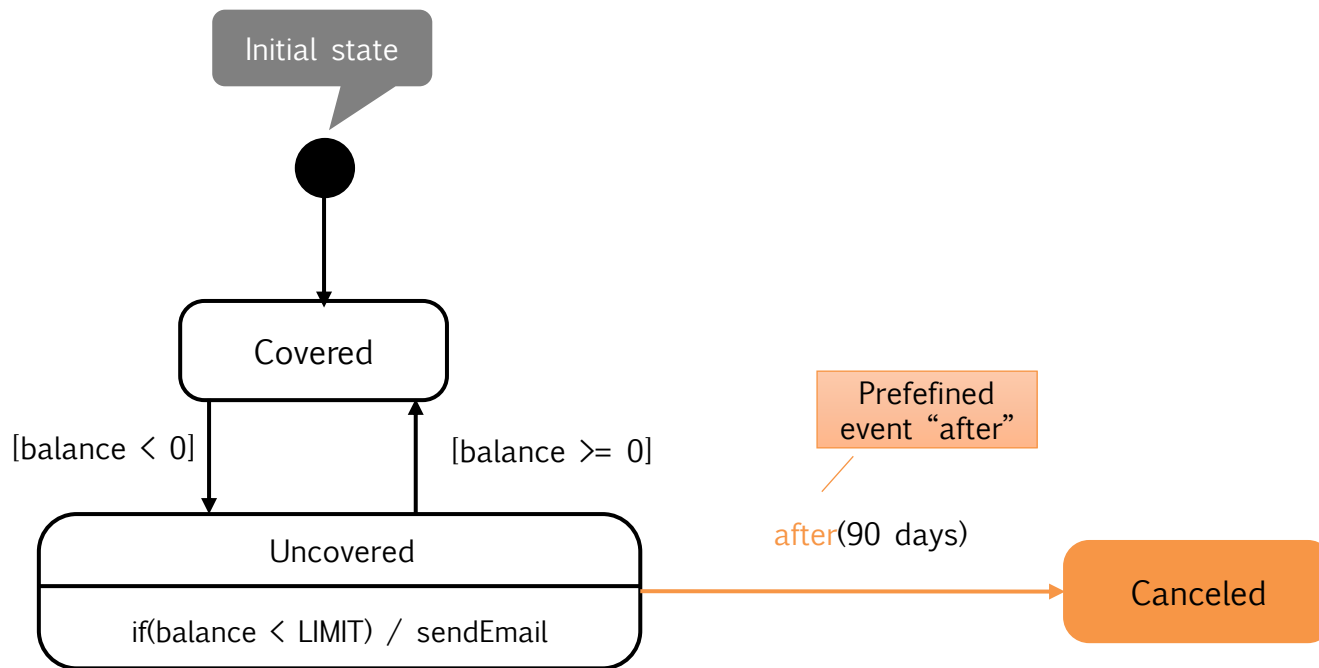Example: Customer changes its status depending on arrears and purchases

Active regular customer

when(30 days past due)

Delinquent

when(purchases > 1000€ in last 6 yrs)

Active golden customer

when(60 days past due)

Prefefined event "when"

Covered

[balance < 0]

[balance >= 0]

Uncovered

if(balance < LIMIT) / sendEmail

Example: a request for purchase can change its status, and being blocked if uncovered

Time events

> Cont'd

Initial state

Covered

[balance < 0]   [balance >= 0]

Uncovered

if(balance < LIMIT) / sendEmail

Prefefined event "after"

after(90 days)

Canceled

Activities and Actions

› Inside states, there can be

Activities

› Non-atomic

› (Typically) they do not alter object state

Actions

› Smaller, atomic

› They alter the object state

Actions

Entry: executed as soon as the object enters a given state

› entry/action name

Exit: executed as soon ase the object exits from a State, due to a transition

› exit/action name

Do: executed as soon as the object is in a State

› do/action name

Include: invokes a «submachine», another state diagram

Event: actions happen as a response to an Event/trigger

Entry/exit Actions

Syntax: **Event**/Action

Internal Transitions

Insert password

Entry/show form
Exit/validate pwd
keyPressed/echo '*'
helpPressed/show help
Do/read input

Internal Activity

Syntax: **Do**/Activity
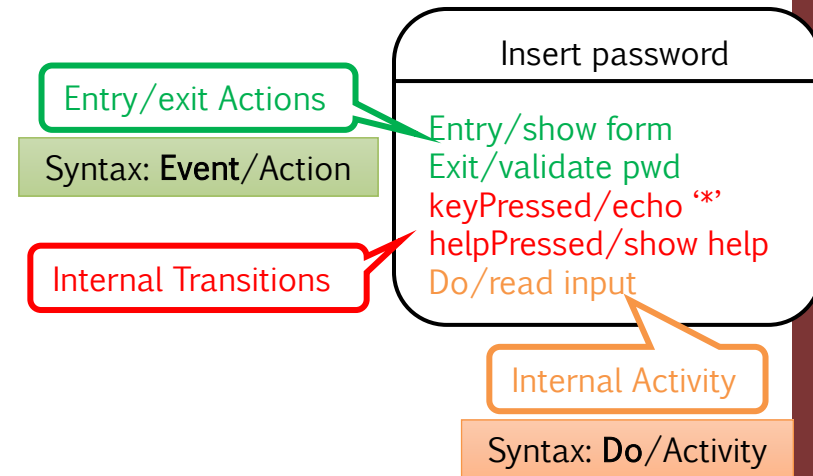
Actions: sequence

When an event triggers a transition

› Executing activities are interrupted ("gracefully", we hope…)

