

Formal System Design Process with UML

Use a formal process & tools to facilitate and automate design steps:

Requirements

Specification

System architecture

Coding/chip design

Testing

Text: Chapter 1.3,1.4

Unified Modeling Language (UML)

- Developed by Grady Booch et al.
 - Version 1.0 in 1997 (current version 2.4.1)
 - Maintained by Object Management Group (OMG) – www.omg.org
 - Resources (tutorials, tools): www.uml.org
- Goals:
 - object-oriented (OO);
 - visual;
 - useful at many levels of abstraction;
 - usable for all aspects of design.
- Encourage design by **successive refinement**
 - Don't rethink at each level
 - CASE tools assist refinement/design

Emphasis of OO Design

- Describe system/design as **interacting objects**
- **Object** = state + methods.
 - **State** defined by set of “attributes”
 - each object has its own identity.
 - user cannot access state directly
 - **Methods** provide an **abstract interface** to the object attributes.
 - Objects map to system HW/SW elements
- Also model the **outside world** (users, machines, environment) **interactions** with the system

UML Elements

- **Model elements**
 - objects, classes, interfaces, components, use cases, etc.
- **Relationships**
 - associations, generalization, dependencies, etc.
- **Diagrams**
 - class diagrams, use case diagrams, interaction diagrams, etc.
 - constructed of model elements and relationships

UML Diagram Types

- **Use-case**: help visualize functional requirements (user-system interaction)
- **Class**: types of objects & their relationships
- **Object**: specific instances of classes
- Interaction diagrams (dynamic)
 - **Sequence**: how sequences of events occur (message-driven)
 - **Collaboration**: focus on object roles
- **Statechart**: describe behavior of system/objects
- **Activity**: flow of activities in a process (flowchart)
- **Component**: physical view of system (code, HW)

Structural vs. Behavioral Models

- **Structural:** describe system components and relationships
 - static models
 - objects of various classes
- **Behavioral:** describe the behavior of the system, as it relates to the structure
 - dynamic models

Objects and classes

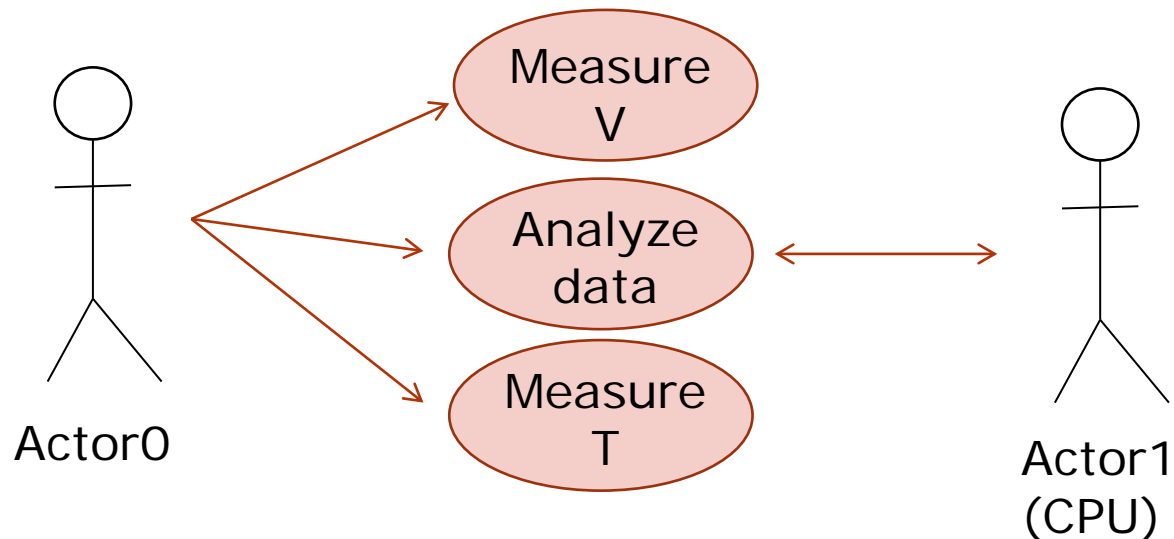
- **Class**: object type.
- Class defines:
 - the object's **state** elements;
 - State values may change over time.
 - Each object has its own state.
 - Elements not directly accessible from outside
 - the **methods** (operations) used to interact with all objects of that type.
 - State elements accessed through methods

OO design principles

- Some objects will closely correspond to real-world objects.
- Other objects may be useful only for description or implementation.
 - “**abstraction**” – list only info needed
(select info listed for an object for each given purpose)
- Objects provide interfaces to read/write the object state, hiding the object’s implementation from the rest of the system.
 - “**encapsulation**” – mask internal op’s/info

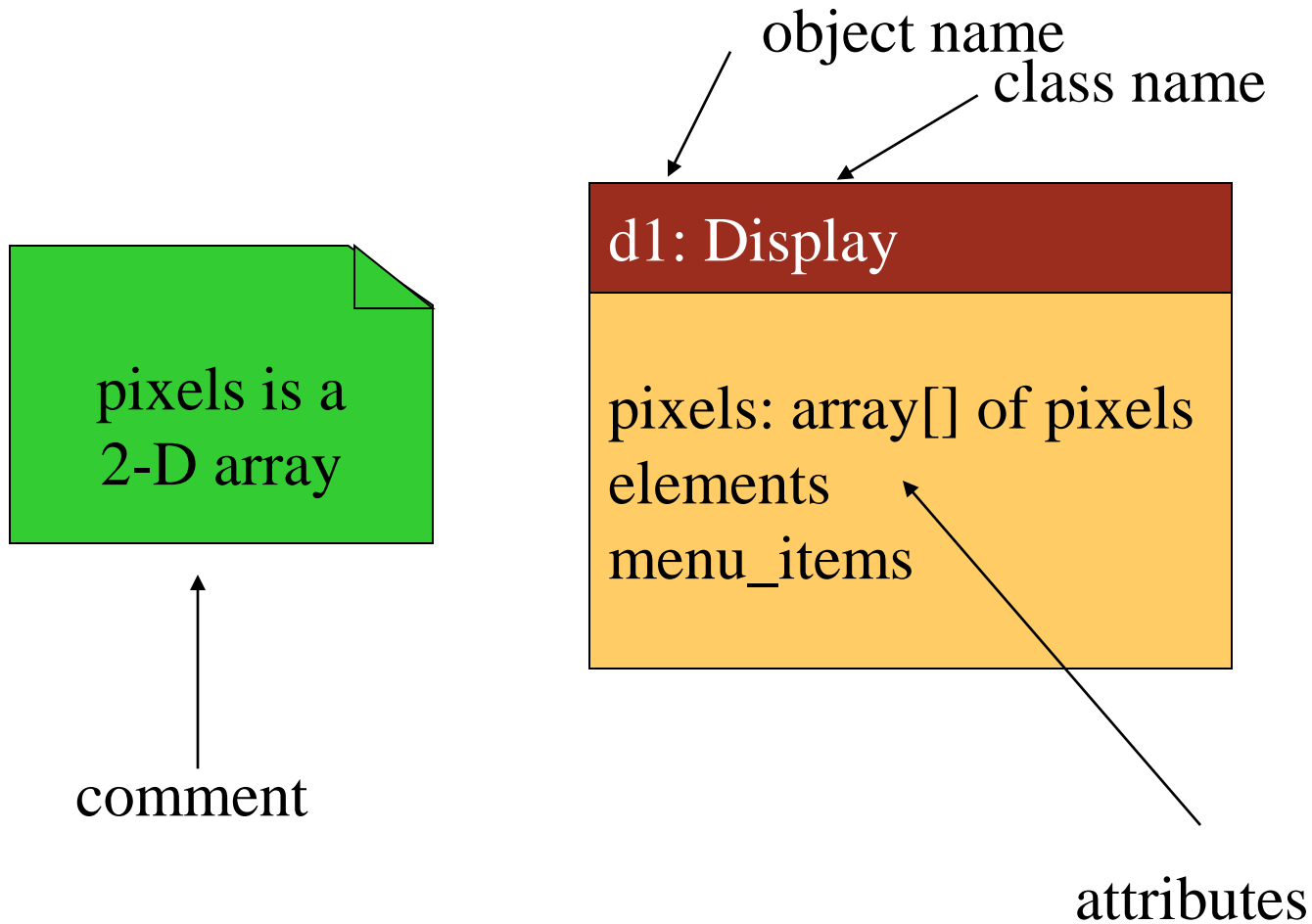
UML use-case diagrams

- Describe behavior user sees/ expects
- Describe user **interactions** with system objects
- Users = “**actors**” (anyone/thing using the system)

