# SOFTWARE MODELING

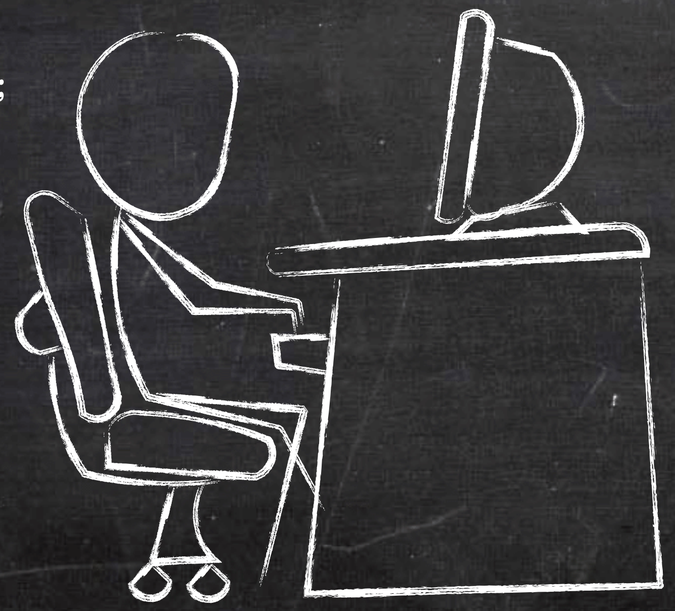
# Motivation

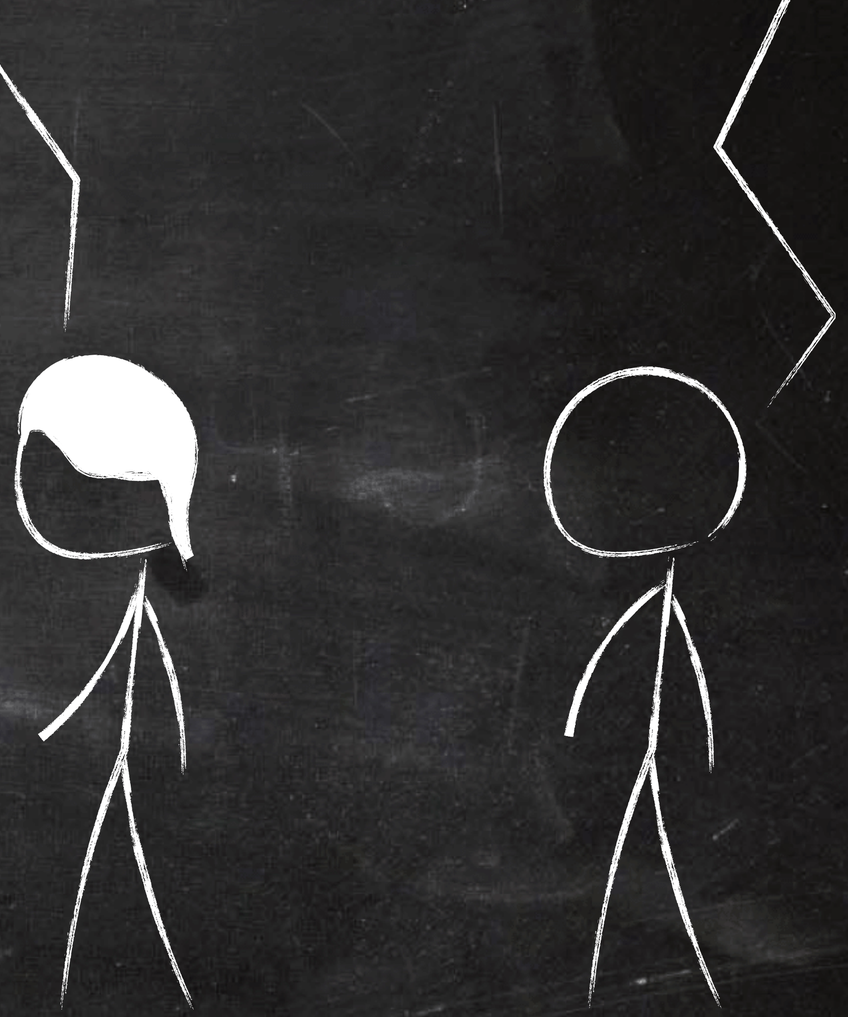
HOW TO ACCESS ALL ITEMS IN A LIST?

```
@Override public Iterator<T> iterator() {
        return new Iterator<T>() {
            private int idx = 0;
            @Override public boolean hasNext() { return idx < size && list[idx] != null; }
            @Override public Type next() { return list[currentIndex++]; }
        }; }
```

NO, NO. IT'S MORE LIKE THIS…

```
template <class Derived, class Value, class CategoryOrTraversal, class Reference =
Value&, class Difference = ptrdiff_t>
class iterator_facade {
 public:
   typedef remove_const<Value>::type value_type;
   typedef Reference reference;
   typedef Value* pointer;
   typedef Difference difference_type;
   reference operator*() const;
  Derived& operator++();
   Derived operator++(int);
   Derived& operator—();
…
```





NEED FOR SIMPLE IMPLEMENTATION— INDEPENDENT WAY TO PRESENT STRUCTURE, BEHAVIOR, INTERACTIONS,…

sloppy based on xkcd.com

# MODELING LANGUAGES

Modeling languages are artificial languages with certain rules to express…

- … information & knowledge
- … structure, dependencies & workflow
- … *and more*

at an abstract (i.e., implementation-independent) level.

Used to provide multiple stakeholders such as

- … software guys (engineers, architects, analyzers & testers), and
- … non-software guys (users, customers, management,…)

insights into a (complex) system or a process without giving overwhelming (technical) details.

# MODELING LANGUAGES (2)

## Pros & Cons

Modeling languages help to…

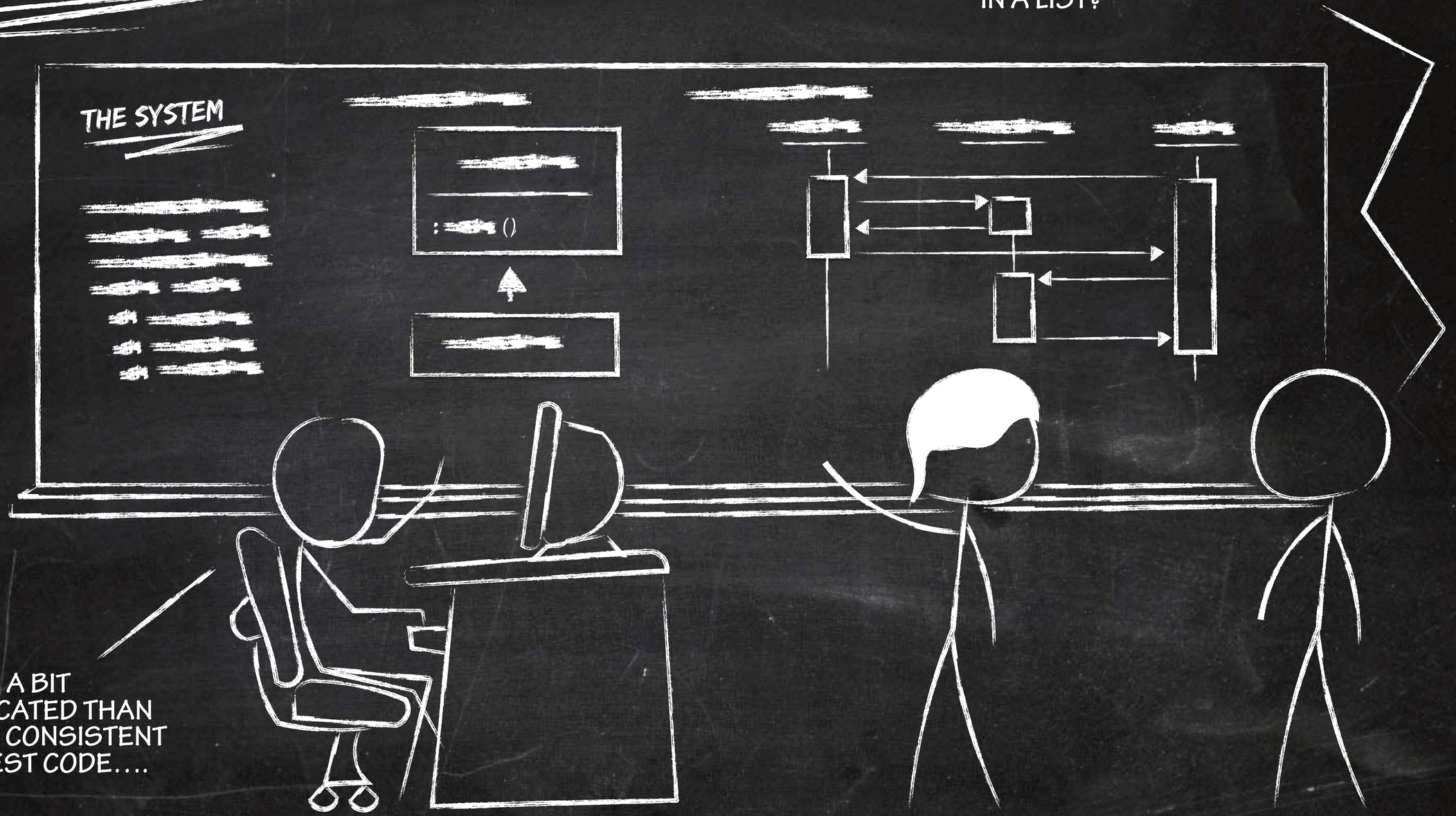
- PRO … avoid need for „reverse engineering & analyzing of code“
- PRO … getting you started in a complex system (i.e., better orientation)
- PRO … find dependencies & understand program flow at a glance
- PRO … communicate with other involved people on a common level

