# *Software Design, Modelling and Analysis in UML*

# *Lecture 10: State Machines Overview*

*2015-12-03*

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

# *Contents & Goals*

**Last Lecture:**

- (Mostly) completed discussion of modelling **structure**.

**This Lecture:**

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What's the purpose of a behavioural model?
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.

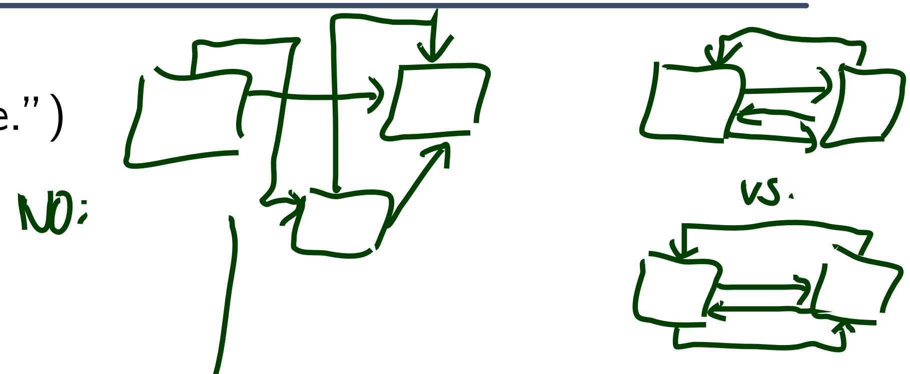
- **Content:**
  - For completeness: Modelling Guidelines for Class Diagrams
  - Purposes of Behavioural Models
  - UML Core State Machines

# *Design Guidelines for (Class) Diagram*

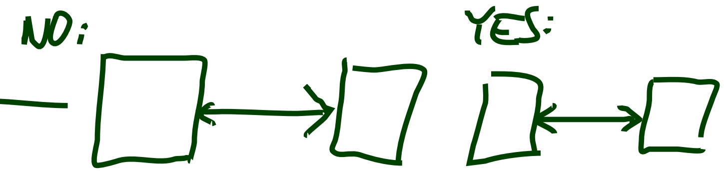
*(partly following Ambler (2005))*

# *General Diagramming Guidelines* *Ambler (2005)*

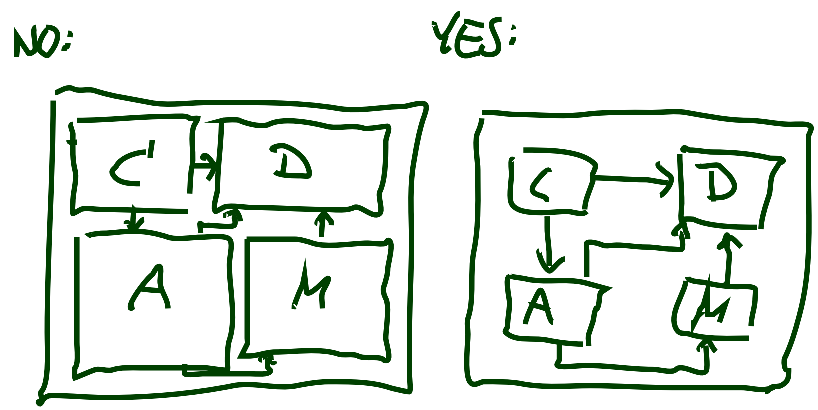
(Note: "Exceptions prove the rule.")



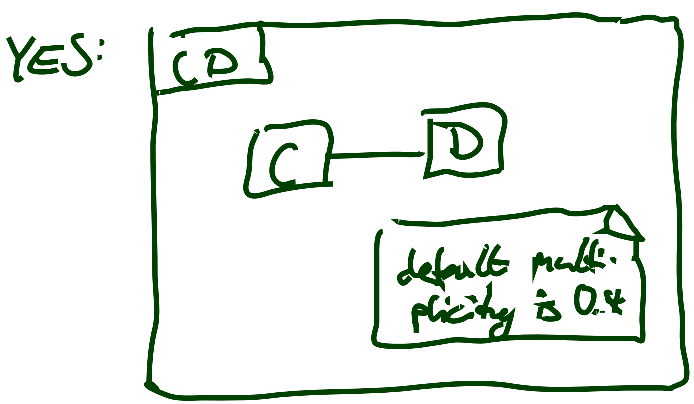

- **2.1 Readability**
  - 1.–3. Support Readability of Lines
  - 4. Apply Consistently Sized Symbols




  - 9. Minimize the Number of Bubbles / Things
  - 10. Include White-Space in Diagrams