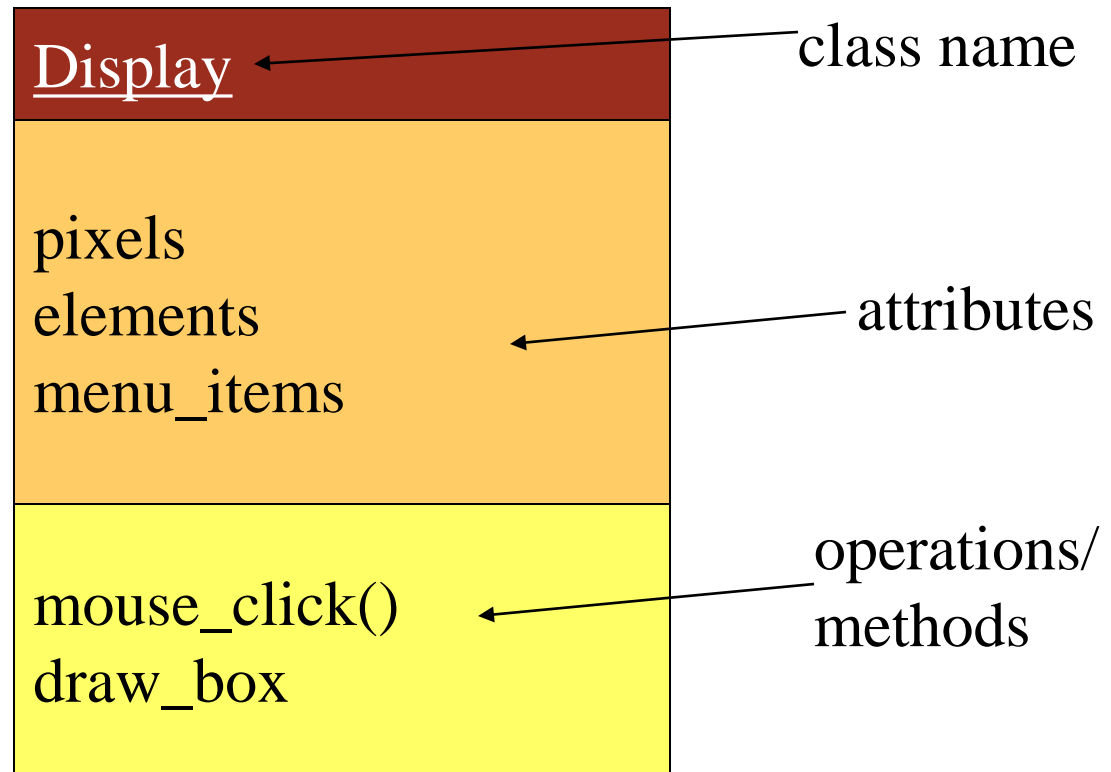


- Translate to algorithm for system design

UML object



UML class



The class interface

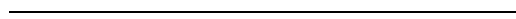
- Operations provide the abstract interface between the class' implementation and other classes.
- Operations may have arguments, return values.
- An operation can examine and/or modify the object's state.
- Implementation of the object is hidden by the class interface (encapsulation)

Choose your interface properly

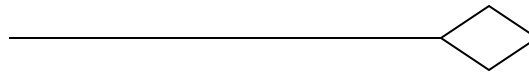
- If the interface is too small/specialized:
 - object is hard to use for even one application;
 - even harder to reuse.
- If the interface is too large:
 - class becomes too cumbersome for designers to understand;
 - implementation may be too slow;
 - spec and implementation are probably buggy.

Relationships between objects and classes

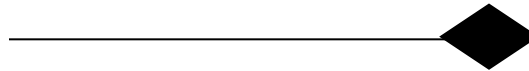
- **Association**: objects “related” but one does not own the other.



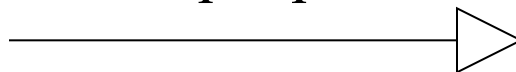
- **Aggregation**: a complex object is made of several smaller objects.



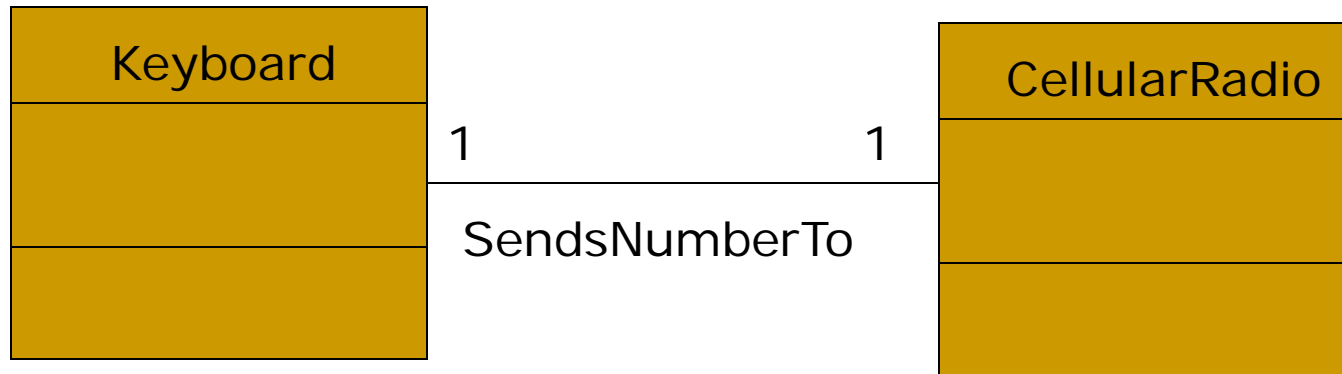
- **Composition**: strong aggregation: part may belong to only one whole – deleting whole deletes parts.



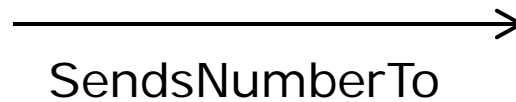
- **Generalization**: define one class in terms of another. Derived class inherits properties.