The downside is…

- CON maintenance costs (time/money): keeping model & code consistent is effort
- CON waste effort if wrong audience: hardliner engineers may not need models
- CON every (required) detail (even abstractly) lead to complex diagrams
- CON over-engineering: investing in complex design for simple problems

… also in practice, it’s rarely the case that models satisfy 100% of the UML specification
… also in data-intensive system remember design & abstraction introduces (often) overhead that comes with performance penalties

# Motivation (2)



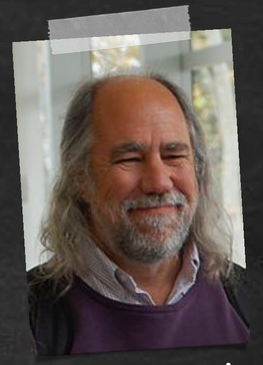

sloppy based on xkcd.com

# MODELING LANGUAGES (3)

Important modeling languages:

UML | ER-MODEL | FLOW CHARTS | ...

# Unified Modeling Language (UML)


Grady Booch


James Rumbaugh


Ivar Jacobson

UML is a kind of visual language rooted back to mid-1990's

- standardized 1997 (UML 1.0)
- de-facto industry standard
- Current version UML 2.5

Set of graphic notations for visual models on different aspects of modularized/object-oriented systems:

- Structural diagrams
  class, object, component, composite structure, package, deployment,...
- Behavior diagrams
  use-case, activity, state machine,...
- Interactions diagrams
  sequence, communication, timing, interaction overview,...

see all details: http://www.uml.org

# UML Structural Diagrams (1)

## Class Diagrams (1)

### Purpose

Class diagrams:

- show (subset) of classes (or interfaces) incl. their members, and their relationships

classes: shared structure and shared behavior of certain objects (aka instances)

- type name (aka class name)
- type related functions (aka methods)
- contained variables (aka members)
- …

relationships:

- parent-child (aka inheritance)
- dependencies
- extension of methods
- …

Purposes

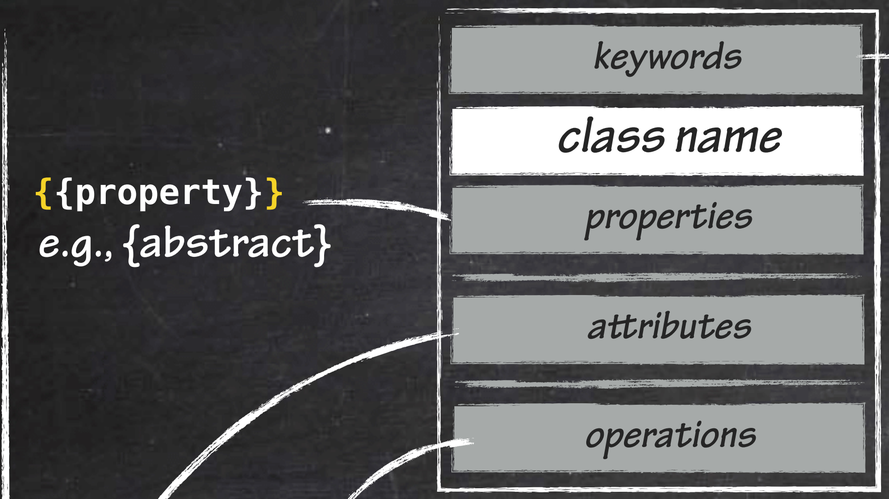
- Design of the system in a static view
- Showing responsibilities in the system
- Required for component and deployment diagrams

# UML Structural Diagrams (1)

like blueprints for objects

## Class Diagrams (2)

### Notation and building blocks




<<{property,…}>>
e.g., <<interface>>

{{property}}
e.g., {abstract}

[visibility] name [({parameters})] [: result type]

[direction] name :type [multiplier] [= default value]

in, out, or inout

[visibility] [/] name [:type] [multiplier] [= initial value]

scoping: underlining represents „same for all instances“ (aka static)

| | |
|---|---|
| + | public |
| - | private |
| # | protected |
| / | derived |
| ~ | package |

**Generalization**


B has implicitly all properties of A

**Association**


A and B have relationship R

**Part-whole relationships**

Aggregation


A is part of B

Composition


A is part of B but A *cannot exists* without B

Visibility

Class

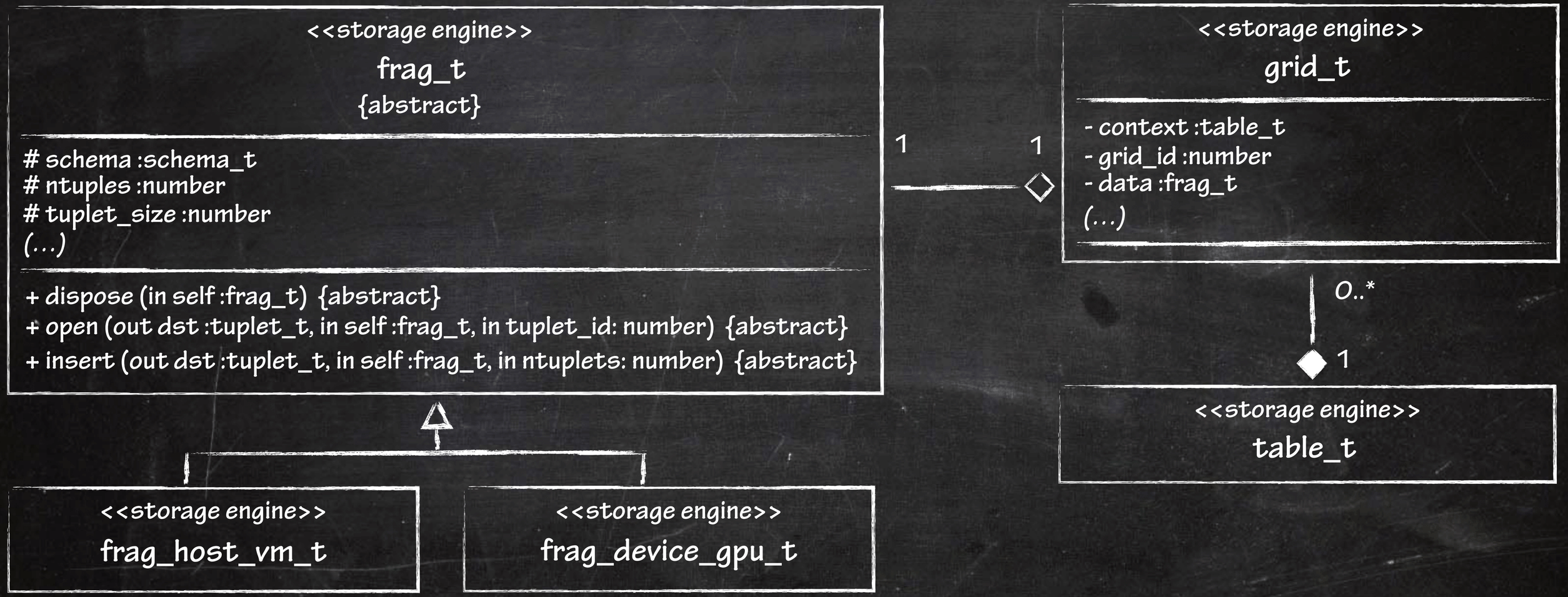
Relationships

Mandatory  Optional  [ ] Optional

# UML Structural Diagrams (1)

## Class Diagrams (3)

Example



# UML Structural Diagrams (2)

## Object Diagrams (1)

### Purpose

Object diagrams:

- show a detailed snapshot of certain instances (incl. values of certain attributes) at a point in time.

it's like pausing your program and take a look into the variable value assignments at a particular time during runtime

Class diagram and object diagram are closely related

- class diagrams focus on classes
- object diagrams focus on instances of these classes (aka objects)

Purposes

- Shows objects in the system during runtime
- Static view on interaction of objects
- Object behavior and relationships at particular moment
- Prototyping a system (cf., fake/simulate data and environment)

cf. debugging

# UML Structural Diagrams (2)

concrete things of classes

## Object Diagrams (2)

Notation and building blocks

*same as for class diagrams with the exception that*

*- it's about objects not classes*

*- lines („links“) connect objects*

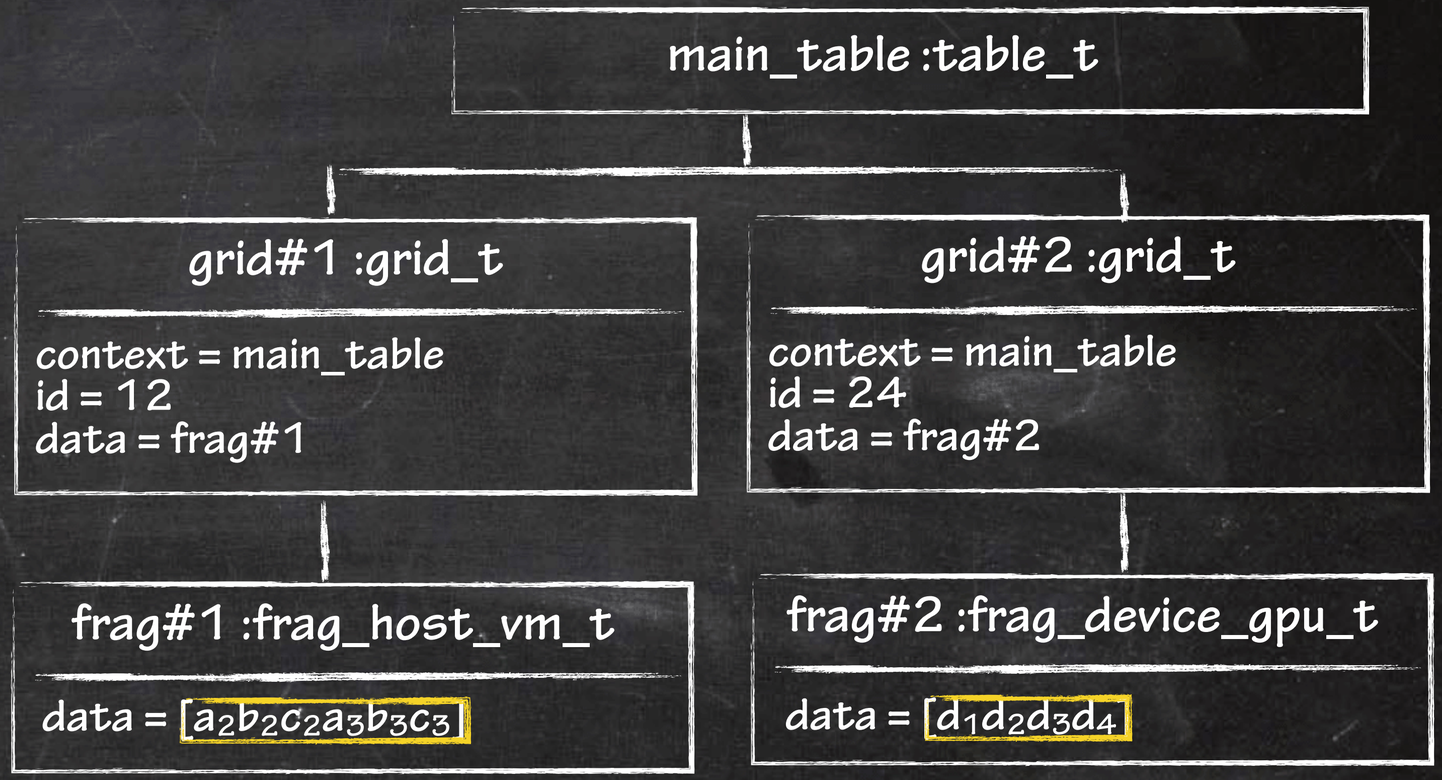
# UML Structural Diagrams (2)

## Object Diagrams (3)

Example

| A | B | C | D |
|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| $a_2$ | $b_2$ | $c_2$ | $d_2$ |
| $a_3$ | $b_3$ | $c_3$ | $d_3$ |
| $a_4$ | $b_4$ | $c_4$ | $d_4$ |

(data view)




(object diagram)

# UML Structural Diagrams (3)

## Component Diagrams (1)

### Purpose

Component diagrams:

- show organization, hierarchy and relationships of (logical) system components*.

larger design units that will be implemented using replaceable things

### Purposes

- Visualize the systems logical components (w.r.t. its architecture)
- Communicate early overall view on the system (e.g., to key project stakeholders)

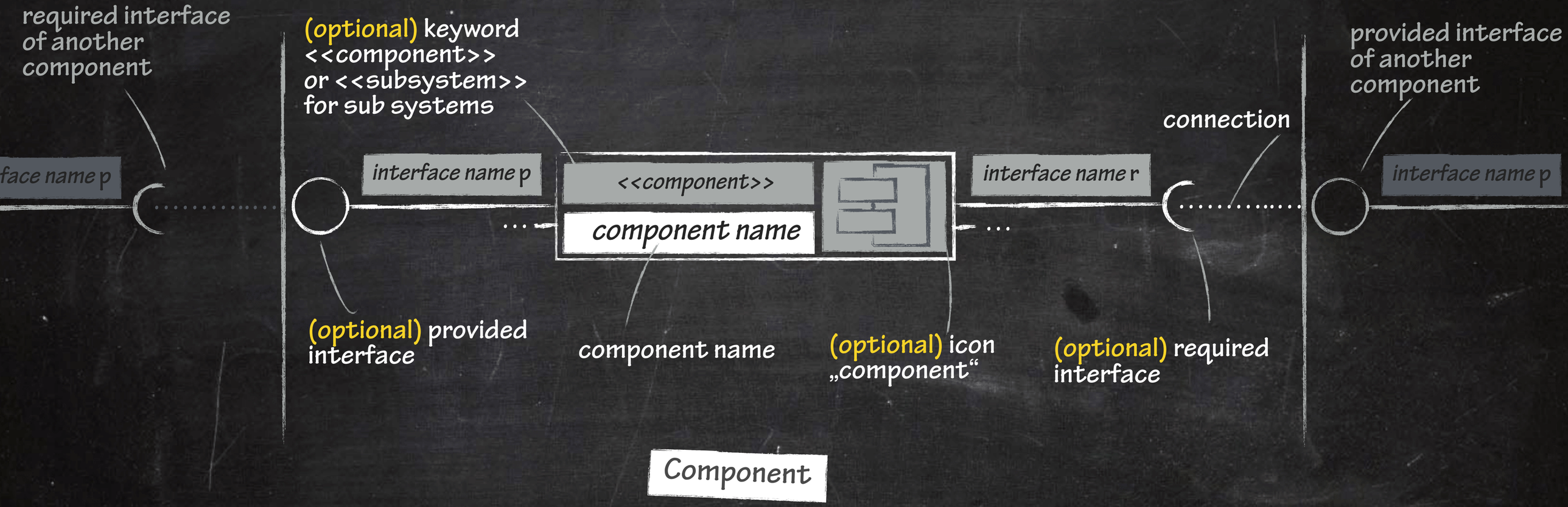
*since UML 2 the concept „component“ refers to autonomous, encapsulated units within a system that expose at least one interface

# UML Structural Diagrams (3)

## Component Diagrams (2)

### Notation and building blocks

Since UML 2.0, the notation is similar to class diagrams (for better scaling)