› Run the Exit action of the «old» state

› Run the Event action

› Run the Entry action of the «new» state

› Run Do actions of the «new» state

Actions: purchase vs. cancellation
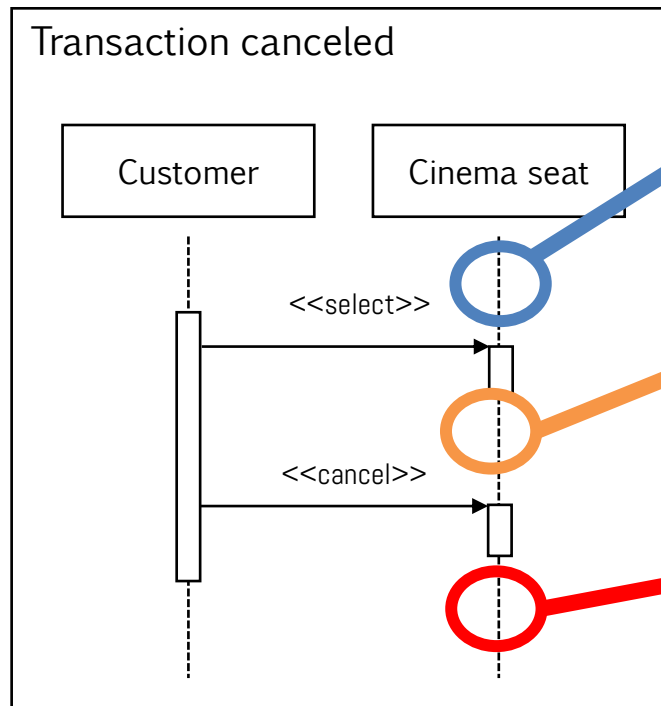
› 2 scenarios (modeled in sequence diagrams)
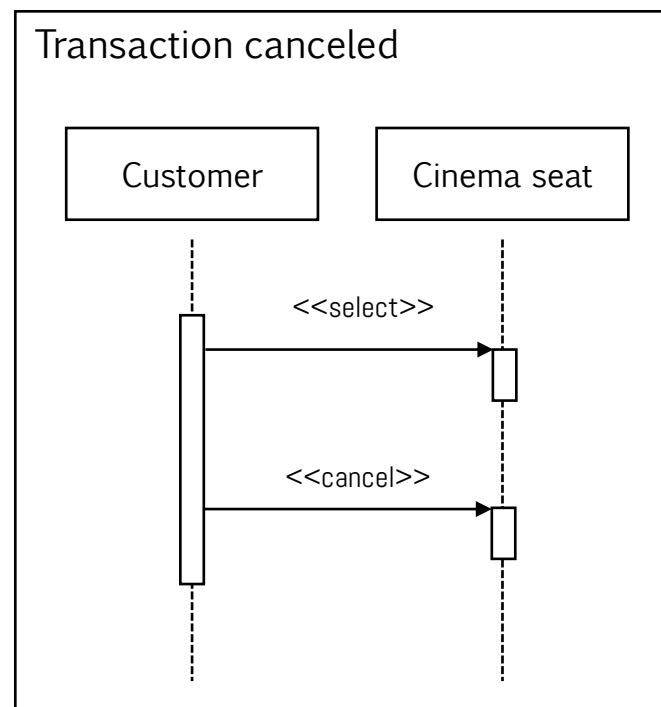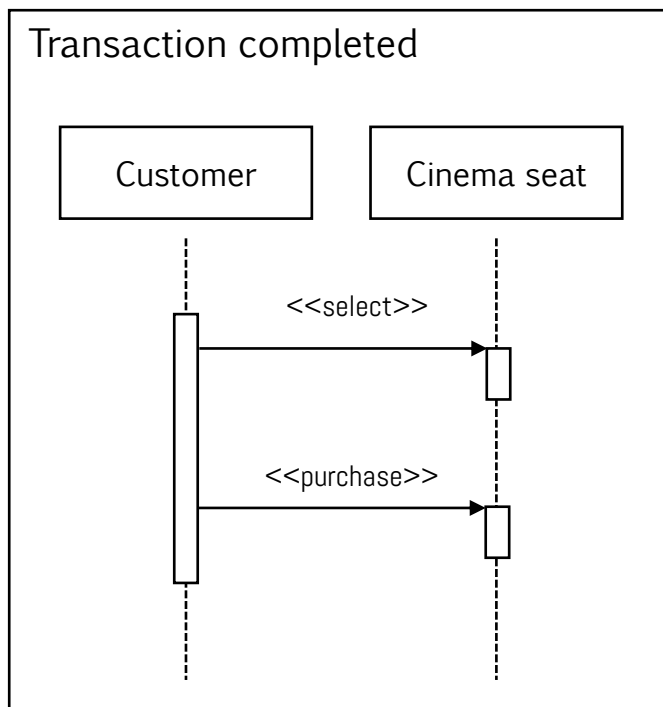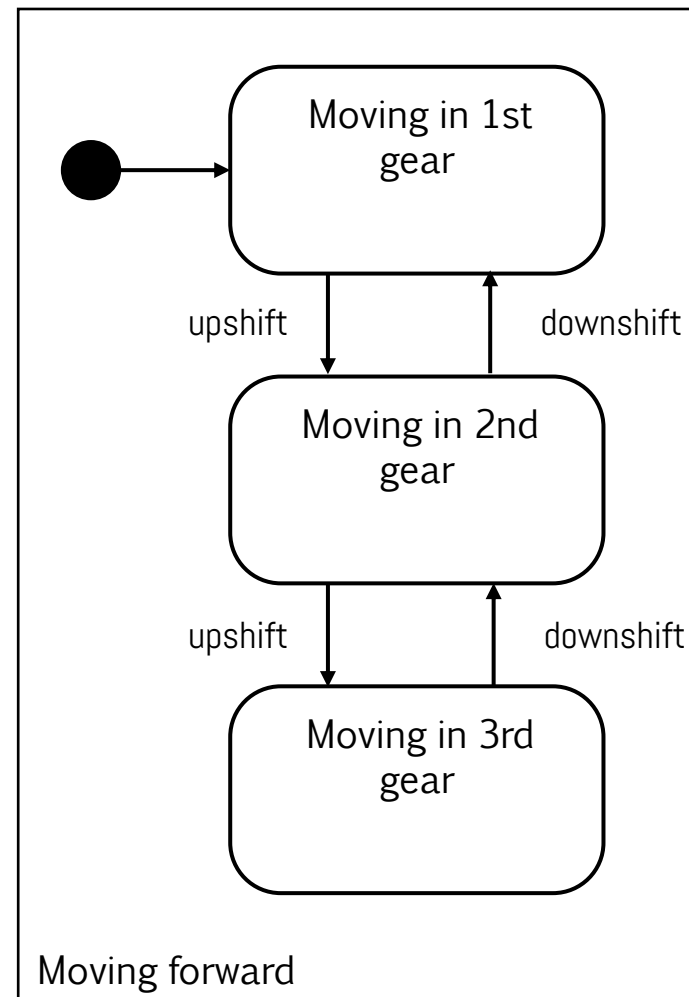
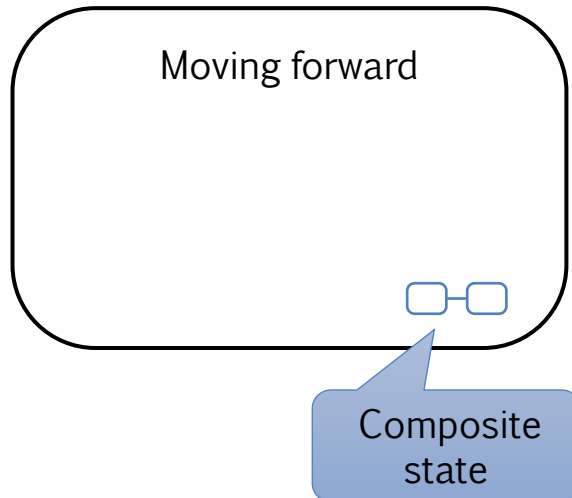# Actions: success scenario