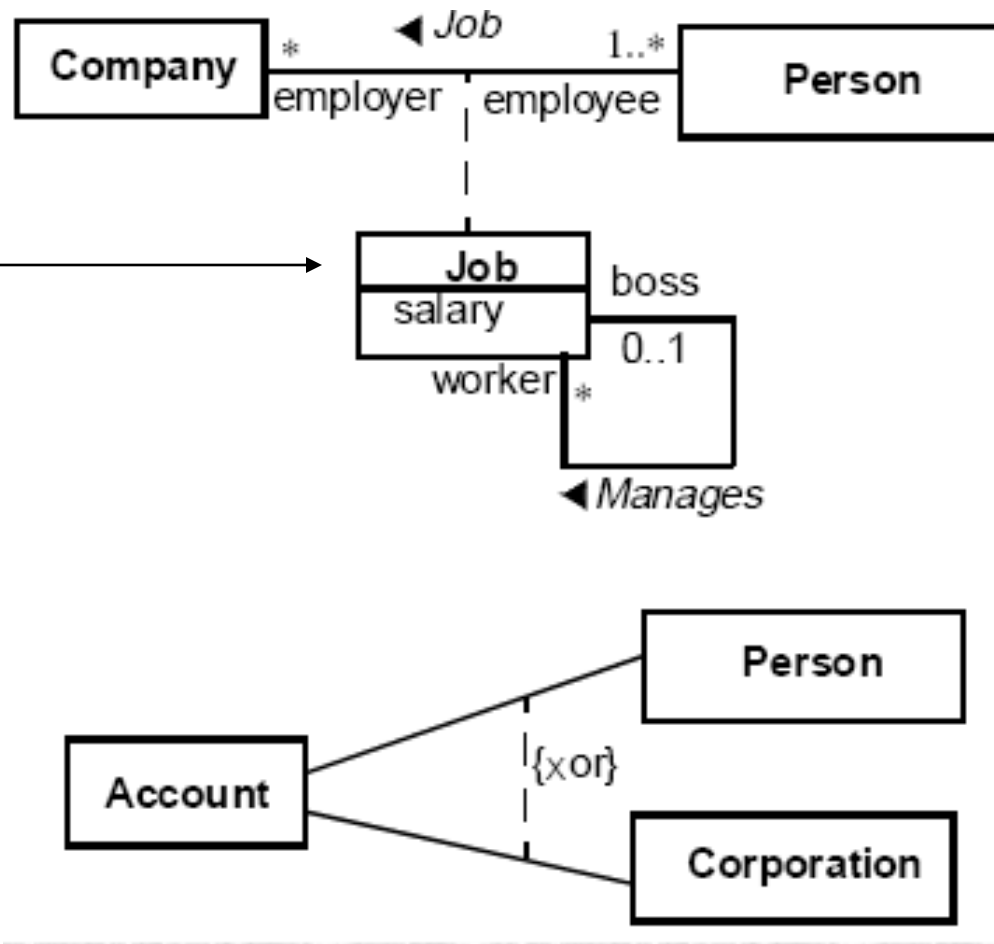
Association Example



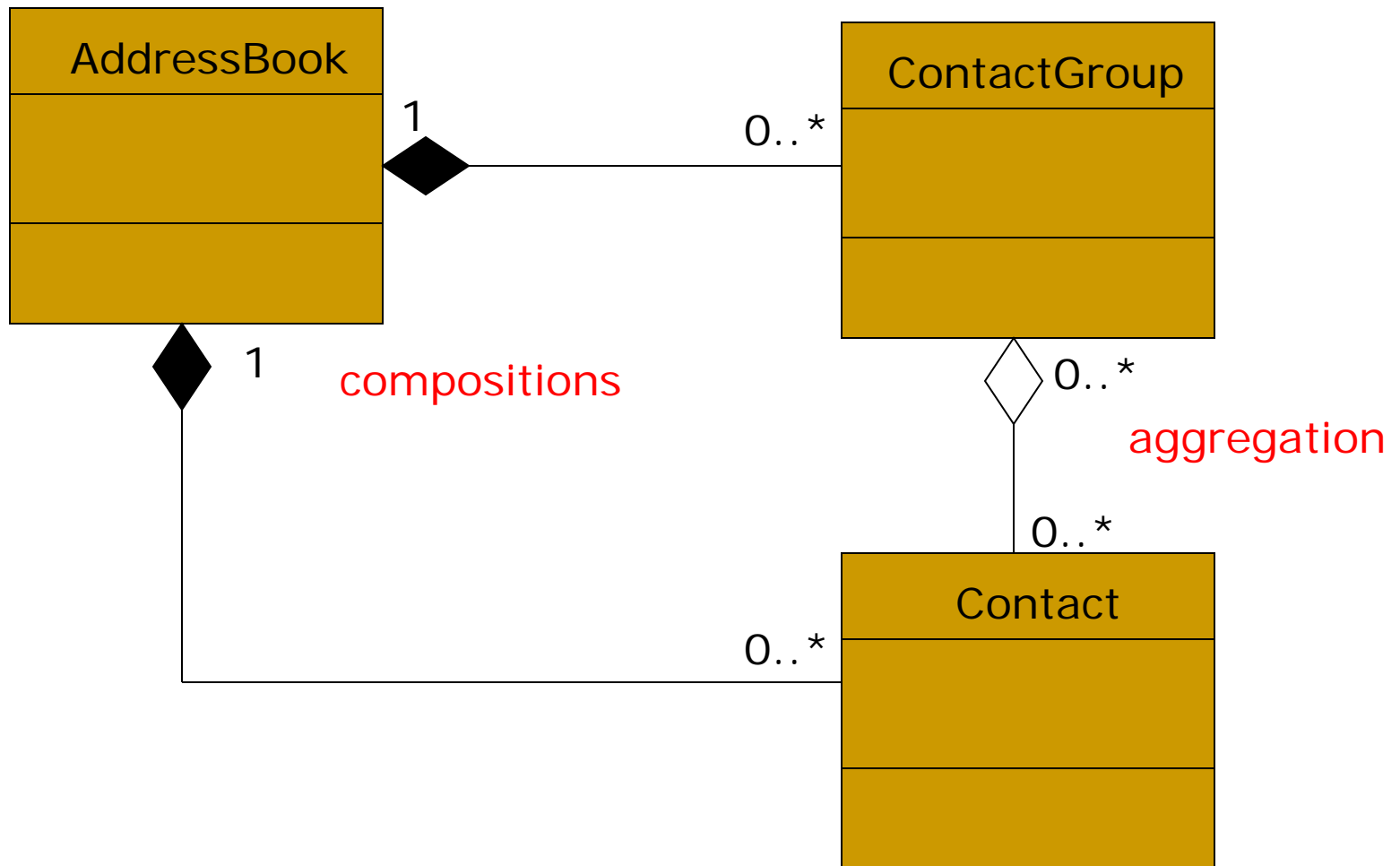
Optionally – show “direction” of association