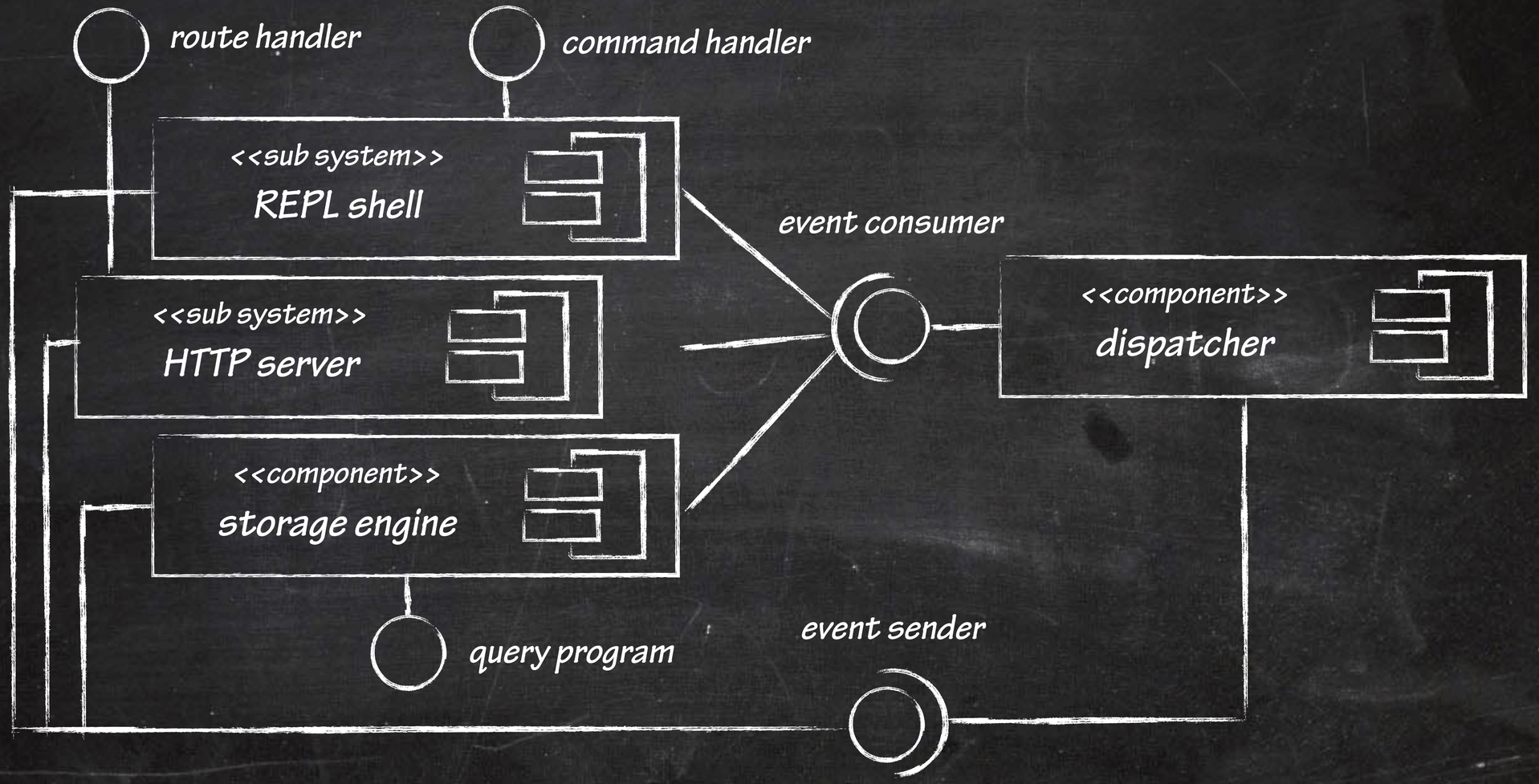



Component

# UML Structural Diagrams (3)

## Component Diagrams (3)

Example



# UML Structural Diagrams (4)

## Composite Diagrams (1)

### Purpose

Composite diagrams:

- show class internal configuration and relationships of ports, parts, and their interaction connectors.
- composite structure fulfills some purpose at runtime by collaboration of interconnected elements.

### Purposes

- Visualize a class' internals structure: properties, parts and relationships
- Visualize class interaction with environment: show ports of a class
- Visualize collaboration and behavior: show functionality and contributors to tasks