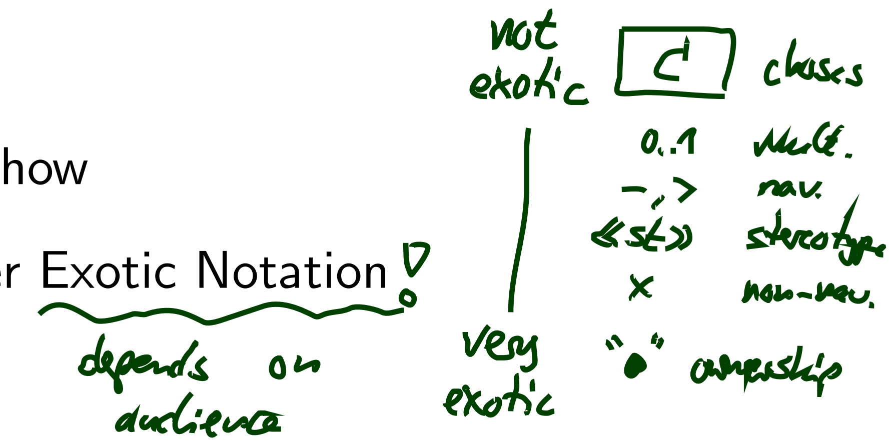


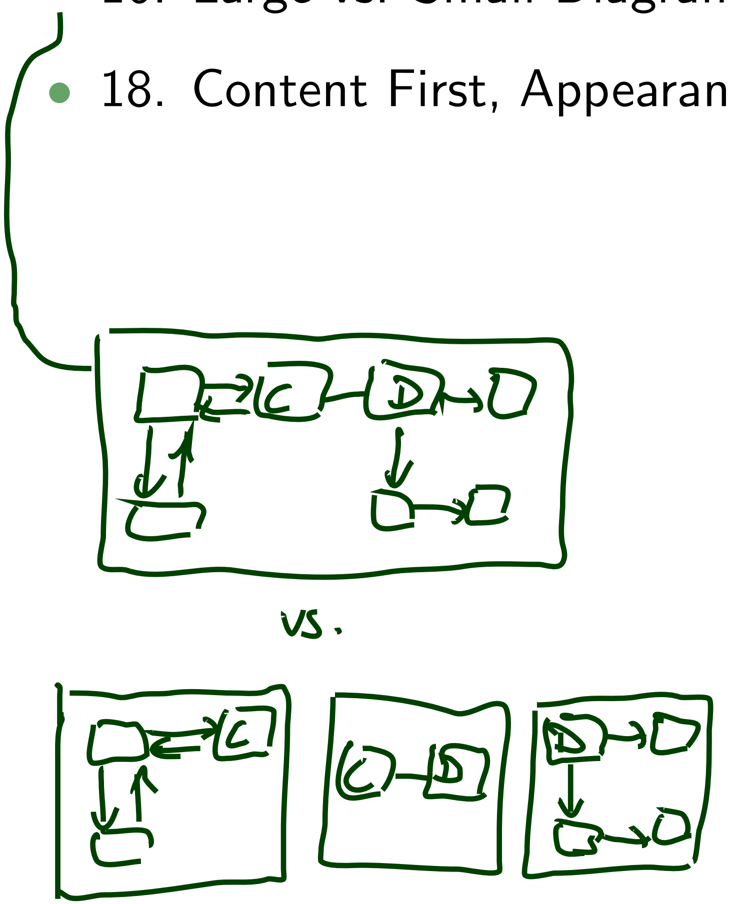

  - 13. Provide a Notational Legend



# *General Diagramming Guidelines Ambler (2005)*

- **2.2 Simplicity**
  - 14. Show Only What You Have to Show
  - 15. Prefer Well-Known Notation over Exotic Notation
  - 16. Large vs. Small Diagrams
  - 18. Content First, Appearance Second



# *General Diagramming Guidelines Ambler (2005)*

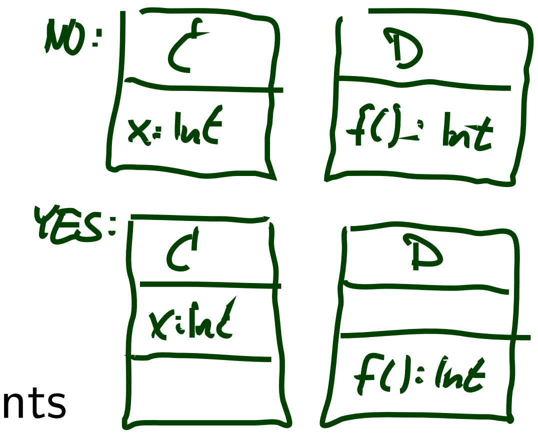
- **2.2 Simplicity**
  - 14. Show Only What You Have to Show
  - 15. Prefer Well-Known Notation over Exotic Notation
  - 16. Large vs. Small Diagrams
  - 18. Content First, Appearance Second
- **2.3 Naming**
  - 20. Set and (23. Consistently) Follow Effective Naming Conventions
- **2.4 General**
  - 24. Indicate Unknowns with Question-Marks
  - 25. Consider Applying Color to Your Diagram
  - 26. Apply Color Sparingly

# *Class Diagram Guidelines Ambler (2005)*

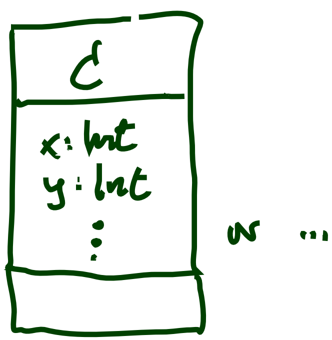
- **5.1 General Guidelines**
  - 88. Indicate Visibility Only on Design Models **(in contrast to analysis models)**
- **5.2 Class Style Guidelines**
  - 96. Prefer Complete Singular Nouns for Class Names
  - 97. Name Operations with Strong Verbs
  - 99. Do Not Model Scaffolding Code **[Except for Exceptions]**

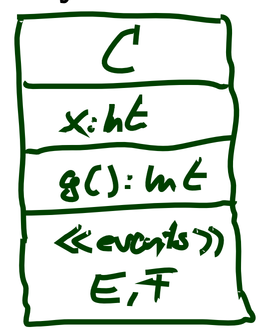
    e.g. get/set methods

# Class Diagram Guidelines Ambler (2005)

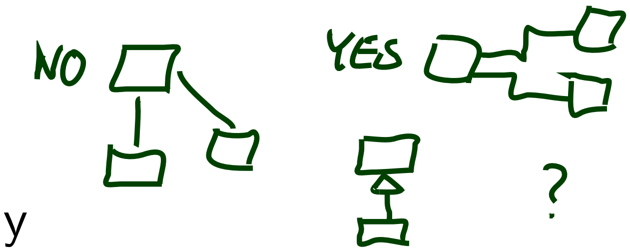



- **5.2 Class Style Guidelines**
  - 103. Never Show Classes with Just Two Compartments
  - 104. Label Uncommon Class Compartments
  - 105. Include an Ellipsis (...) at the End of an Incomplete List
  - 107. List Operations/Attributes in Order of Decreasing Visibility