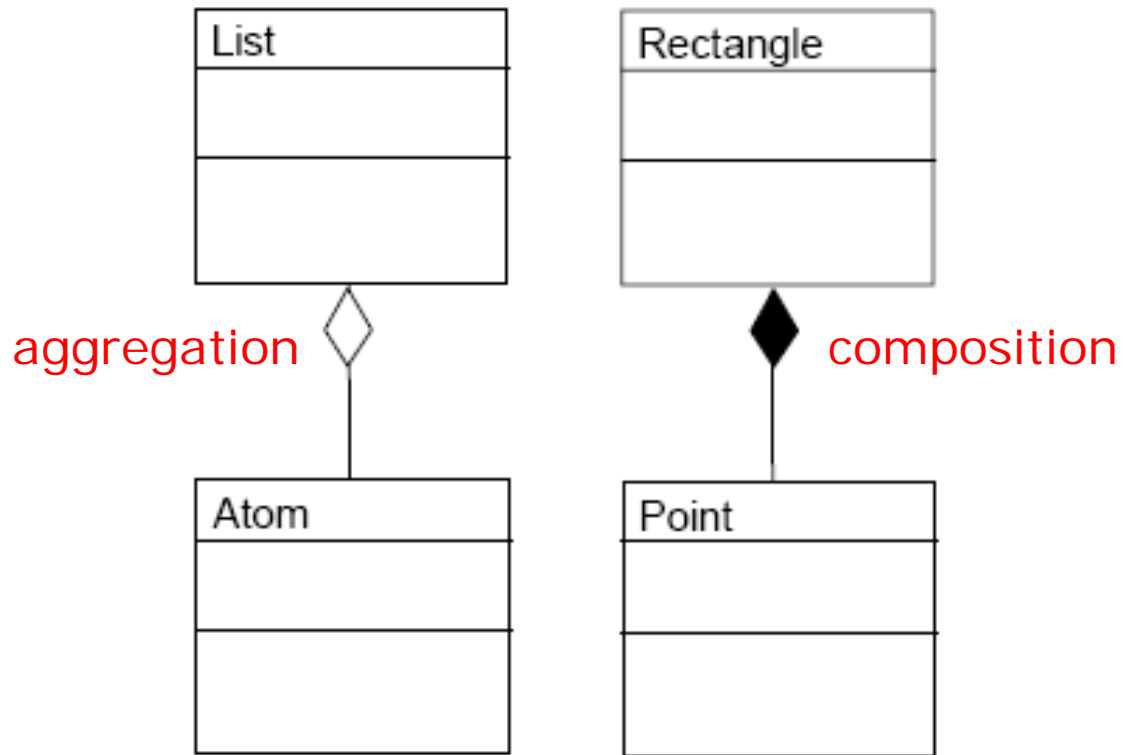
Association Object



Aggregation/Composition Examples

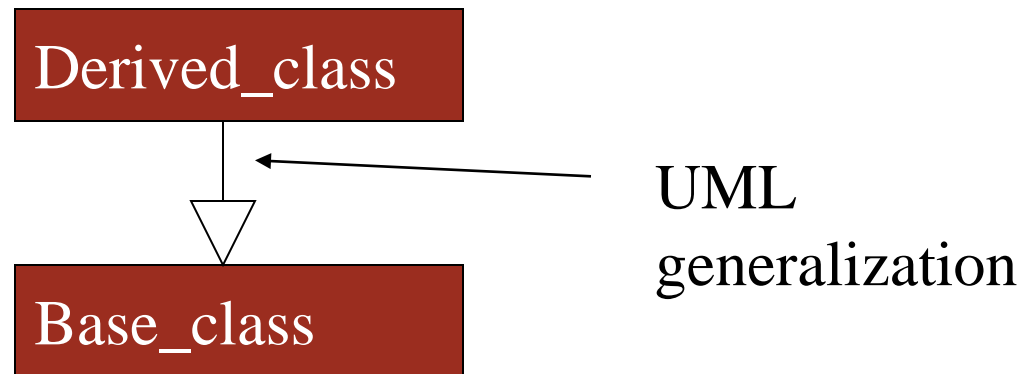


Examples

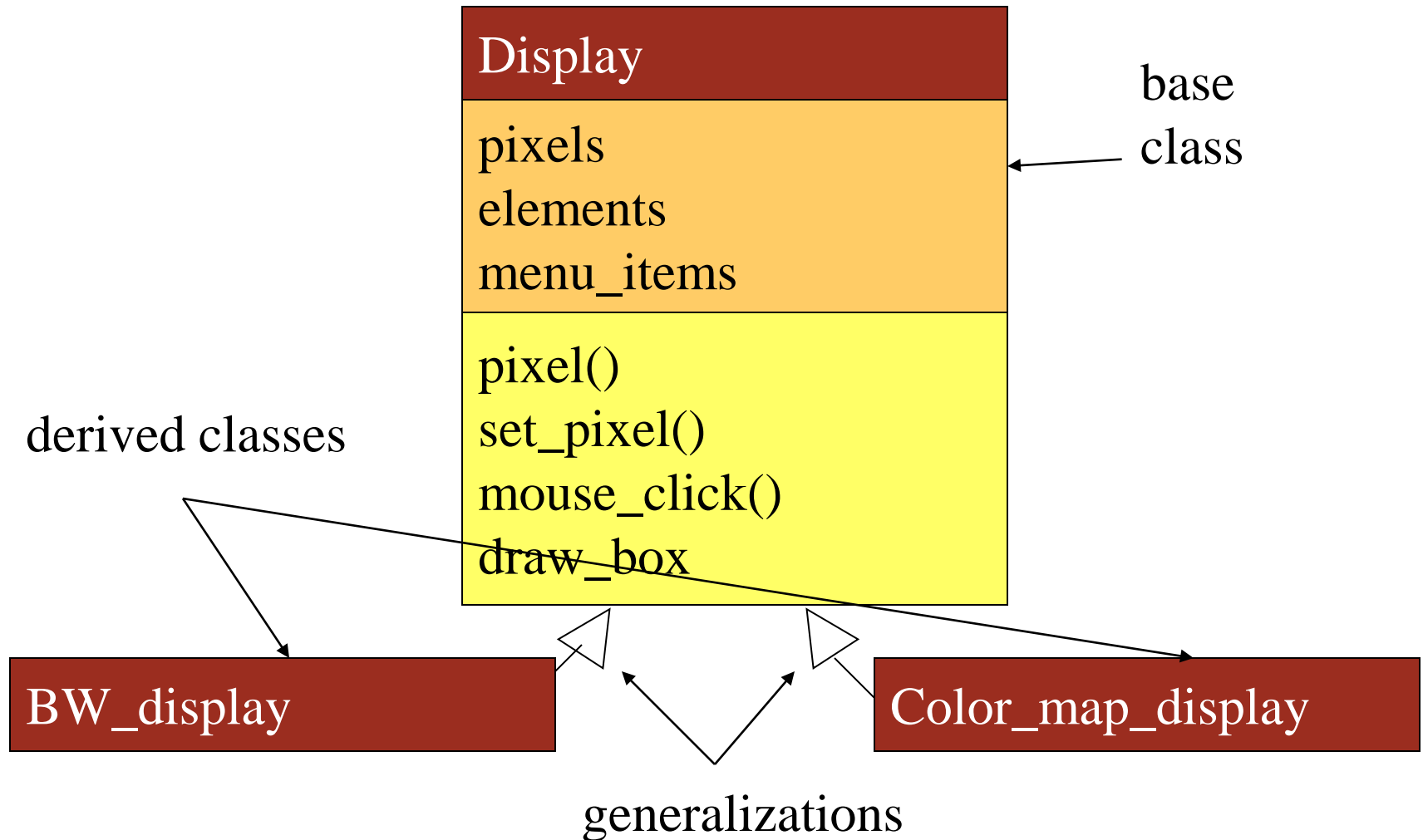


Generalization/Class derivation

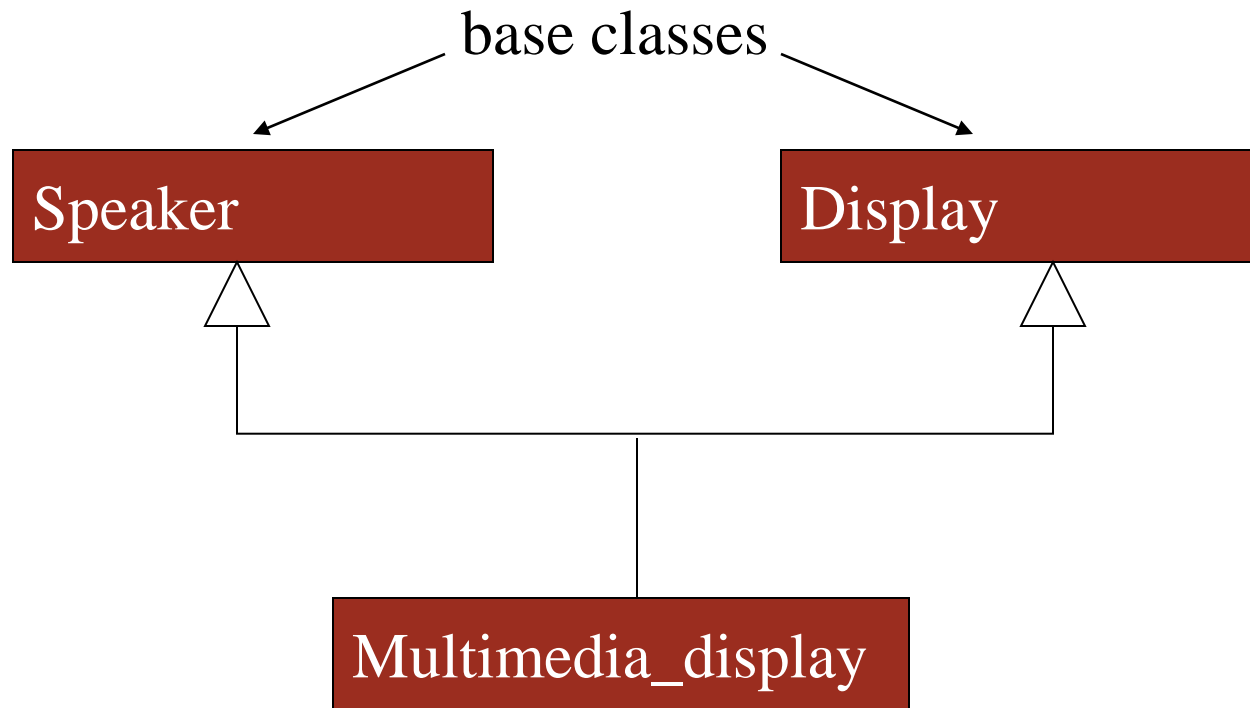
- May want to define one class in terms of another more “general” class.
- Derived class **inherits** attributes & operations of base class.