# UML Structural Diagrams (4)

# Composite Diagrams (2)

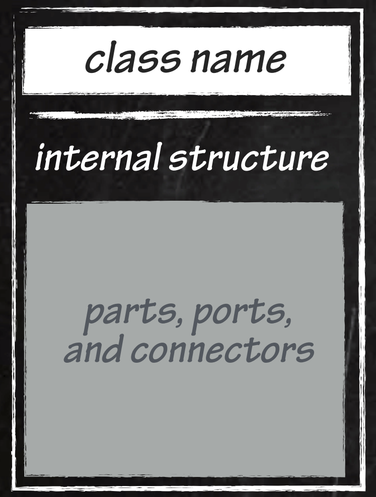
## Notation and building blocks

### Internal Structure Diagrams

### Collaboration Use Diagrams

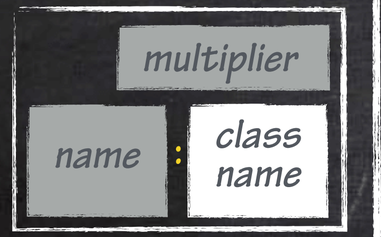
**Structured Classes**

Class in composite diagram




**Parts**

set of class instances plays a certain role at runtime

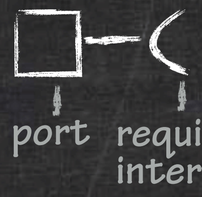



**Ports**

interaction points to connect classes with parts and environment

define exposing & requiring services

requests delegation to internal parts or the containing class




**Connectors**

connects parts, ports, or classes to each other

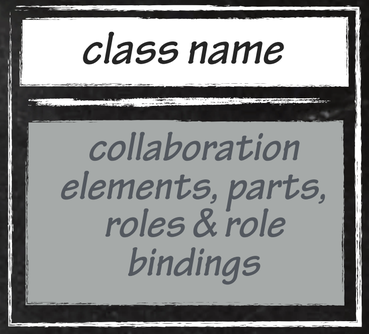
interaction during runtime


**Collaborations**

set of class instances sending messages or invoke operations to fulfill purpose

show (sub-)set of instances by identifying roles for these instances

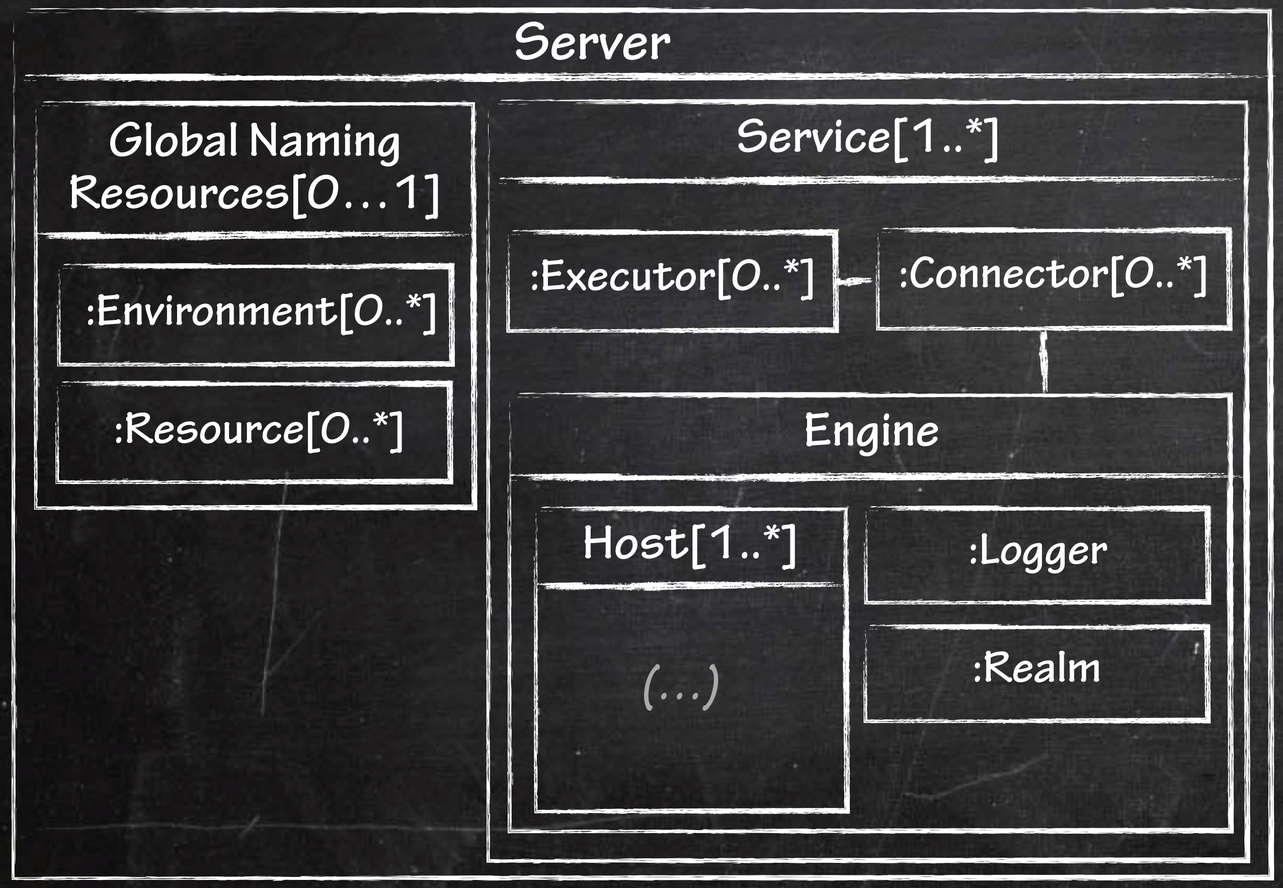

collaboration elements




collaboration using elements

# UML Structural Diagrams (4)

## Internal Structure Diagrams

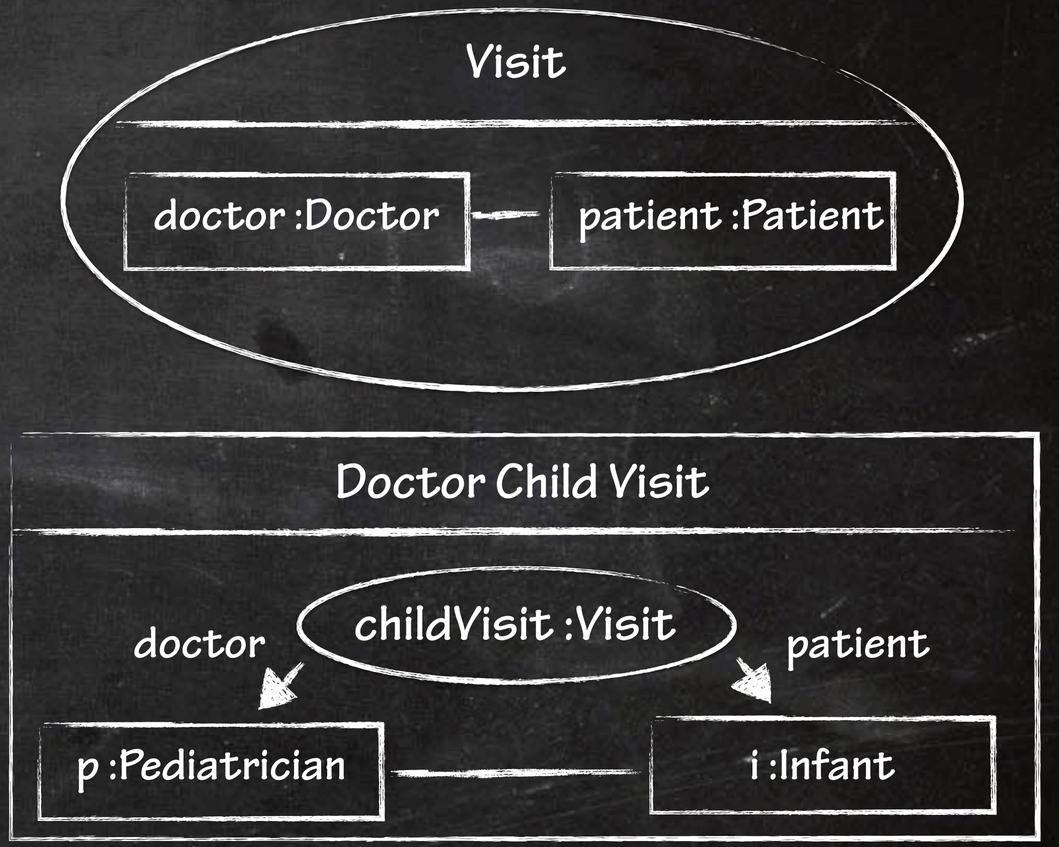



Taken from: http://www.uml-diagrams.org/tomcat-server-uml-composite-structure-diagram-example.html?context=cst-examples

# Composite Diagrams (3)

## Example

## Collaboration Use Diagrams




Taken from: http://www.uml-diagrams.org/composite-structure-diagrams.html

# UML Behavior Diagrams (3)

## State Machine Diagrams (1)

### Purpose

State machine diagrams:

- adaption of finite automaton concepts in UML (cf. Mealy and Moore machines from Theoretical Computer Science)
- abstract machine (system, processes,...) accepts or rejects sequence of inputs while internally transiting from one internal state to another
- States and transitions between states is well-defined
  - State transition depends on input and current active state
- Two types of state machines
  - behavior state machines: model class instance behavior
  - protocol state machines: model protocol usage for classes, interfaces & ports

Purposes

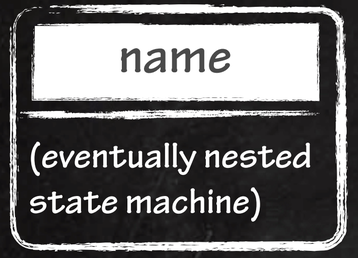
- Support event-driven system design (external, internal event e.g., button press, timeout,...)
- Visualize actions depending on system states and triggering events (cf. Mealy Machine)
- Visualize entry and exit actions depending on states rather transitions (cf. Moore Machine)

# UML Behavior Diagrams (3)

# State Machine Diagrams (2)

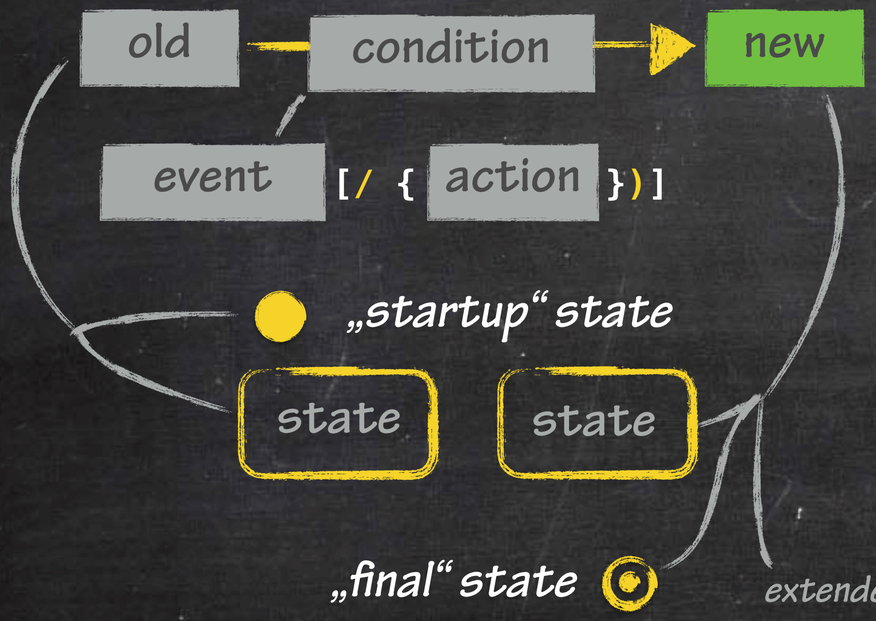
## Notation and building blocks

Similar to visualization of finite automatons in Theoretical Computer Science (i.e,. directed graph where nodes are states, and edges are state transitions)




Either a simple (numeric) value indicating the current state, or a class implementing the systems behavior & internal variables (extended state machine)

**State**



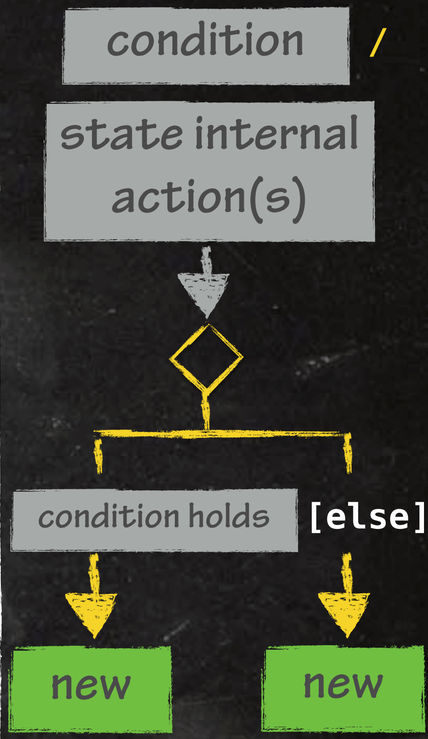

**State Transition**

Something that occurs, have parameters and affects the system (e.g., key stroke)