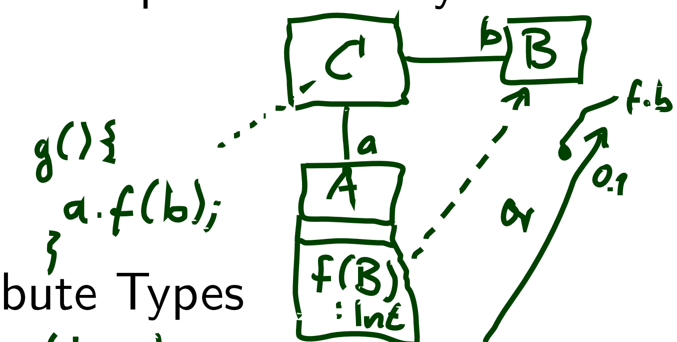


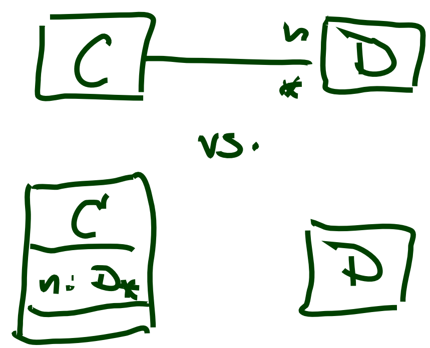



- **5.3 Relationships**

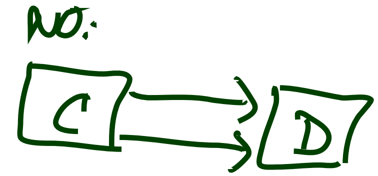



  - 112. Model Relationships Horizontally
  - 115. Model a Dependency When the Relationship is Transitory
  - 117. Always Indicate the Multiplicity
    (or have good defaults)
  - 118. Avoid Multiplicity "$*$"
  - 119. Replace Relationship Lines with Attribute Types
    (to have fewer lines)

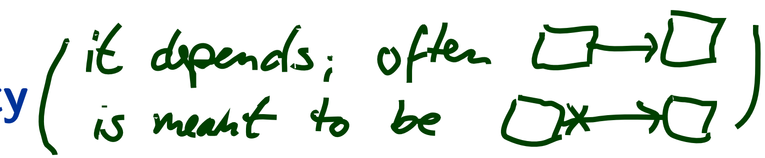


# Class Diagram Guidelines Ambler (2005)



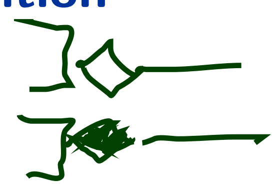

- **5.4 Associations**
  - 127. Indicate Role Names When Multiple Associations Between Two Classes Exist
  - 129. Make Associations Bidirectional Only When Collaboration Occurs in Both Directions
  - **131. Avoid Indicating Non-Navigability**




  - 133. Question Multiplicities Involving Minimums and Maximums

e.g. 3..10

- **5.6 Aggregation and Composition**



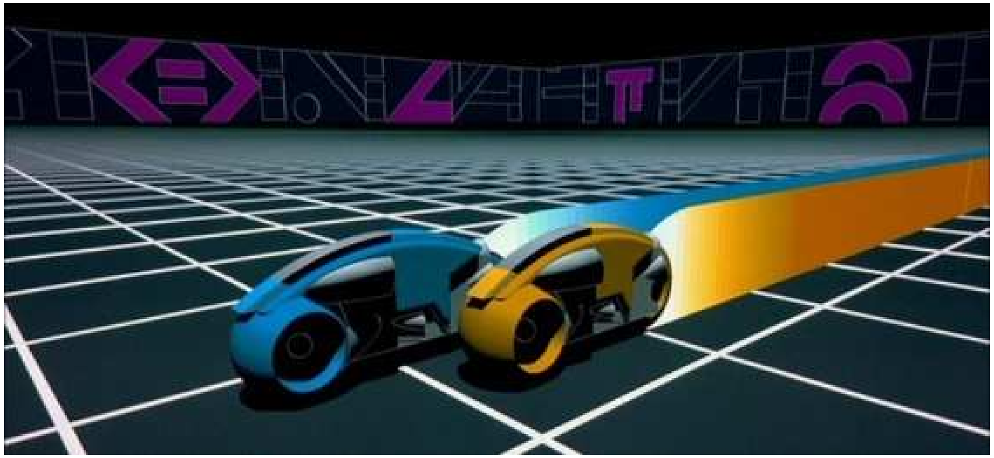
  - → exercises

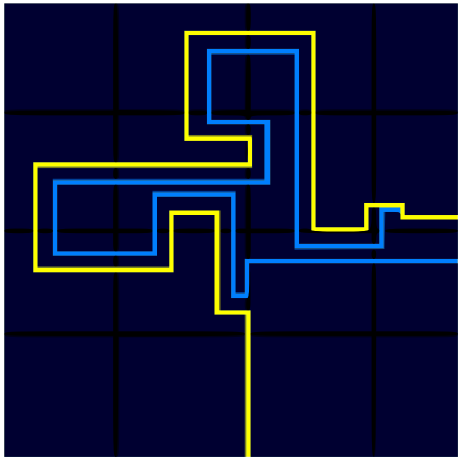
*Example: Modelling Games*

# *Task: Game Development*

**Task**: **develop** a **video game**. **Genre**: **Racing**. **Rest**: **open**, i.e.

| Degrees of freedom: | Exemplary choice: 2D-Tron |
|---|---|
| • simulation vs. arcade | arcade |
| • platform (SDK or not, open or proprietary, hardware capabilities...) | open |
| • graphics (3D, 2D, ...) | 2D |
| • number of players, AI | min. 2, AI open |
| • controller | open (later determined by platform) |
| • game experience | minimal: main menu and game |