Class derivation example

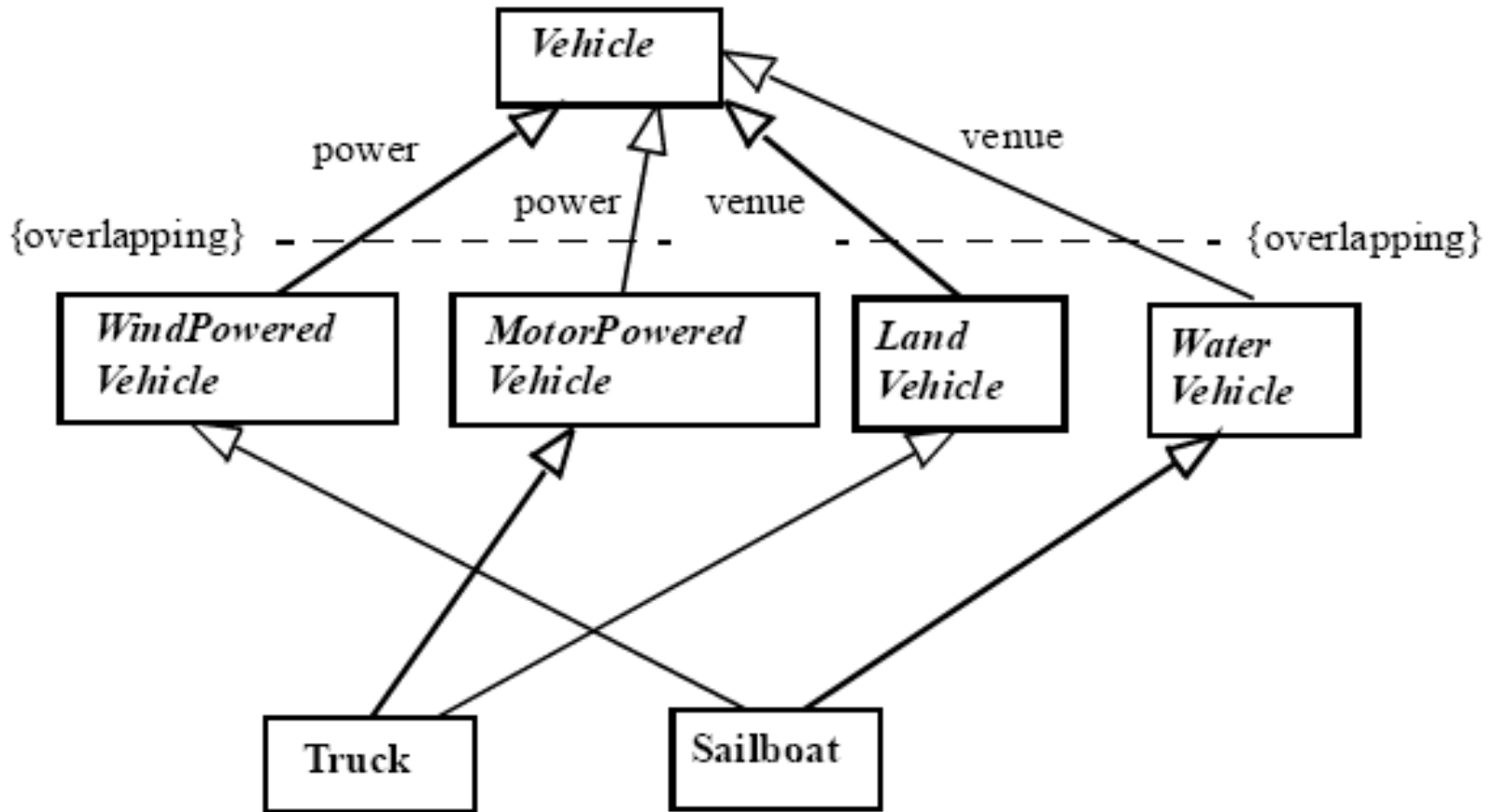


Multiple inheritance

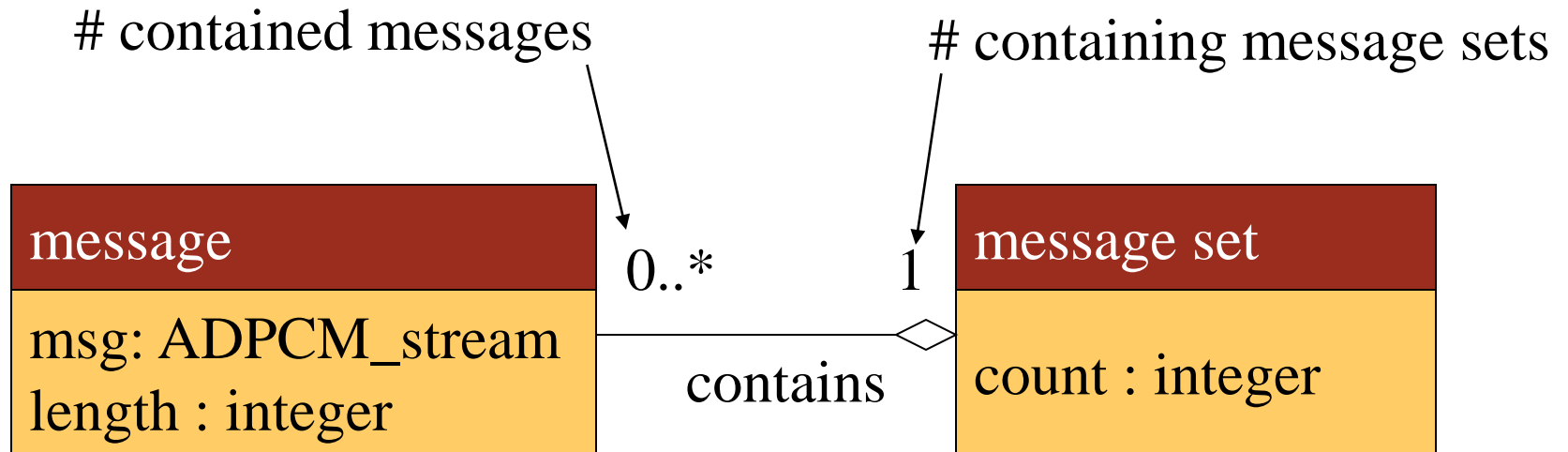


derived class inherits properties of both base classes

Generalization example



Association example



ADPCM: adaptive differential pulse-code modulation

Links and associations

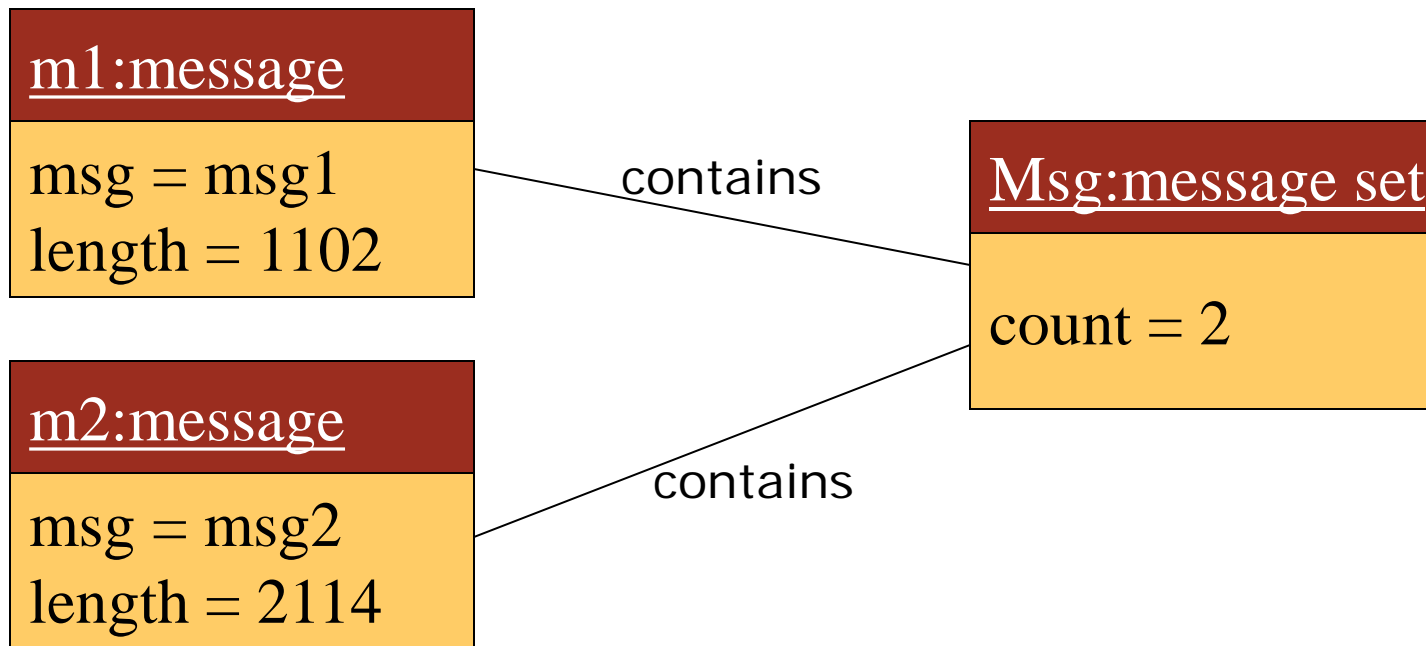