› Refer to State Diagram

Transaction completed

Customer

Cinema seat

<<select>>

<<purchase>>

Priced – Not selected – Not sold

<<select>>

Priced - Selected – Not sold

<<purchase>>

Priced - Not selected – Sold

Transaction events and states
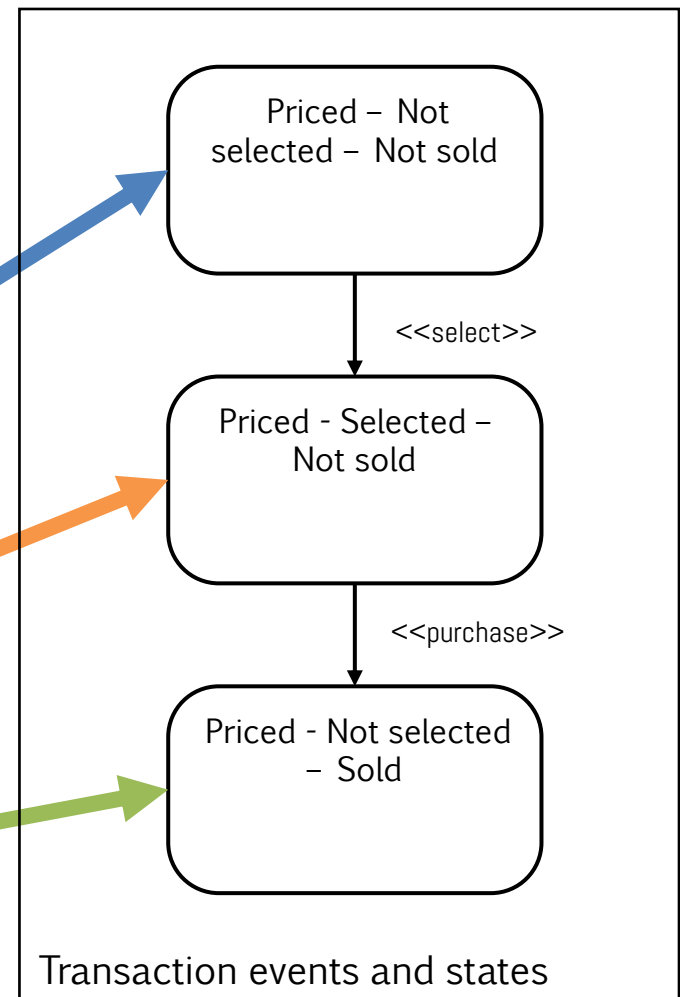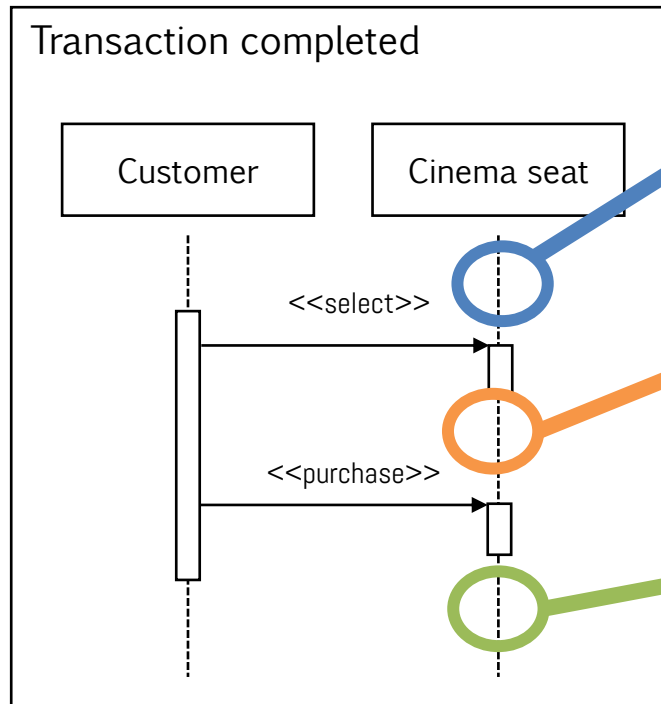
Action: cancellation scenario