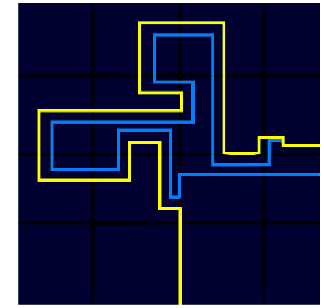




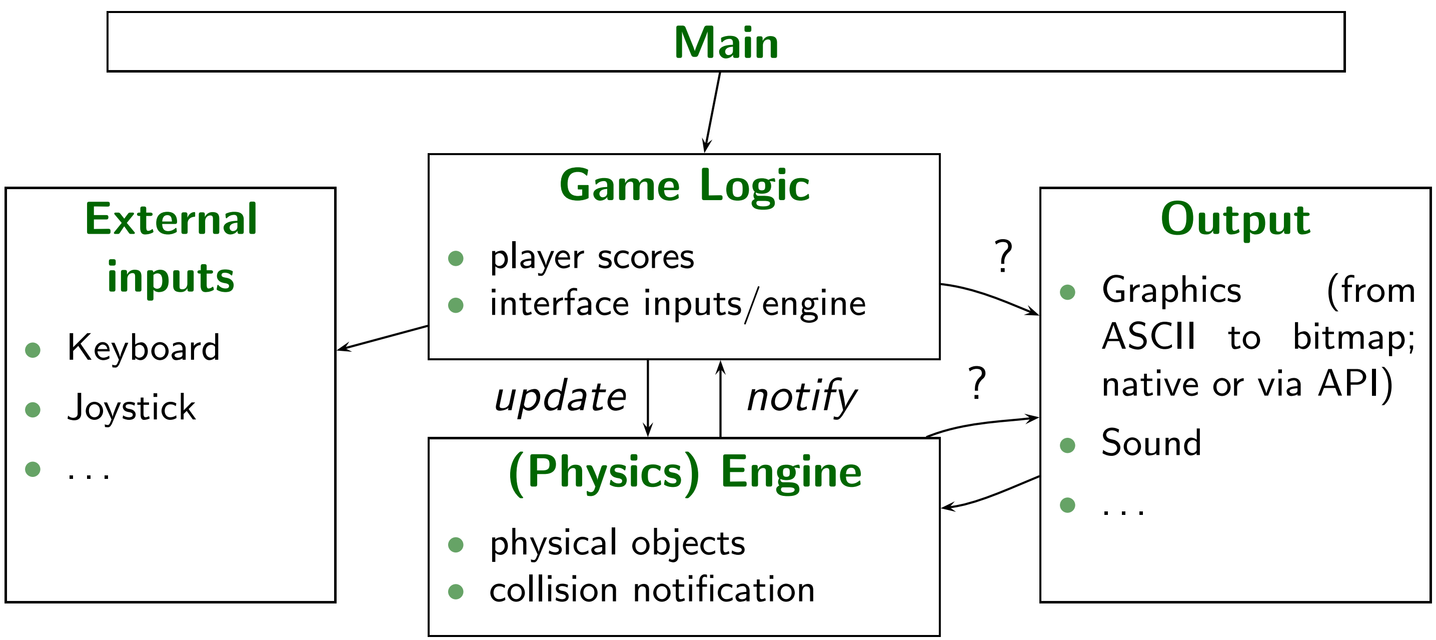
# *Modelling Structure: 2D-Tron*

- In many domains, there are canonical architectures – and adept readers try to see/find/match this!
- For games:

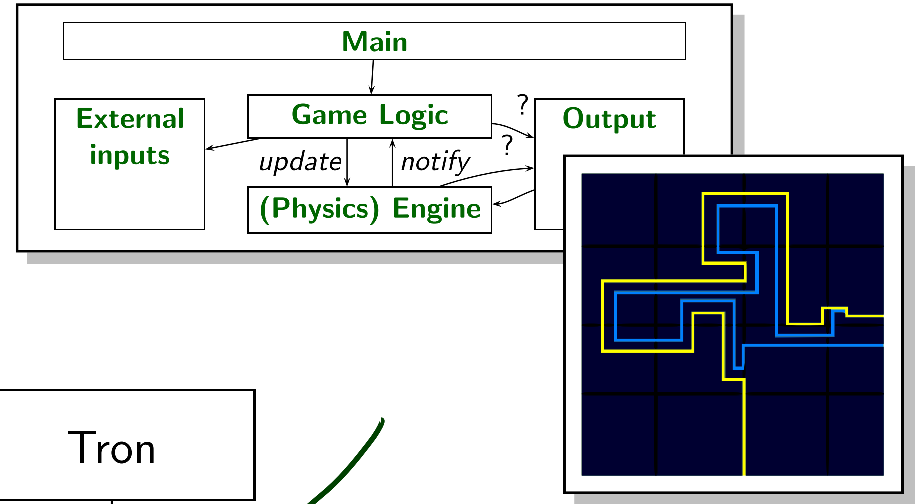
**2D-Tron**

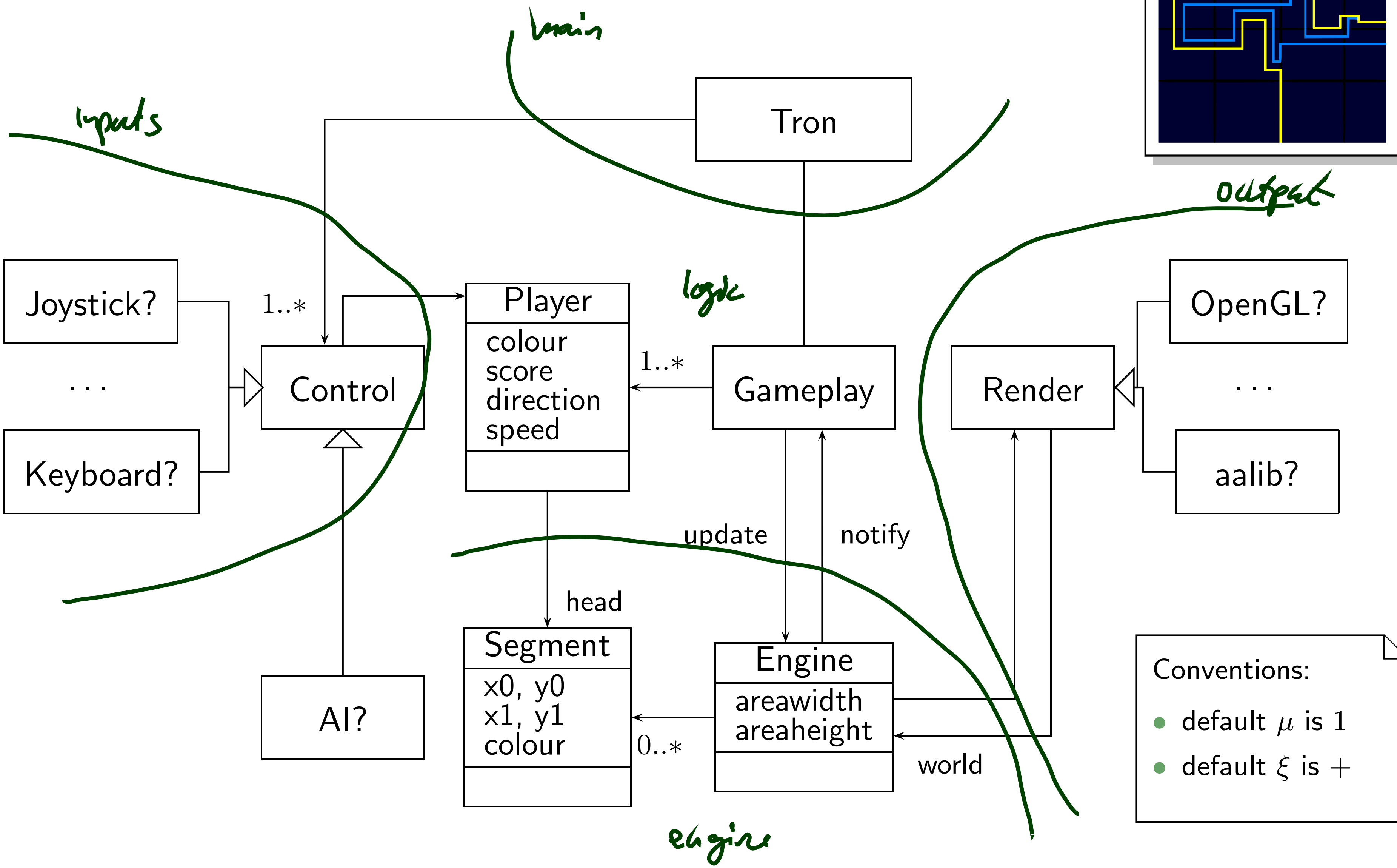


- arcade
- platform open
- 2D
- min. 2, AI open
- controller open
- only game, no menues



# *Modelling Structure: 2D-Tron*



# *Modelling Behaviour*

# *Stocktaking...*

**Have:** Means to model the **structure** of the system.

- Class diagrams graphically, concisely describe sets of system states.
- OCL expressions logically state constraints/invariants on system states.

**Want:** Means to model **behaviour** of the system.

- Means to describe how system states **evolve over time**, that is, to describe sets of **sequences**

$$\sigma_0, \sigma_1, \dots \in \Sigma^\omega$$

  of **system states**.

# *What Can Be Purposes of Behavioural Models?*

**Example**: Pre-Image **Image**
(the UML model is supposed to be the blue-print for a software system).

A description of behaviour could serve the following purposes:

- **Require** Behaviour. **"System definitely does this"**
  *"This sequence of inserting money and requesting and getting water must be possible."*
  (Otherwise the software for the vending machine is completely broken.)

- **Allow** Behaviour. **"System does subset of this"**
  *"After (inserting money and choosing a drink), the drink is dispensed (if in stock)."*
  (If the implementation insists on taking the money first, that's a fair choice.)

- **Forbid** Behaviour. **"System never does this"**
  *"This sequence of getting both, a water and all money back, must not be possible."*
  (Otherwise the software is broken.)

**Note**: the latter two are trivially satisfied by doing nothing...

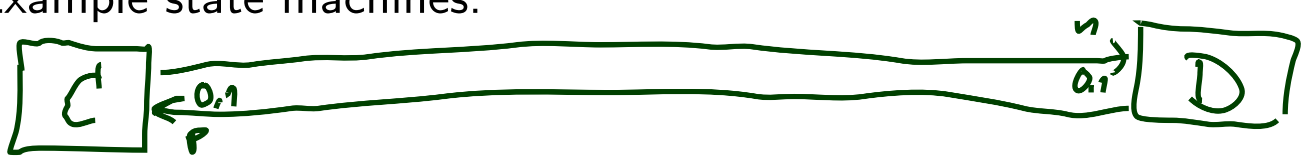
# *Constructive Behaviour in UML*

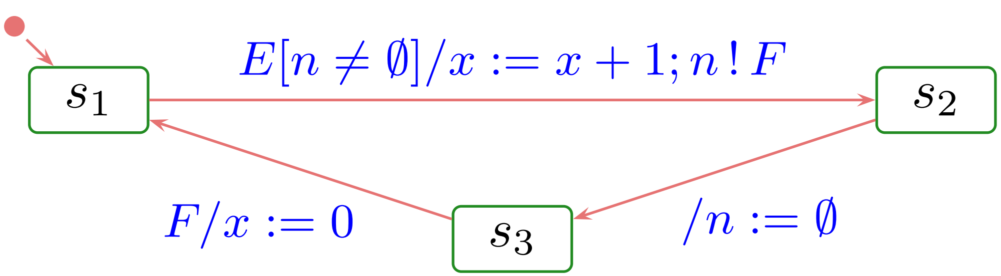
UML provides two visual formalisms for constructive description of behaviours:

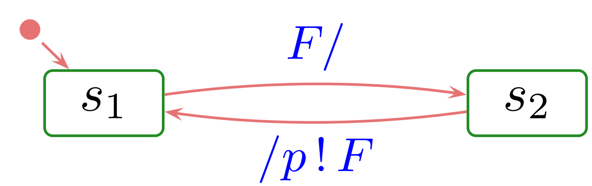
- **Activity Diagrams**
- **State-Machine Diagrams**