- **Association**: describes relationship between **classes**.
- **Link**: describes relationships between **objects**.

Association is to link as class is to object.

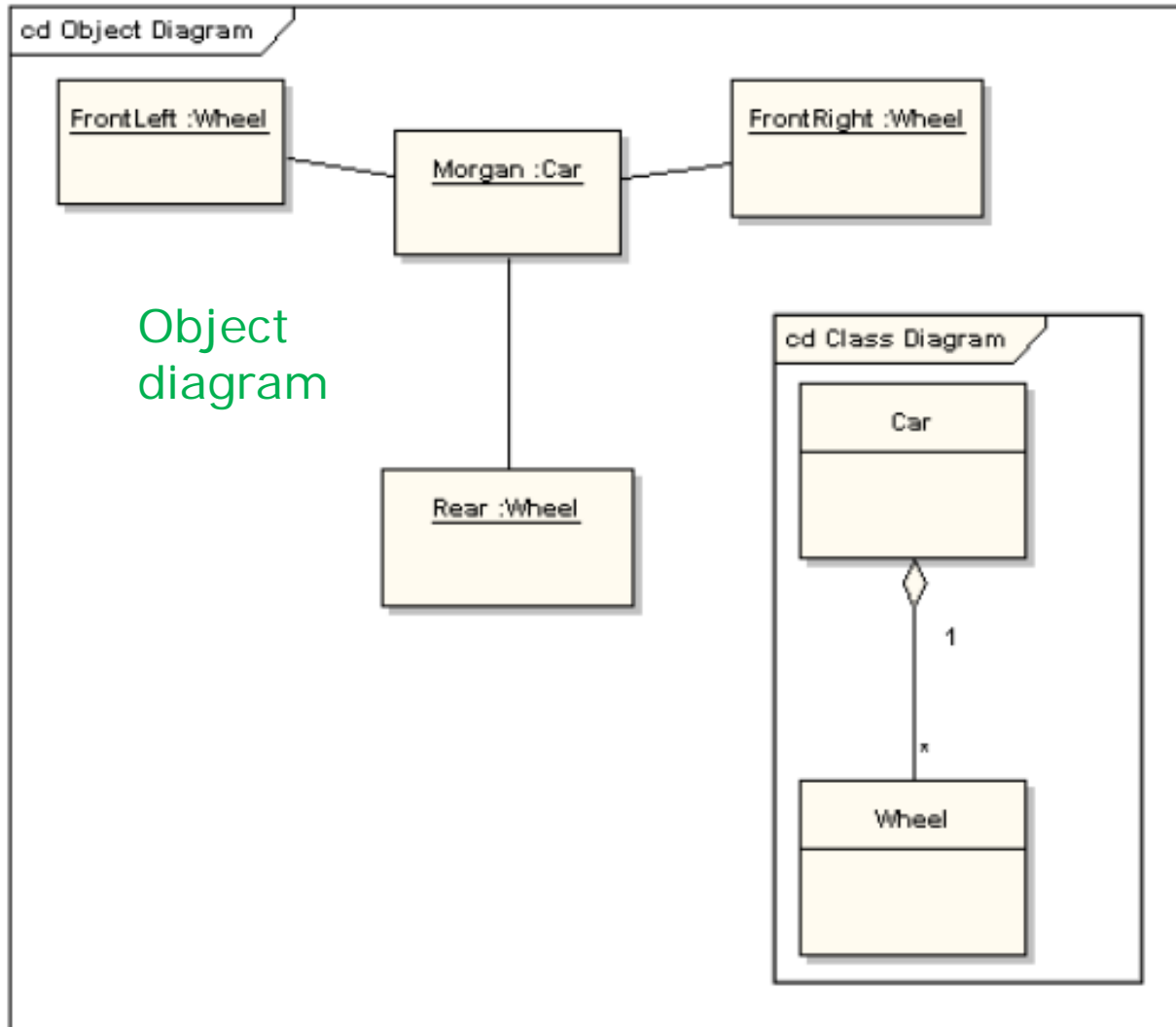
- Association/class = abstract
- Link/object = physical

Link example

- Link defines the **contains** relationship:



Object & Class Diagram Example



Object
diagram

Class
diagram

OO implementation in C++

```
/* Define the Display class */
class Display {
    pixels : pixeltype[IMAX,JMAX];
public:
    Display() { } /* create instance */
    pixeltype pixel(int i, int j) {
        return pixels[i,j]; }
    void set_pixel(pixeltype val, int i,
        int j) { pixels[i,j] = val; }
}
```

Instantiating an object of a class in C++

```
/*instantiate Display object d1*/
```

```
Display d1;
```

```
/* manipulate object d1 */
```

```
apixel = d1.pixel(0,0);
```

object

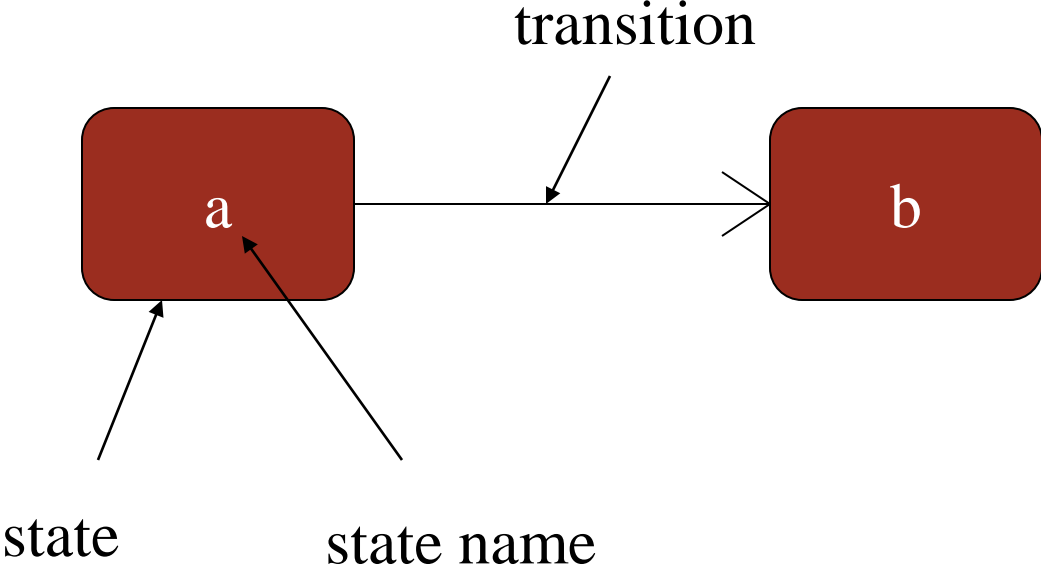


method

Behavioral description

- Several ways to describe behavior:
 - internal view;
 - external view.
- Dynamic models:
 - **State diagram**: state-dependent responses to events
 - **Sequence diagram**: message flow between objects over time
 - **Collaboration diagram**: relationships between objects

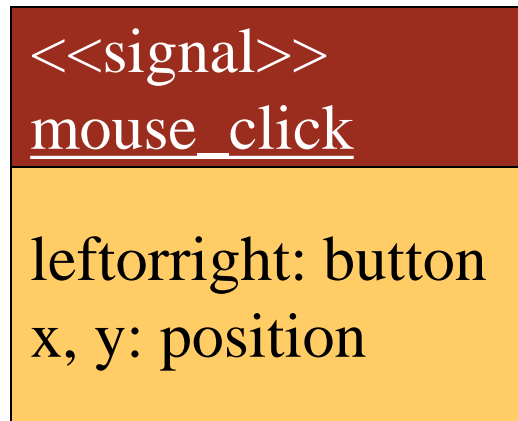
State machines



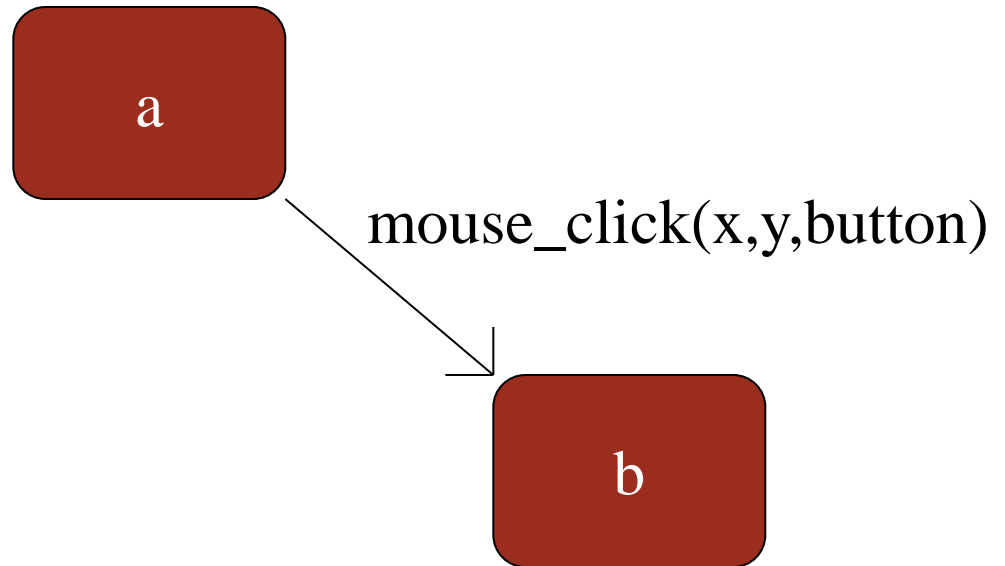
Event-driven state machines

- Behavioral descriptions are written as **event-driven** state machines.
 - Machine changes state when receiving an input.
- An event may come from inside or outside of the system.
 - **Signal**: asynchronous event.
 - **Call**: synchronized communication.
 - **Timer**: activated by time.