Event life cycle

- Received: added to event queue.
- Dispatched: popped from event queue, processed by state machine
- Consumed: event instance is deleted

**Event**



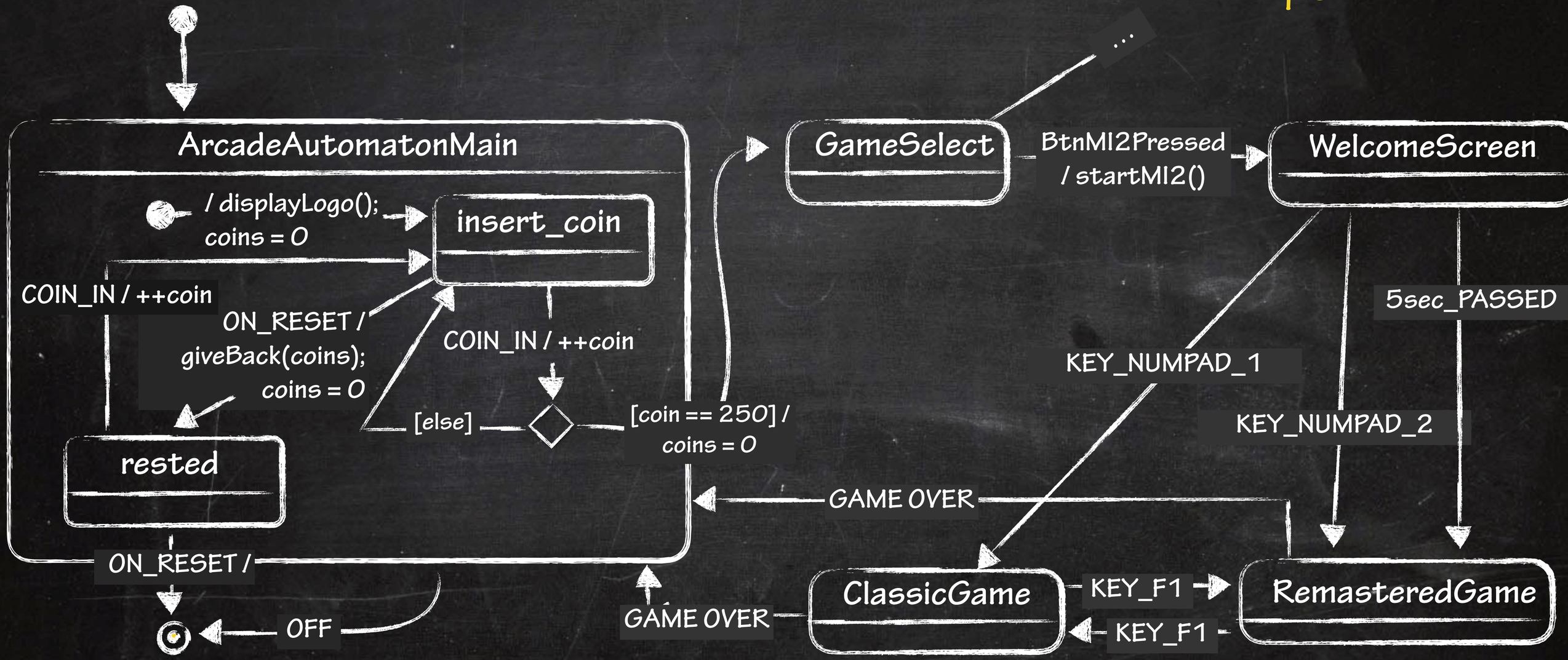

**Extended State**

# UML Behavior Diagrams (3)

# State Machine Diagrams (3)

(Extended State Machine)

**Example**



# UML Interaction Diagrams (1)

## Sequence Diagrams (1)

### Purpose

Sequence diagrams:

- Show interactions between objects (with limited lifetime) in sequential order:
  - Usage scenarios: describes ways to use the system in several use cases
  - Function logic: describes the logic of (complex) functions or operations
  - Service logic: describes interaction between/with web-services, broker,…

### Purposes

- Visualize interactions between objects in sequential order in which those interactions occur

# UML Interaction Diagrams (1)

## Sequence Diagrams (2)

### Notation and building blocks

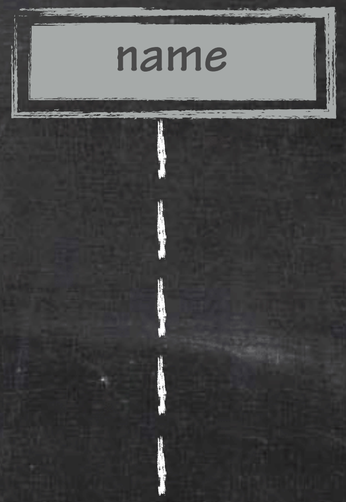
classes or instances (w/o attributes)


Class Box

represents the time needed for a task done by a certain class

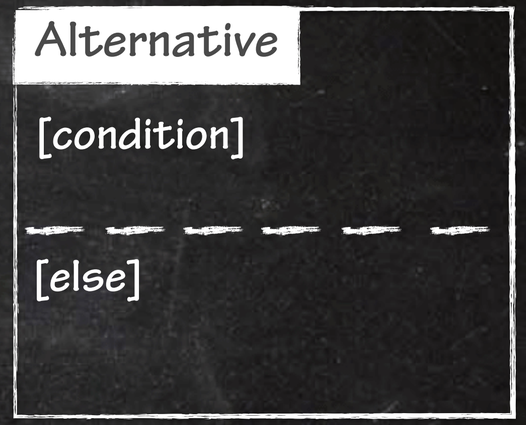

Activation Box

represents sequential order on events that affect an object




Life Line

represents choice between one or more event sequences




Alternative

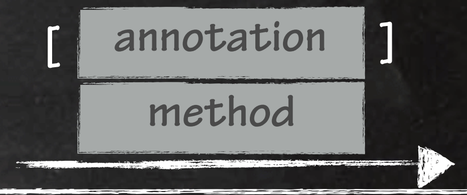
# UML Interaction Diagrams (1)