> Refer to State Diagram

Concurrent states

Moving forward

Composite state

Moving in 1st gear

upshift   downshift

Moving in 2nd gear

upshift   downshift

Moving in 3rd gear

Moving forward
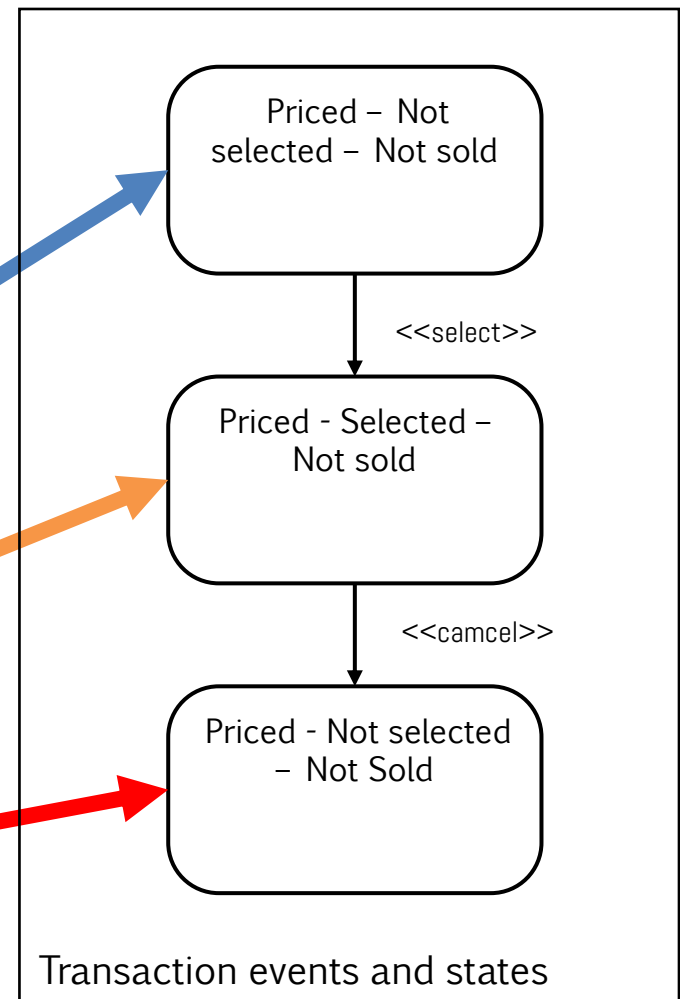
› Or decomposition (of sub-states)
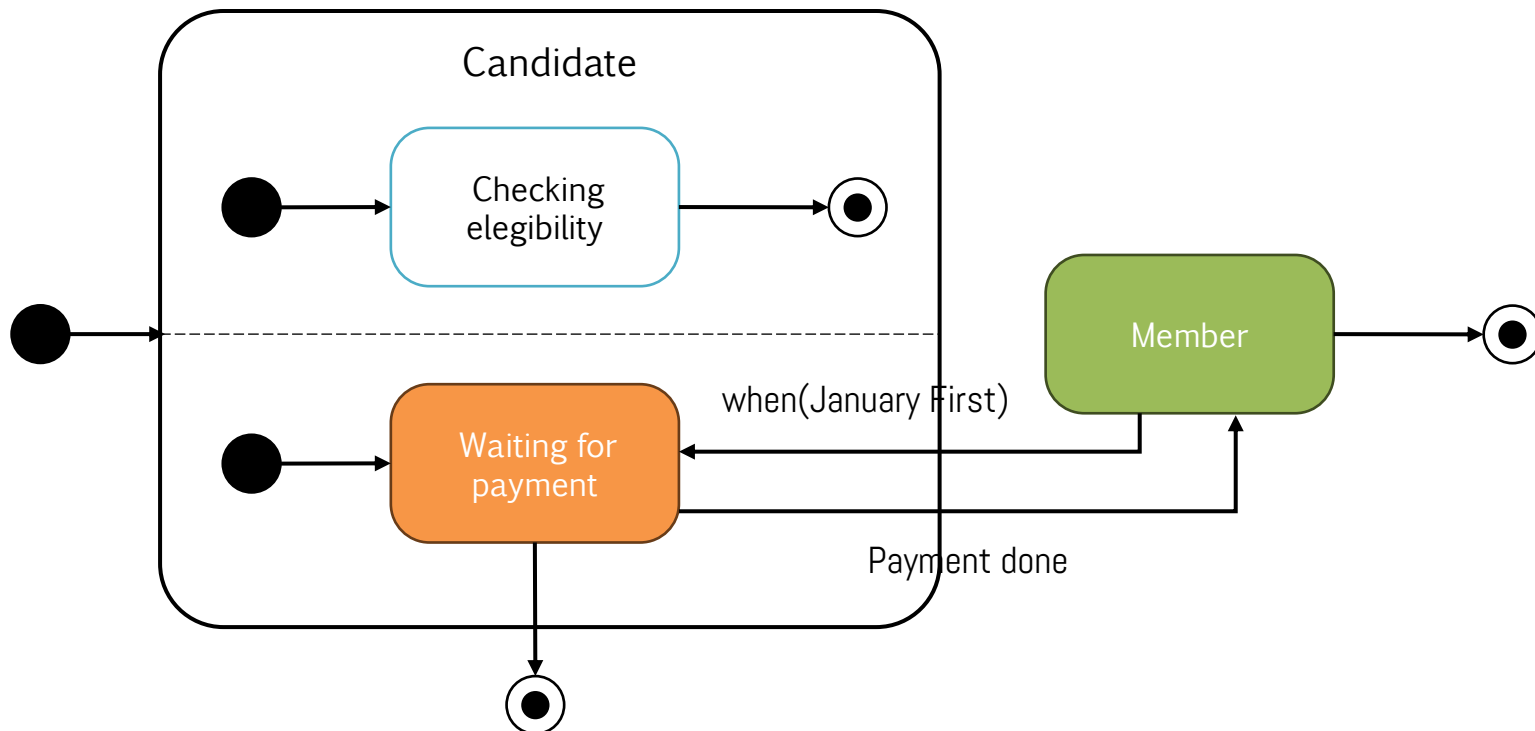
Concurrent states
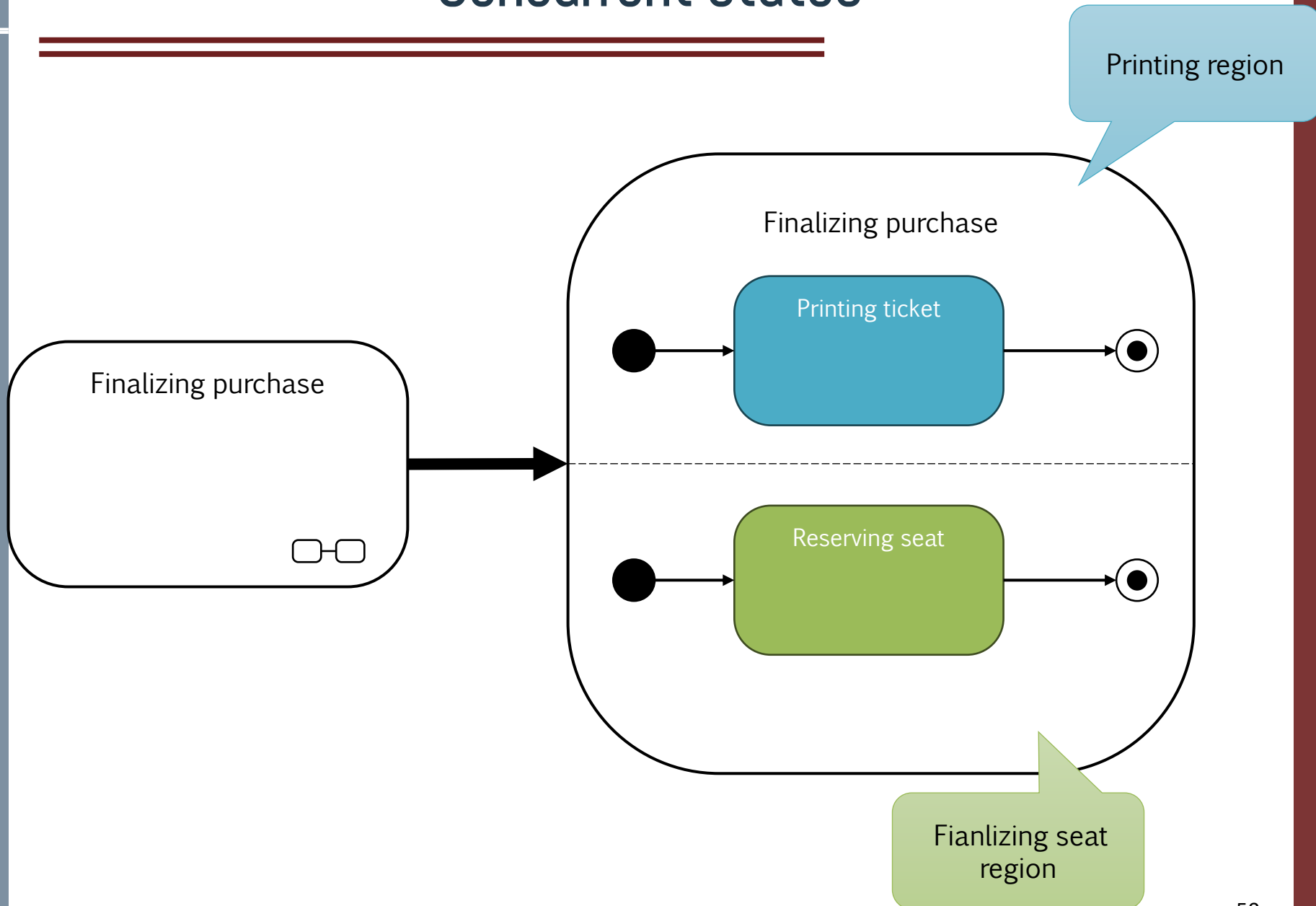
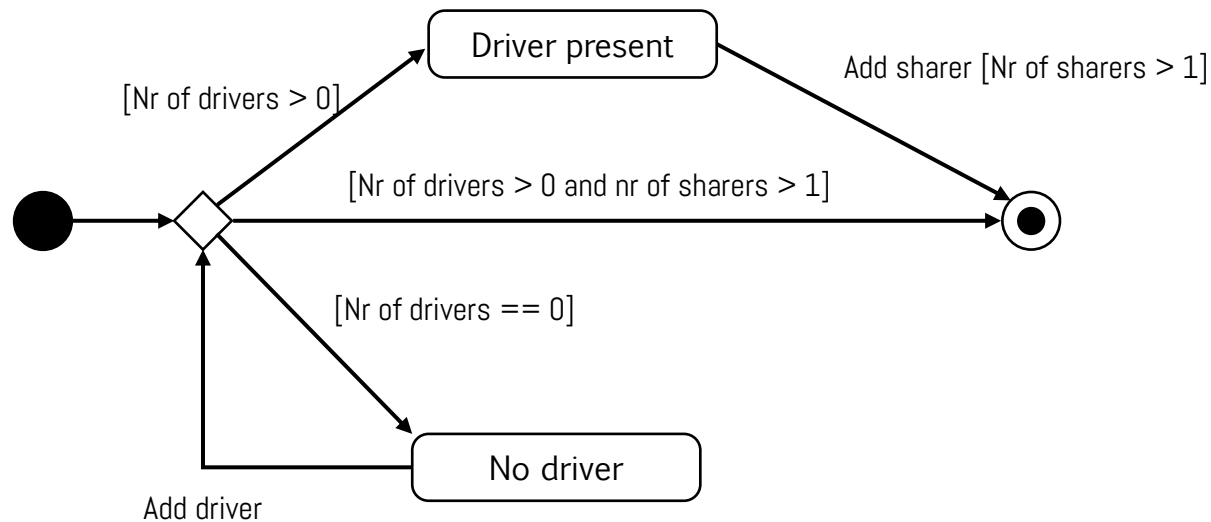# Concurrent states