We (exemplary) focus on State-Machines because

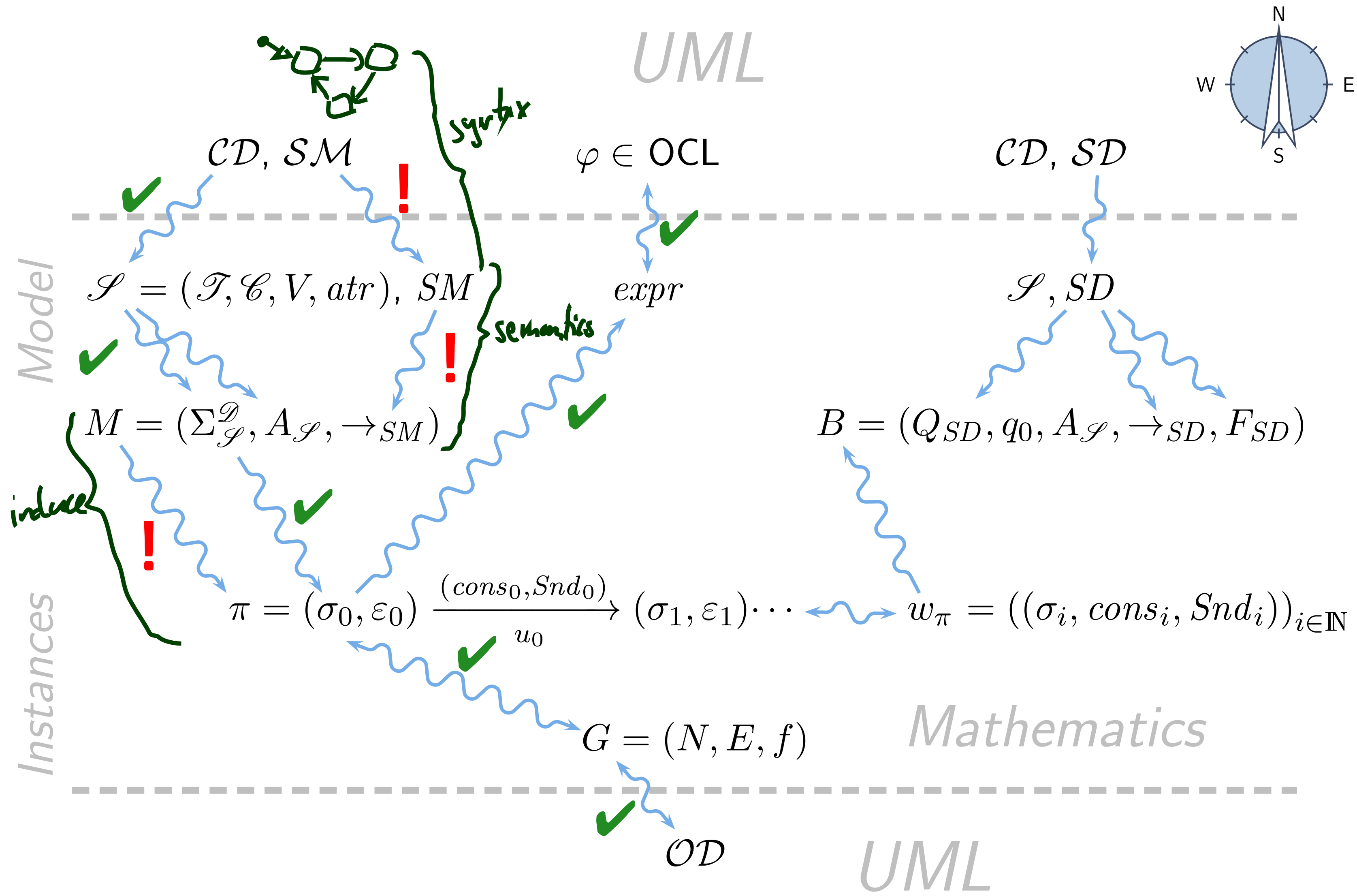
- somehow "practice proven" (in different flavours),
- prevalent in embedded systems community,
- indicated useful by Dobing and Parsons (2006) survey, and
- Activity Diagram's intuition changed (between UML 1.x and 2.x) from transition-system-like to petri-net-like...
- Example state machines:



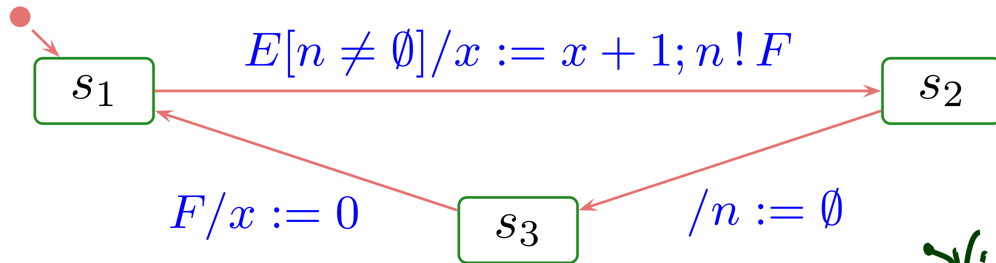
# Course Map

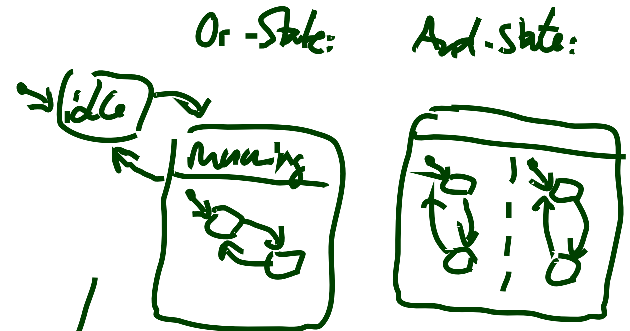


UML
syntax
$\mathcal{CD}$, $\mathcal{SM}$
$\varphi \in$ OCL
$\mathcal{CD}$, $\mathcal{SD}$
N
W
E
S
Model
semantics
$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$, $SM$
$expr$
$\mathscr{S}$, $SD$
$M = (\Sigma_{\mathscr{S}}^{\mathscr{D}}, A_{\mathscr{S}}, \to_{SM})$
$B = (Q_{SD}, q_0, A_{\mathscr{S}}, \to_{SD}, F_{SD})$
induce
$\pi = (\sigma_0, \varepsilon_0) \xrightarrow[u_0]{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \cdots$
$w_\pi = ((\sigma_i, cons_i, Snd_i))_{i \in \mathbb{N}}$
Instances
$G = (N, E, f)$
Mathematics
$\mathcal{OD}$
UML

# *UML State Machines: Overview*

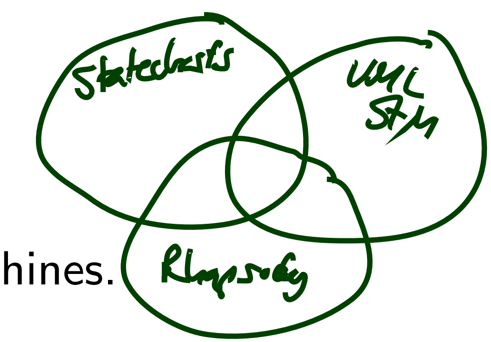
# UML State Machines




**Brief History**:

- Rooted in **Moore/Mealy machines**, Transition Systems, etc.
- Harel (1987): **Statecharts** as a concise notation, introduces in particular hierarchical states.
- Manifest in tool **Statemate** Harel et al. (1990) (simulation, code-generation); nowadays also in **Matlab/Simulink**, etc.
- From UML 1.x on: **State Machines** (not the official name, but understood: **UML-Statecharts**)
- Late 1990's: tool **Rhapsody** with code-generation for state machines.