## Sequence Diagrams (3)

### Notation and building blocks (2)

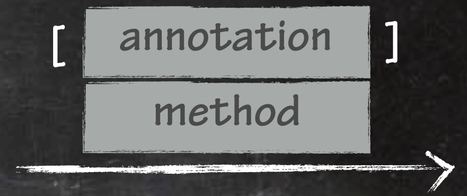
**Synchronous Message**

blocking: sender must wait for response before program flow continues



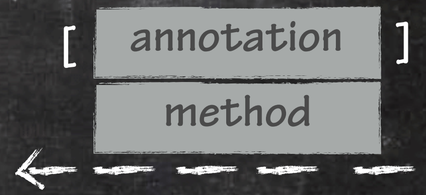

**Asynchronous Message**

non-blocking: sender program flow continues without waiting for response




**Reply Message**

indicates that a response to an synchronous/ asynchronous message is available




**Asynchronous Create Message**

sent to lifelines in order to create those

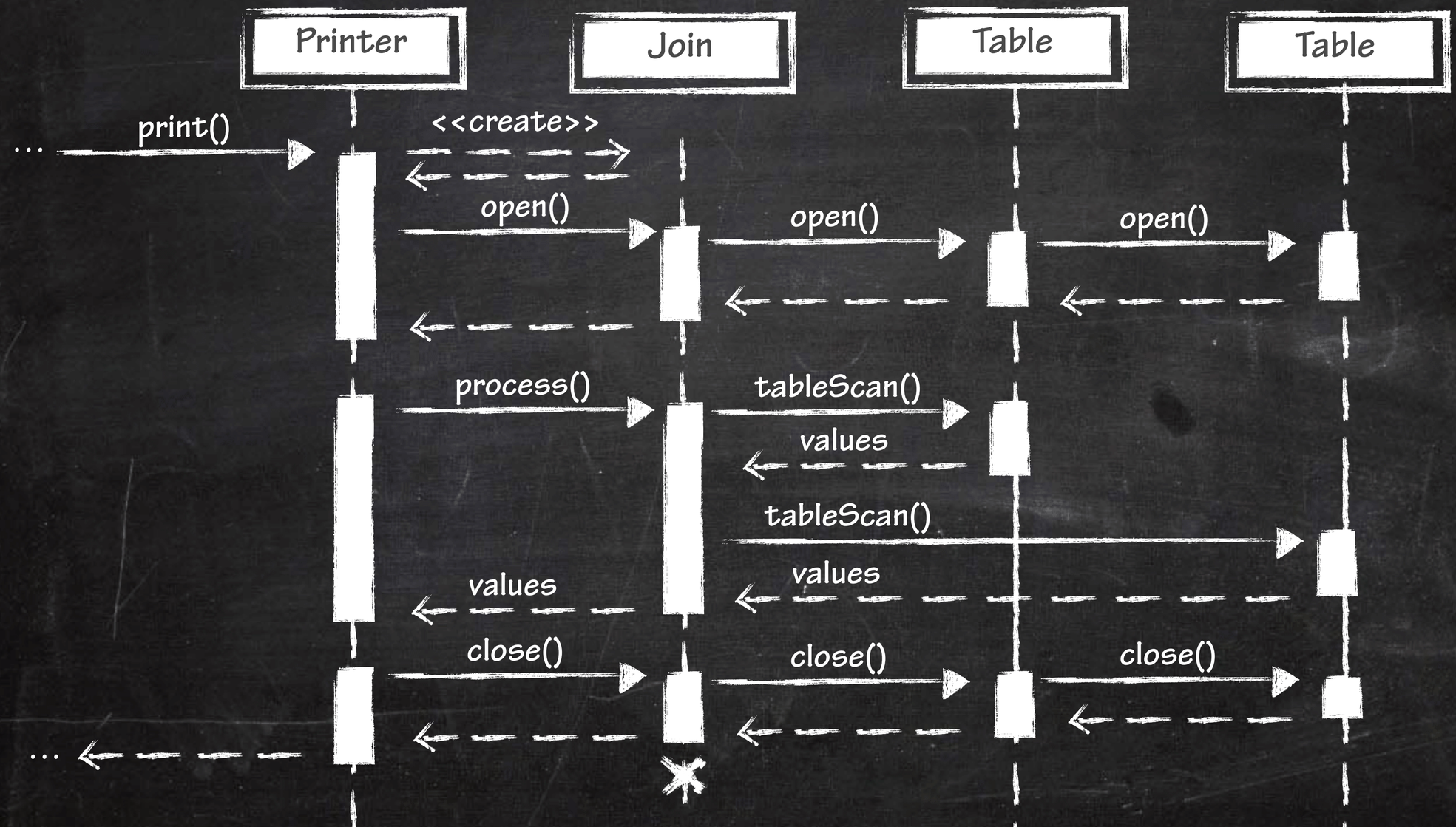

**Delete Message**

sent to lifelines and indicates the object's destruction

# UML Interaction Diagrams (1)

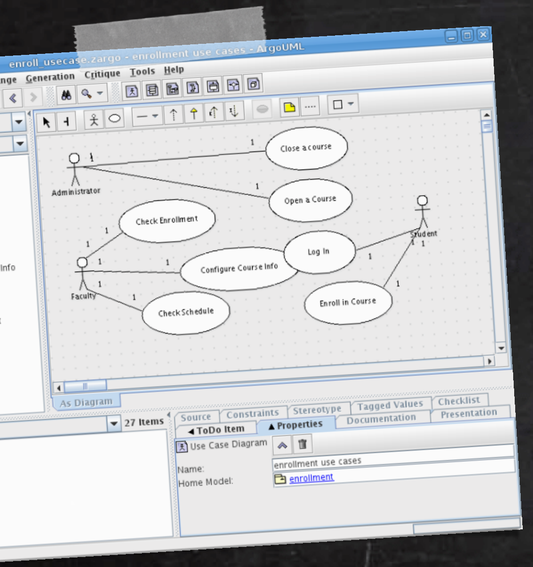
## Sequence Diagrams (3)

### Example

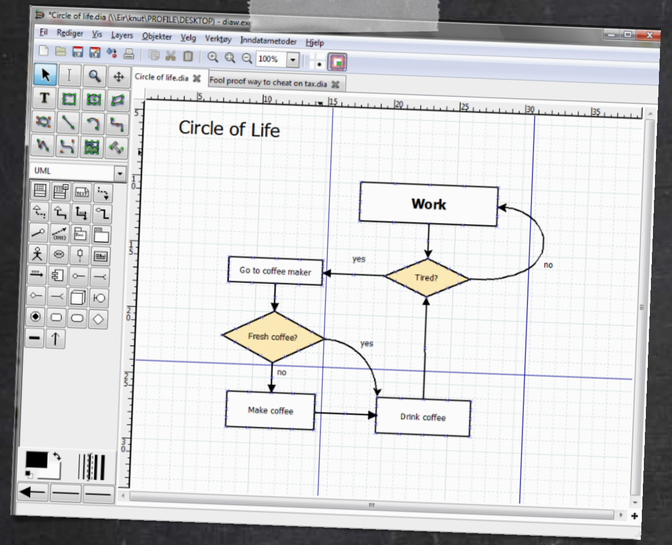



Example: pull-based operator-at-a-time processing

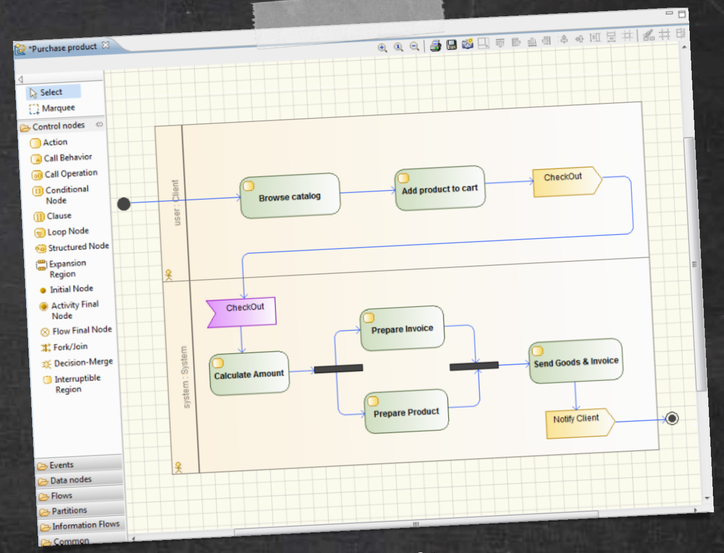
# TOOLING



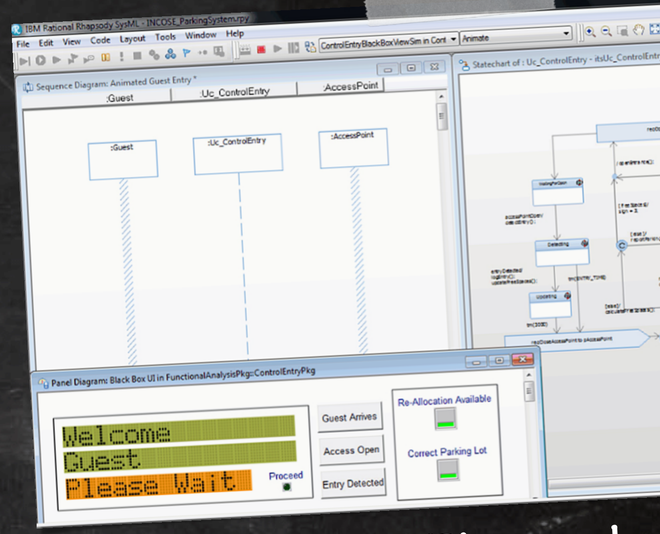

ArgoUML
(Cross-Platform, OpenSource)




Dia
(Cross-Platform, OpenSource)




Modelio
(Windows & Linux, OpenSource)