> Here, a person candidates for an association
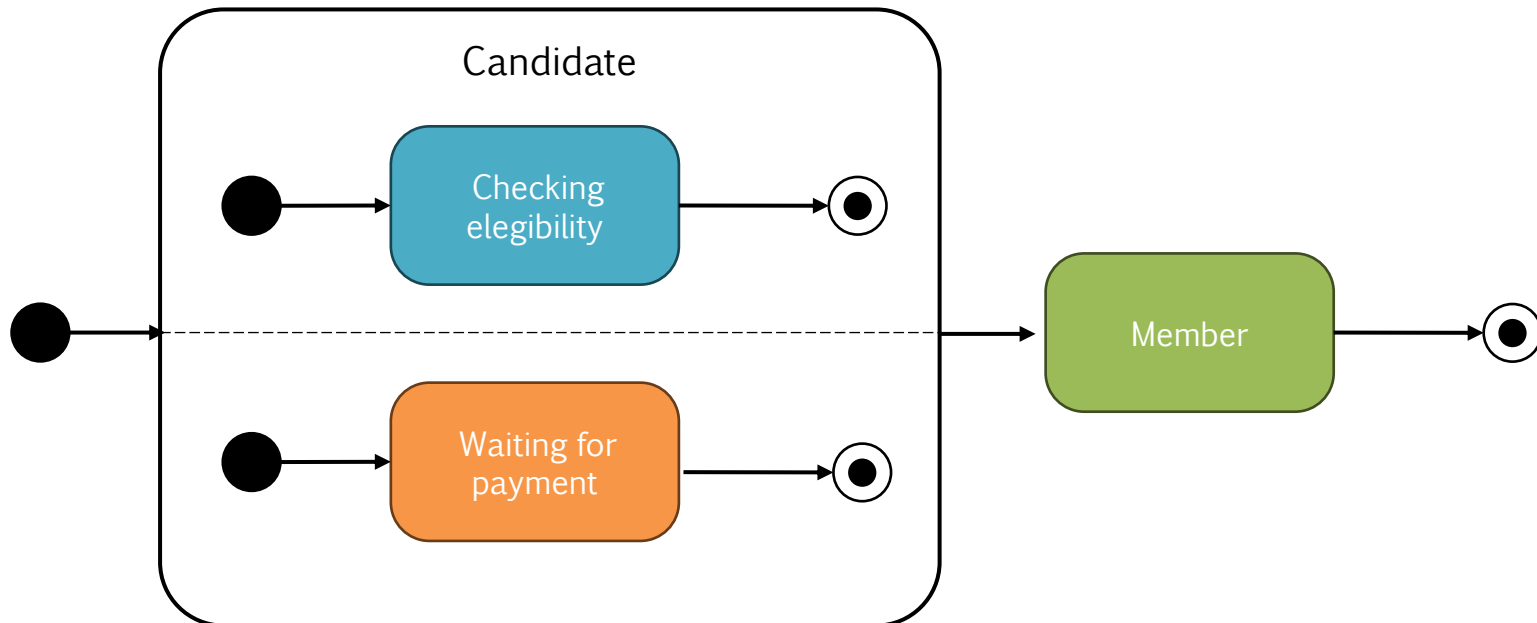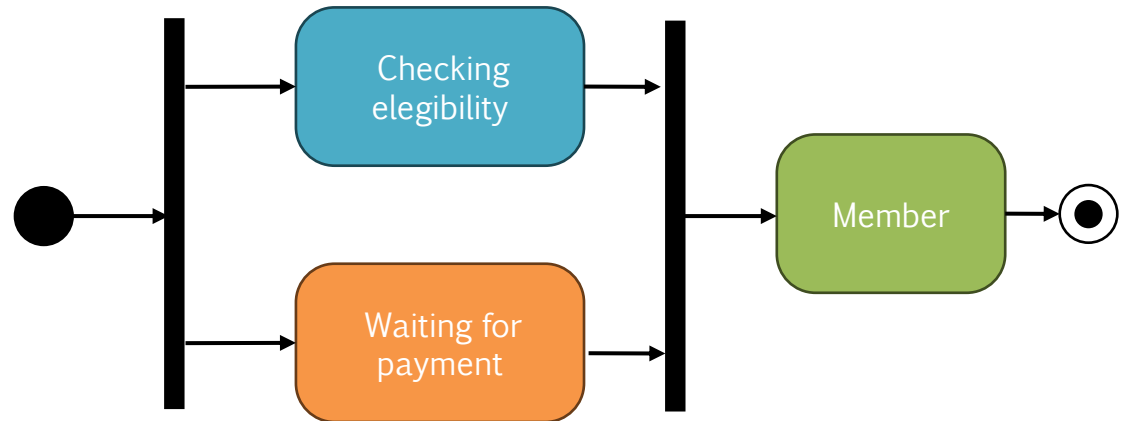
Composite states

› Here, a person renews the subscription to entering an association

Conditional branching

› Same trigger/event, different conditions, bring to different states

› Example: car sharing

Driver present

[Nr of drivers > 0]

Add sharer [Nr of sharers > 1]

[Nr of drivers > 0 and nr of sharers > 1]

[Nr of drivers == 0]

Add driver

No driver

Activity diagrams

Activity Diagrams in a nutshell

Model the behavior of any entity in the system, for every state

› "Object-Oriented" flow diagrams

› They include (timing and spatial) dependencies

› Give more detail of activities (and sub-activities) happen in every state, under inputs