**Note**: there is a common core, but each dialect interprets some constructs subtly different Crane and Dingel (2007). *(Would be too easy otherwise. . . )*

# Roadmap: Chronologically

**Syntax**:

(i) UML State Machine Diagrams.

(ii) Def.: Signature with **signals**.

(iii) Def.: **Core state machine**.

(iv) Map UML State Machine Diagrams to core state machines.

**Semantics**:
The Basic Causality Model

(v) Def.: **Ether** (aka. event pool)

(vi) Def.: **System configuration**.

(vii) Def.: **Event**.

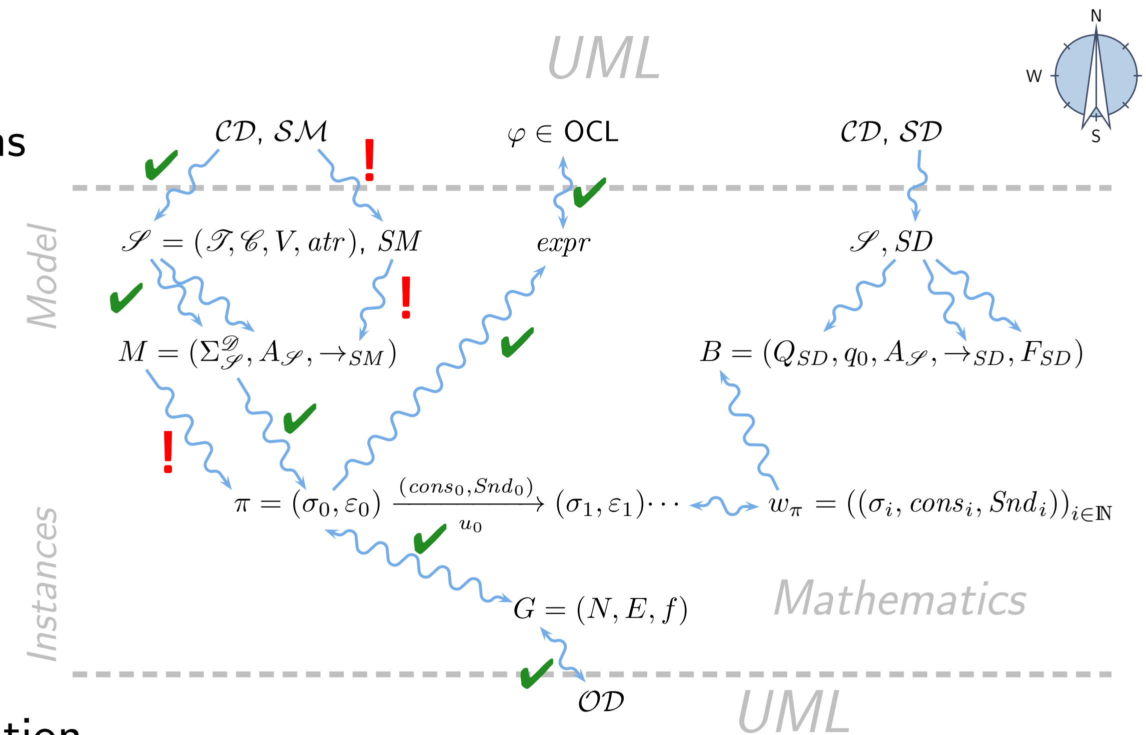
(viii) Def.: **Transformer**.

(ix) Def.: **Transition system**, computation.

(x) Transition relation induced by core state machine.

(xi) Def.: **step**, **run-to-completion step**.

(xii) Later: Hierarchical state machines.



# *UML State Machines: Syntax*

# *Signature With Signals*

**Definition.** A tuple

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}), \qquad \mathscr{E} \text{ a set of signals},$$

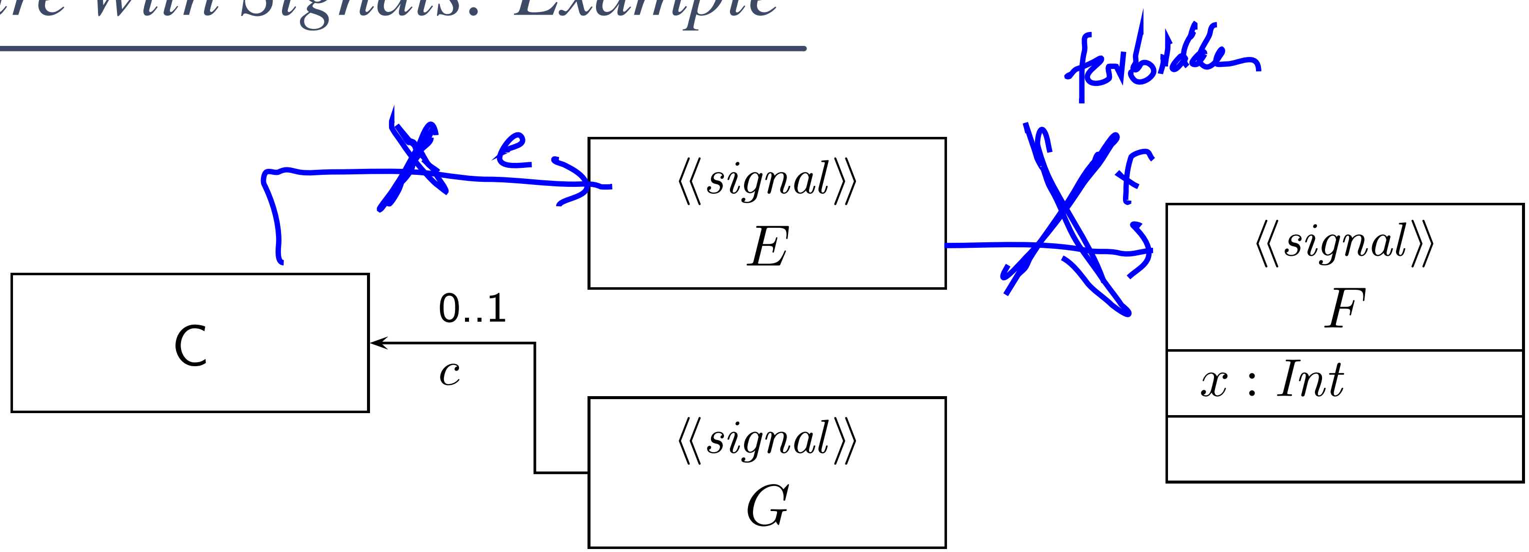
is called signature (with signals) if and only if

$$(\mathscr{T}, \mathscr{C} \cup \mathscr{E}, V, atr)$$

is a signature (as before).