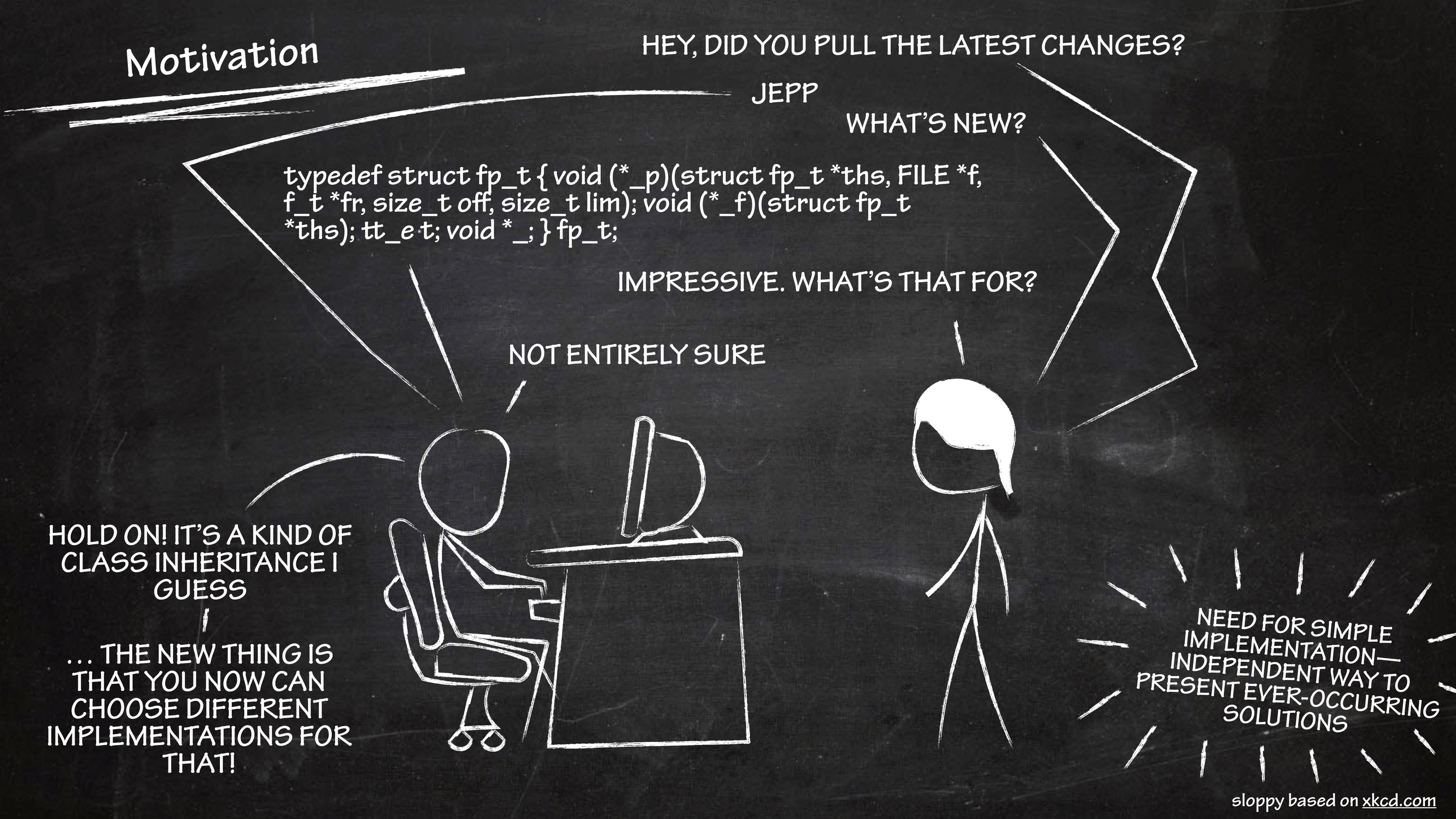



Rational Rhapsody
(Windows, Linux & macOS, Comm

see more: https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools
or google for it: uml software tools


# DESIGN PATTERNS

# Motivation




sloppy based on xkcd.com


# DOCUMENTATION

# Motivation

OH, THE FUNCTION *BIND* IS WHAT I'M LOOKING FOR

LET'S SEE WHAT IT DOES AND HOW TO USE

```
44
45
46
47
48
49
50
51  result = bind(|
```

DOCUMENTATION

DECLARED IN: bind.h

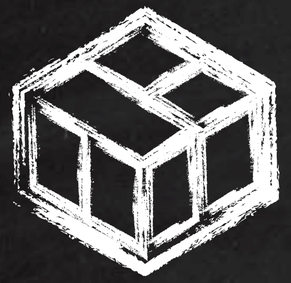
int bind(void *, const void **)

...

THANKS.

NEED FOR EXPLANATION

sloppy based on xkcd.com



# Chapter 1
# A Full-Stack Engineer's Basic Kit


## In the lecture...

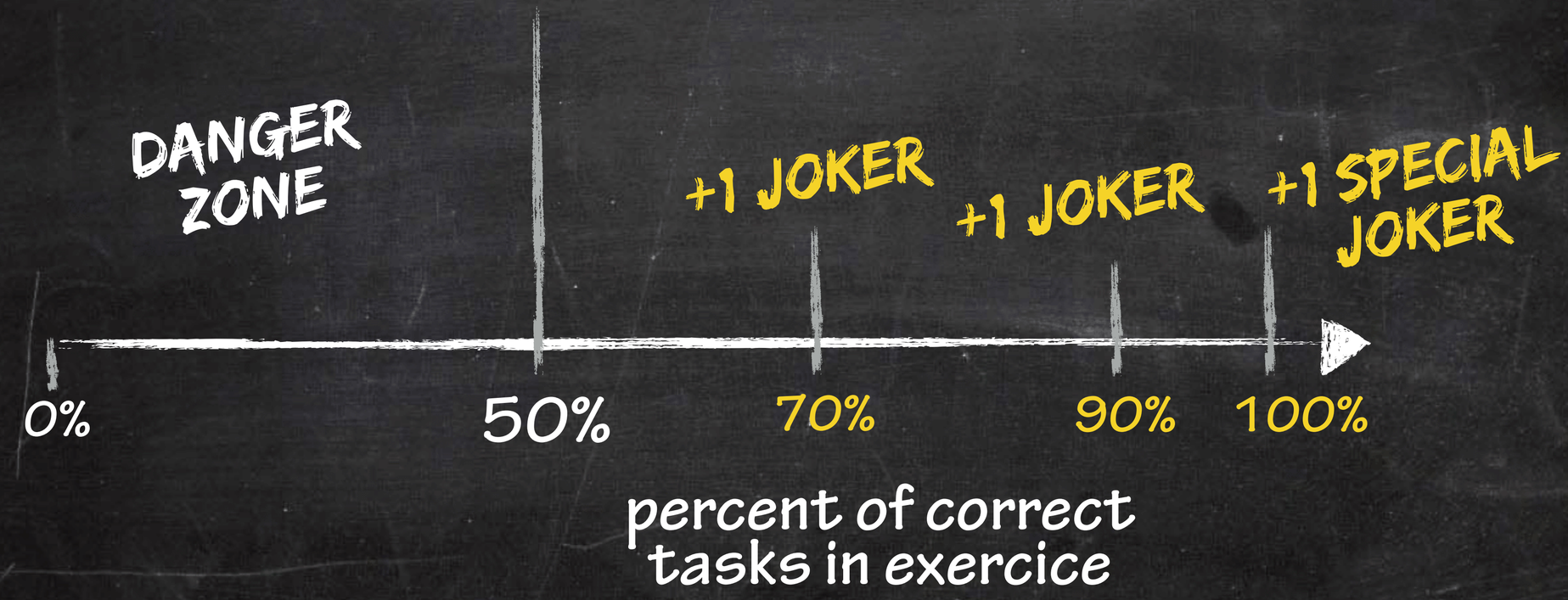
- ✓ What are Version Control Systems
- ✓ Centralized vs Decentralized VCS
- ✓ Git Local Repository Handling
- ✓ Git Remote Repository Handling
- ✓ Best Practice on Commit Messages
- ✓ Branching, Diffs, Merge-Conflicts
- ✓ Fast-Forward vs Non-Fast-Forward
- ✓ Pull-Requests, Merge and Rebase
- ✓ Tagging and Git-Ignore files
- ✓ Most useful Git commands
- ✓ Idea & motivation for branching workflows
- ✓ Four popular branching workflows, incl. forking
- ✓ Fundamental concepts in project management
- ✓ Reasons for failed projects
- ✓ Non-agile vs agile development methodologies
- ✓ Waterfall Model, Spiral Model
- ✓ Agile methods characteristics, pro's and con's
- ✓ Modeling languages, why to use, pro's and con's
- ✓ UML, 3 types of diagrams & representatives
- ✓ Purpose,building blocks of seven diagrams

## ... from the exercise

- ✓ Git setup, GitHub usage, pull requests
- ✓ Fork, cloning repositories
- ✓ Basic and advanced Git operations: committing one out of two Ignoring files, merging vs rebasing, rewriting history, stash your work
- ✓ Basic understanding of compiler & linker, build automation, build automation generators, libraries, IDEs, debugger, memory debugger, profiler, test frameworks, code style formatter
- ✓ Most important design patterns
- ✓ Code documentation basics

# INTERMEZZO

## REGARDING THE JOKERS



# INTERMEZZO (2)

GETTING SOMETHING SWEET FOR PRESENTING IN EXERCISE AT THE END