Actions, and transitions
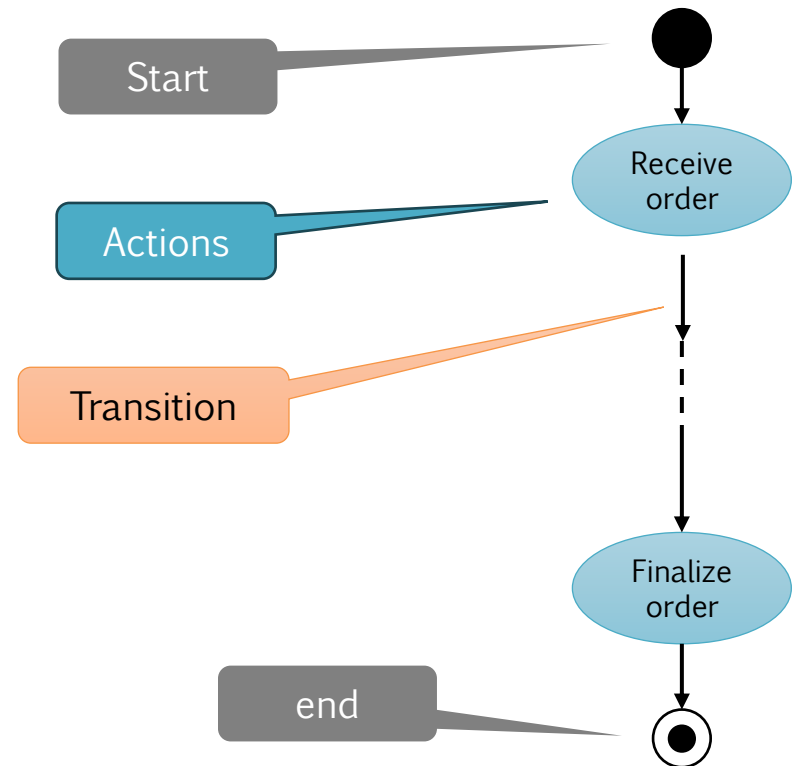
Action states are atomic activities

> Cannot be split

> Cannot be interrupted (preempted)

> (In the model) happen instantly

Special states

> Start/end state

Transition

> When Action states ends

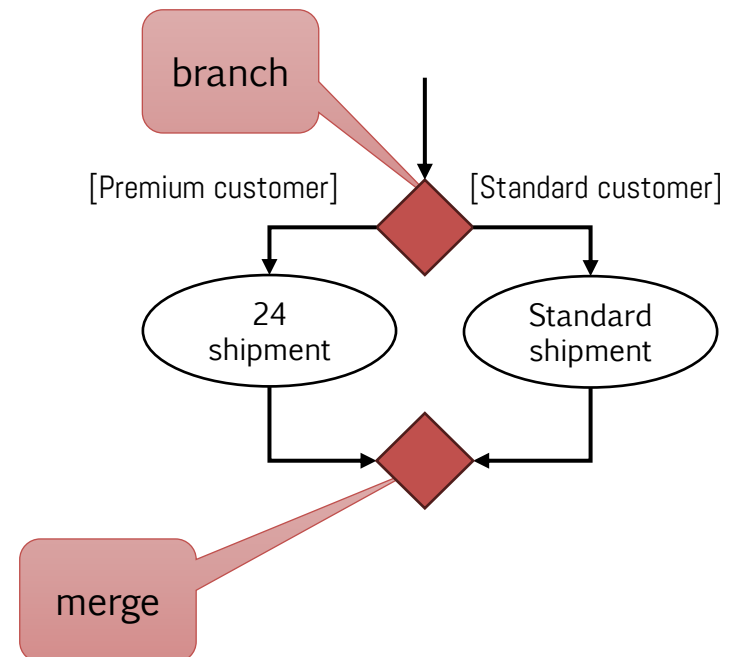# Branching/Merging

Branching based on condition

› Captures IF-ELSE semantics

› Rhombus: one-to-many

Merging

› Rhombus: many-to-one

Important

› You do not have to model the whole application!

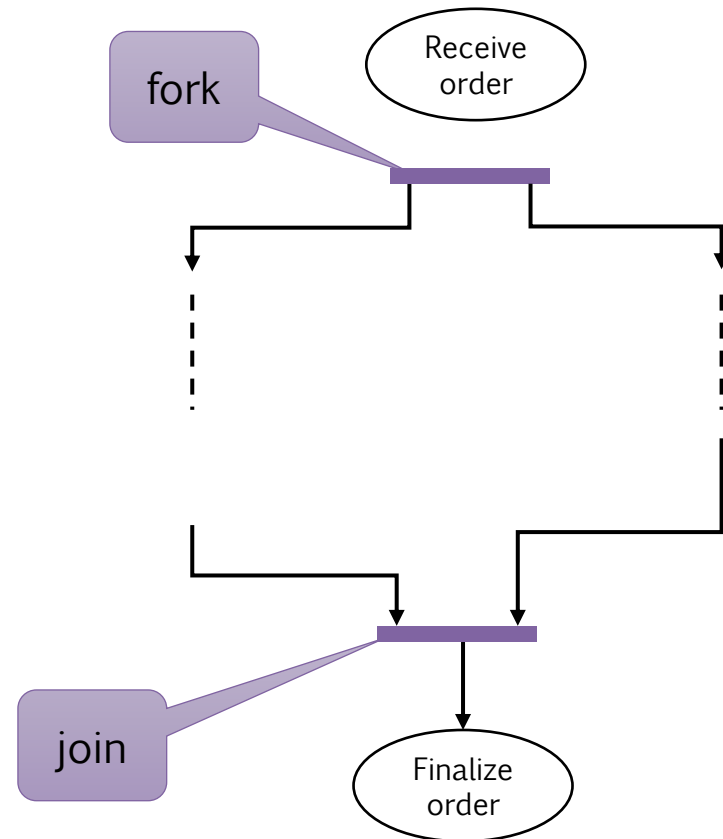› Choosing the right abstraction level, lets you cut the clutter
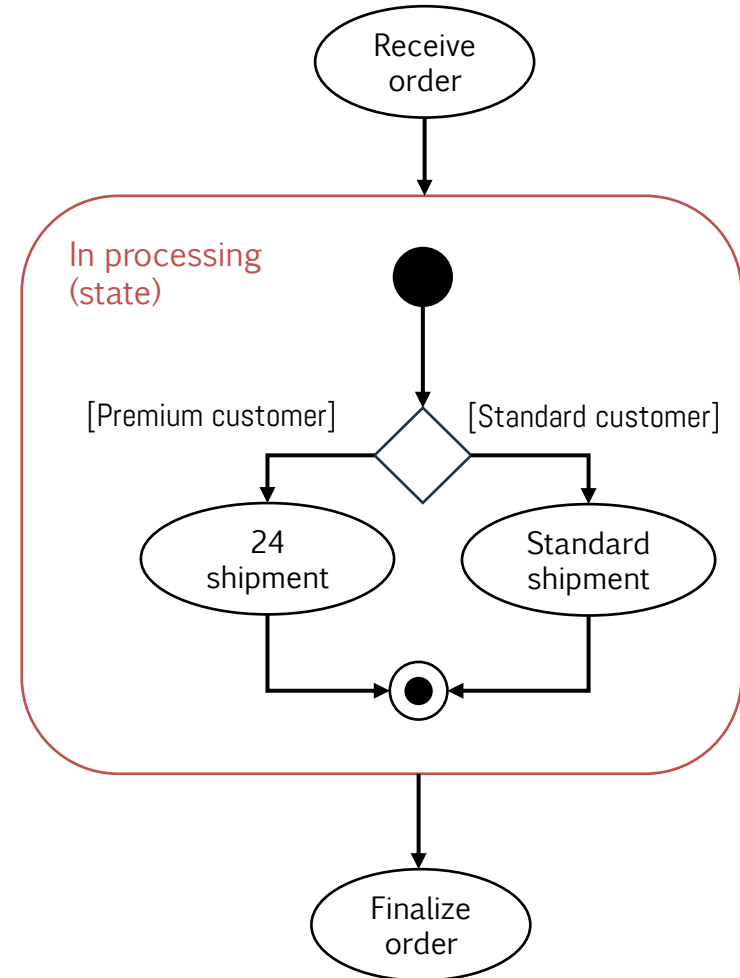
Fork/Join

Model concurrent actions

› Implemented with parallel threads/processes..