**Note:** Thus conceptually, **a signal is a class** and can have attributes of plain type, and participate in associations.

# *Signature with Signals: Example*




$$\mathcal{S} = (\{Int\}, \{C\}, \{x : Int, c : C_{0,1}\}, \{C \mapsto \emptyset, E \mapsto \emptyset, G \mapsto \{c\}, F \mapsto \{x\}\}, \{E, F, G\})$$

**Definition.**
A core state machine over signature $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a tuple

$$M = (S, s_0, \rightarrow)$$

where

- $S$ is a non-empty, finite set of **(basic) states**,
- $s_0 \in S$ is an **initial state**,
- and

source state

destination state

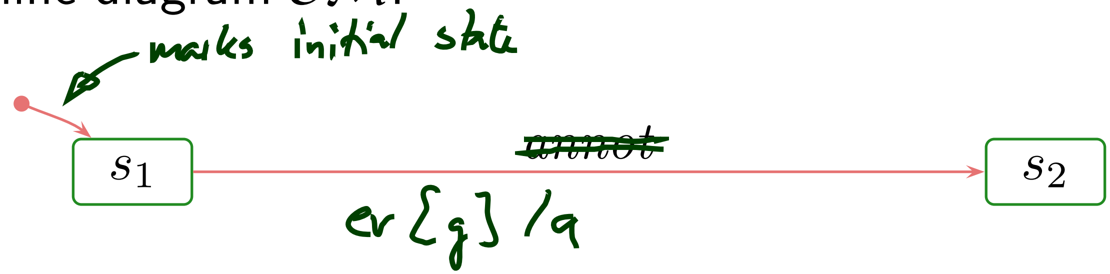
$$\rightarrow \subseteq S \times \underbrace{(\mathscr{E} \,\dot{\cup}\, \{\_\})}_{\text{trigger}} \times \underbrace{Expr_{\mathscr{S}}}_{\text{guard}} \times \underbrace{Act_{\mathscr{S}}}_{\text{action}} \times S$$

  is a labelled transition relation.

We assume a set $Expr_{\mathscr{S}}$ of boolean expressions over $\mathscr{S}$ (for instance OCL, may be something else) and a set $Act_{\mathscr{S}}$ of **actions**.

# *From UML to Core State Machines: By Example*

UML state machine diagram $\mathcal{SM}$:




$$annot ::= \big[\ \langle event\rangle [\,.\,\langle event\rangle]^*\ \big]\ \big[\,[\ \langle guard\rangle\ ]\,\big]\ \big[\ /\ \ [\langle action\rangle]\ \big]$$


with

- $event \in \mathscr{E}$,
- $guard \in Expr_{\mathscr{S}}$ (**default**: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (**default**: `skip`, assumed to be in $Act_{\mathscr{S}}$)

**maps to**

$$M(\mathcal{SM}) = \big(\{s_1, s_2\}, \{s_1\}, \{(s_1, ev, g, a, s_2)\}\big)$$

# Abbreviations and Defaults in the Standard

**Reconsider** the syntax of transition annotations:

$$annot ::= \big[\ \langle event\rangle [\,.\,\langle event\rangle]^{*}\ \big]\ \big[\,\mathbf{[}\,\langle guard\rangle\,\mathbf{]}\,\big]\ \big[\,/\ \ [\langle action\rangle]\ \big]$$

where $event \in \mathscr{E}$, $guard \in Expr_{\mathscr{S}}$, $action \in Act_{\mathscr{S}}$.

**What if things are missing?**

| | | |
|---|---|---|
| | $\rightsquigarrow$ | _ [true] / skip — (.., _, true, skip, ..) |
| $/$ | $\rightsquigarrow$ | _ [true] / skip — \~"— |
| $E\ /$ | $\rightsquigarrow$ | E [true] / skip — (.., E, true, skip, ..) |
| $/\ act$ | $\rightsquigarrow$ | _ [true] / act — (.., _, true, act, ..) |
| $E\ /\ act$ | $\rightsquigarrow$ | E [true] / act — (.. E, true, act, ..) |

**In the standard**, the syntax is even more elaborate:

- $E(v)$ — when consuming $E$ in object $u$,
  attribute $v$ of $u$ is assigned the corresponding attribute of $E$.
- $E(v : T)$ — similar, but $v$ is a local variable, scope is the transition

# *State-Machines belong to Classes*

In the following, we assume that

- a UML model consists of a set $\mathscr{CD}$ of class diagrams and a set $\mathscr{SM}$ of **state chart diagrams** (each comprising one **state machine** $\mathcal{SM}$).
- each state machine $\mathcal{SM} \in \mathscr{SM}$ is **associated with a class** $C_{\mathcal{SM}} \in \mathscr{C}(\mathscr{S})$.
- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathscr{C}(\mathscr{S})$ has a state machine $\mathcal{SM}_C$ and that its class $C_{\mathcal{SM}_C}$ is $C$.

  If not explicitly given, then this one:

  $$\mathcal{SM}_0 := (\{s_0\}, s_0, (s_0, \_, true, \mathtt{skip}, s_0)).$$

  We will see later that this choice does no harm semantically.

**Intuition 1**: $\mathcal{SM}_C$ describes the behaviour of **the instances** of class $C$.

**Intuition 2**: Each instance of class $C$ executes $\mathcal{SM}_C$.

**Note**: we don't consider **multiple state machines** per class. We will see later that this case can be viewed as a single state machine with as many AND-states.

# *References*

# *References*


Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.

Crane, M. L. and Dingel, J. (2007). UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and Systems Modeling*, 6(4):415–435.

Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.