› Join point represent

› (We'll go back to this..)
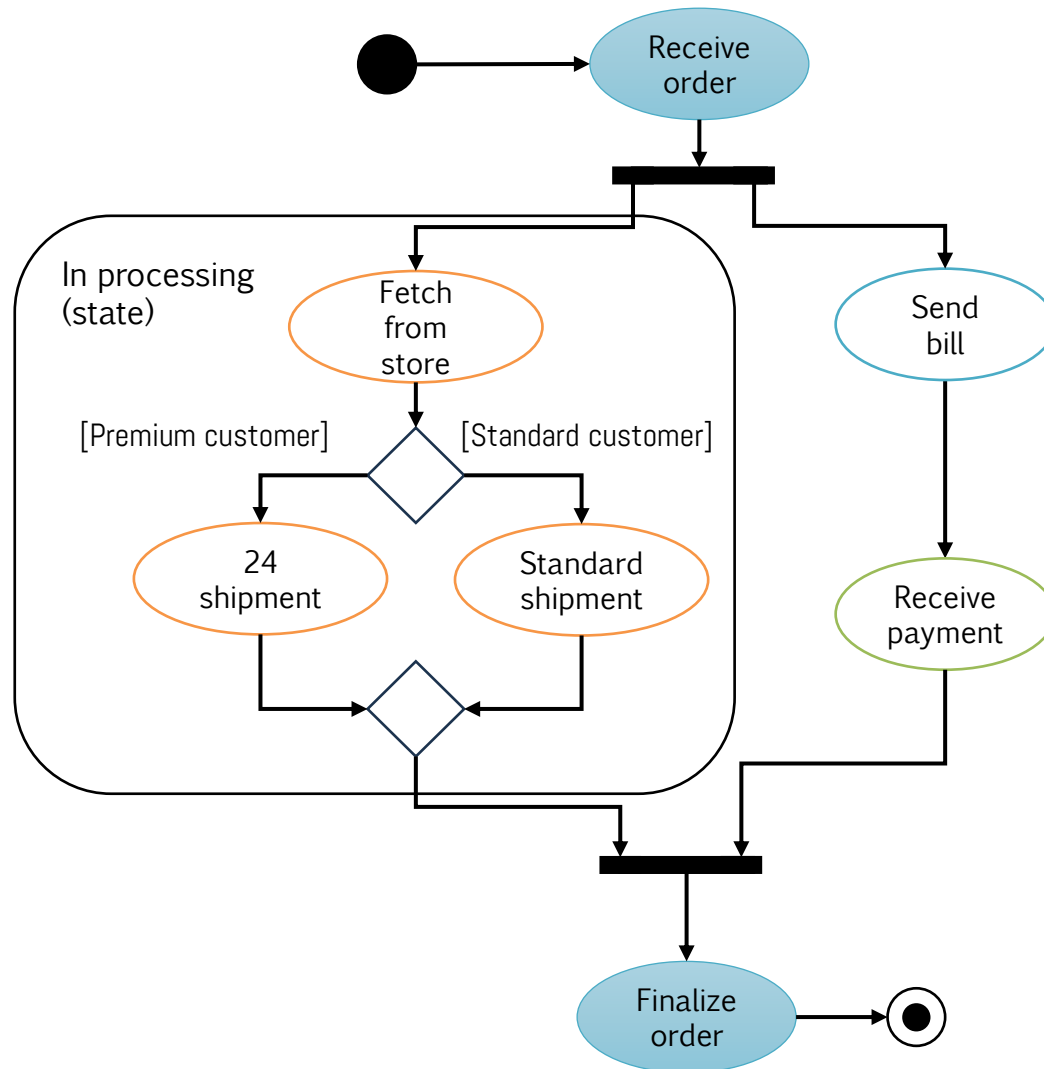
fork

Receive order

join

Finalize order

Sub-activity states

Group activity states /subdiagram into macro functionalities

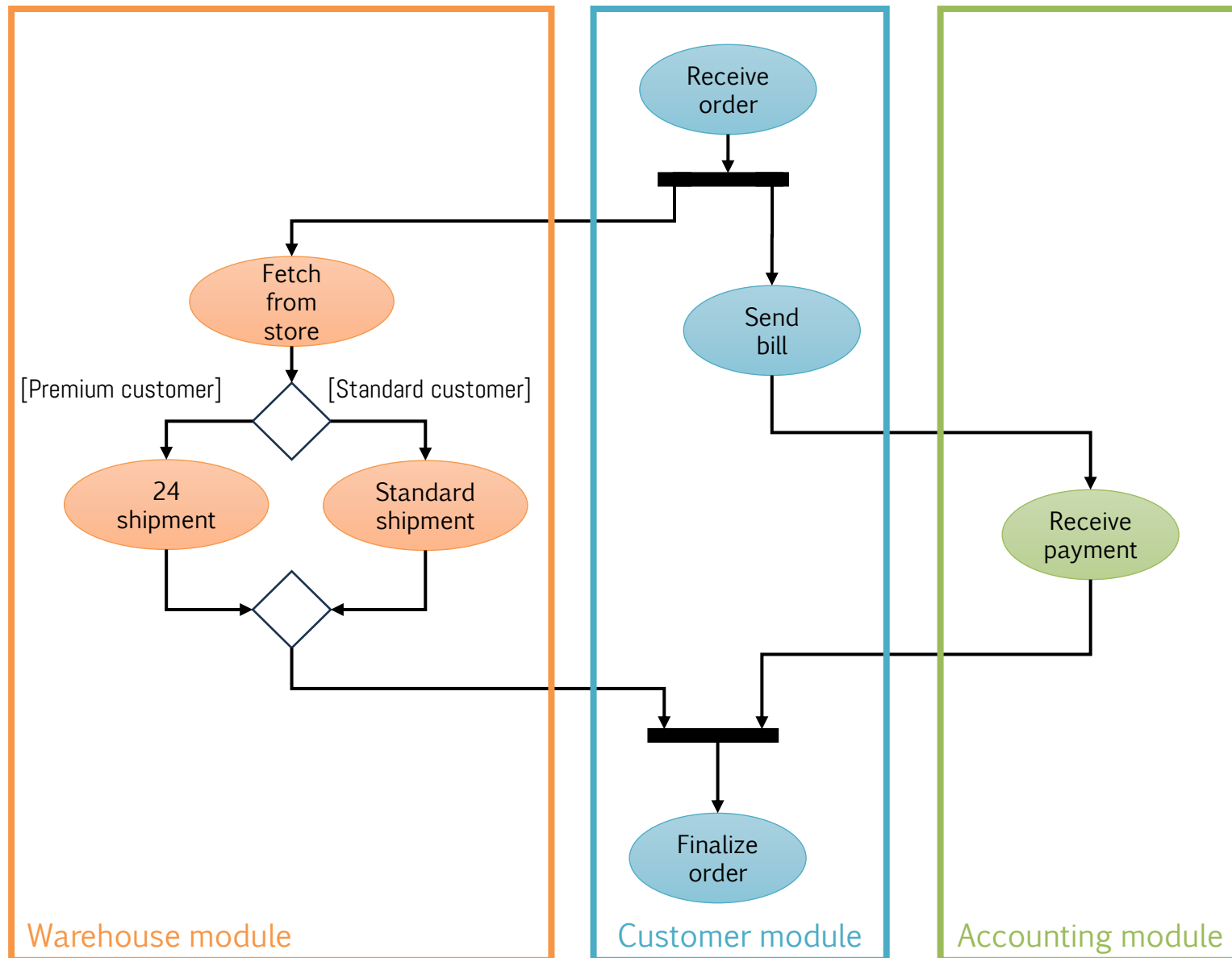› Non atomic

› Can be interrupted

› Non-zero execution time

The full example

Swimlanes

Warehouse module | Customer module | Accounting module

61

Swimlanes

Group activities into <u>areas</u>
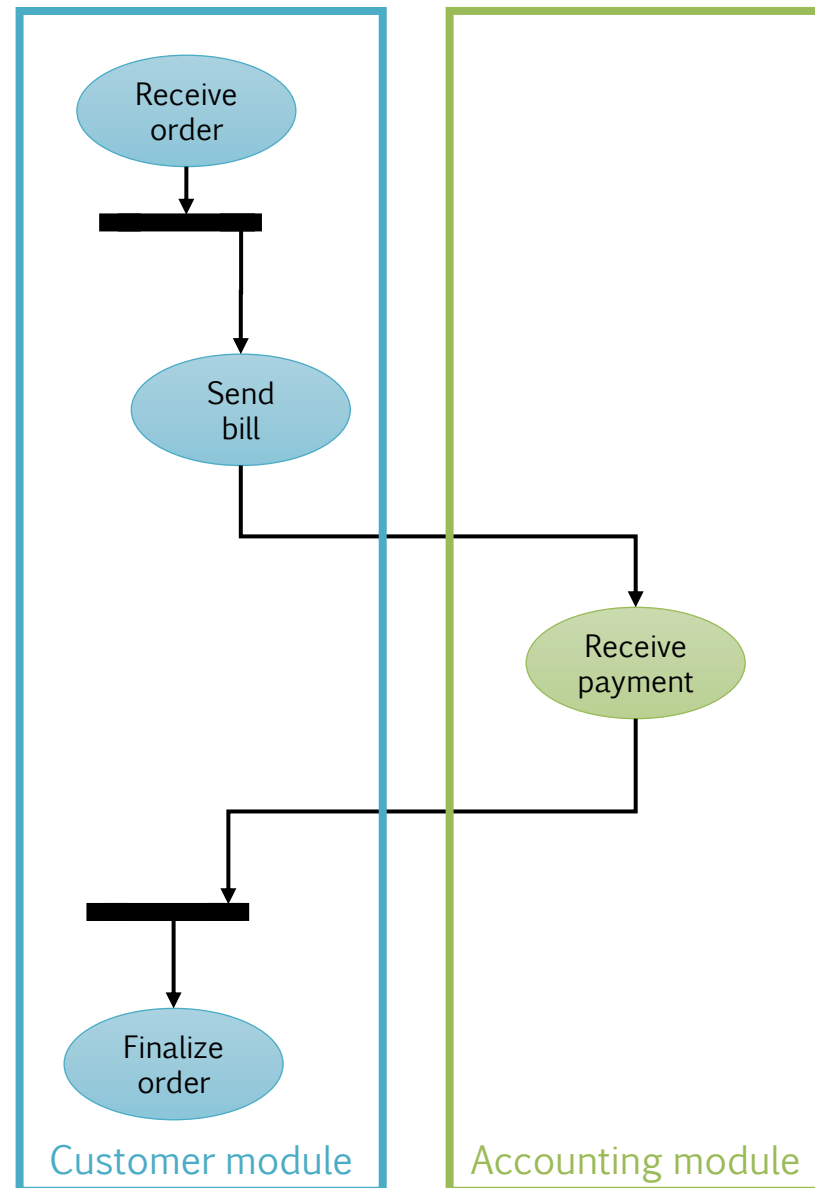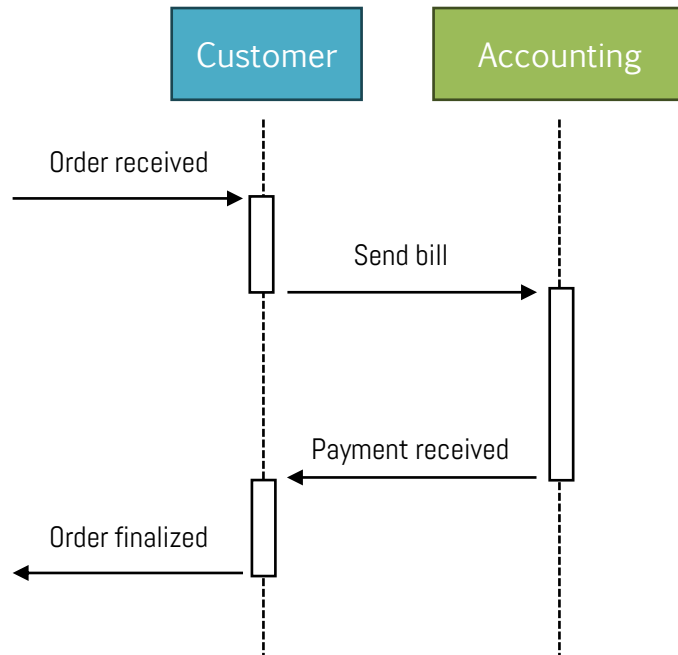
› Partition the activity diagram

What do we model with areas?

› Use cases

› Classes/Objects

› Components

› Business units

› Roles

From activity diagrams to Sequence diagrams

> › Activity diagrams identify scenarios, and we can/must write a Sequence diagrams for each of them!

63

References

Course website

› http://hipert.unimore.it/people/paolob/pub/ProgSW/index.html

Book

› I. Sommerville, ''Introduzione all ingegneria del software moderna'', Pearson
  – Chapter 3

My contacts

› paolo.burgio@unimore.it

› http://hipert.mat.unimore.it/people/paolob/

› https://github.com/pburgio