Signal event

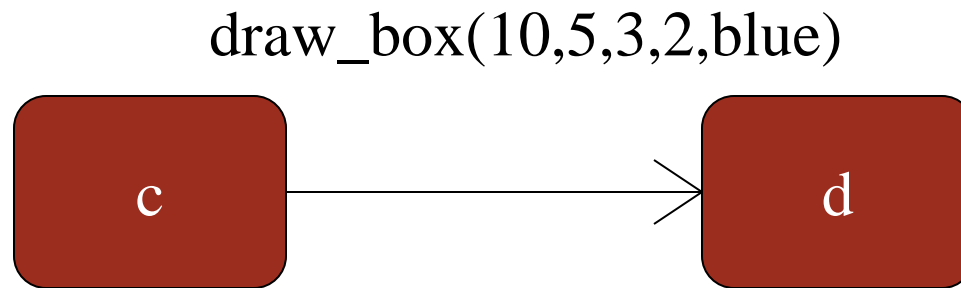


declaration

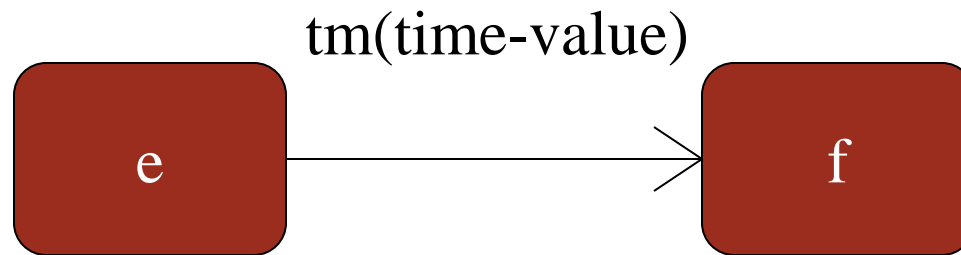


event description

Call event

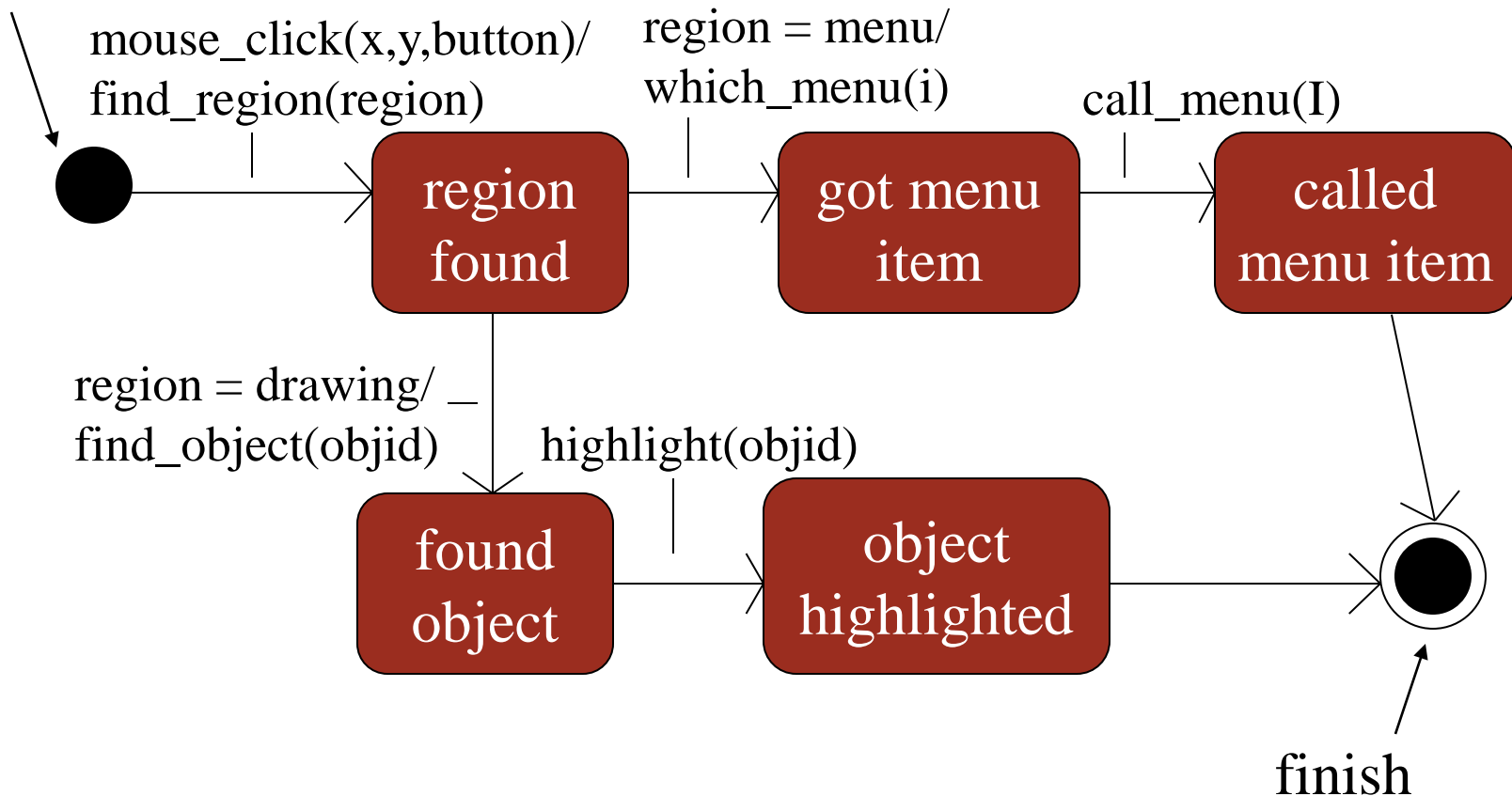


Timer event



Example: click on display

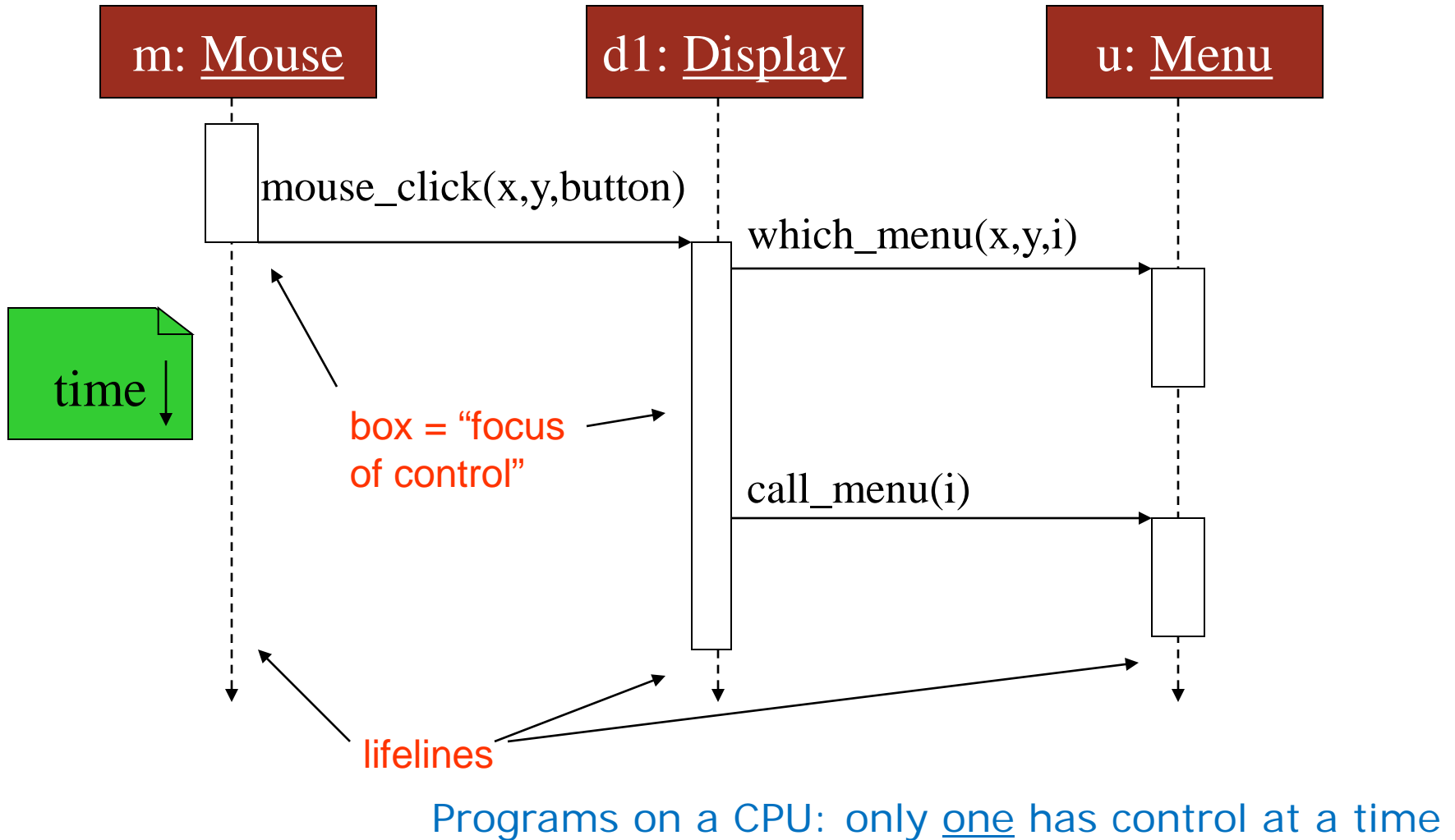
start



Sequence diagram

- Shows sequence of operations over time.
- Relates behaviors of multiple objects.

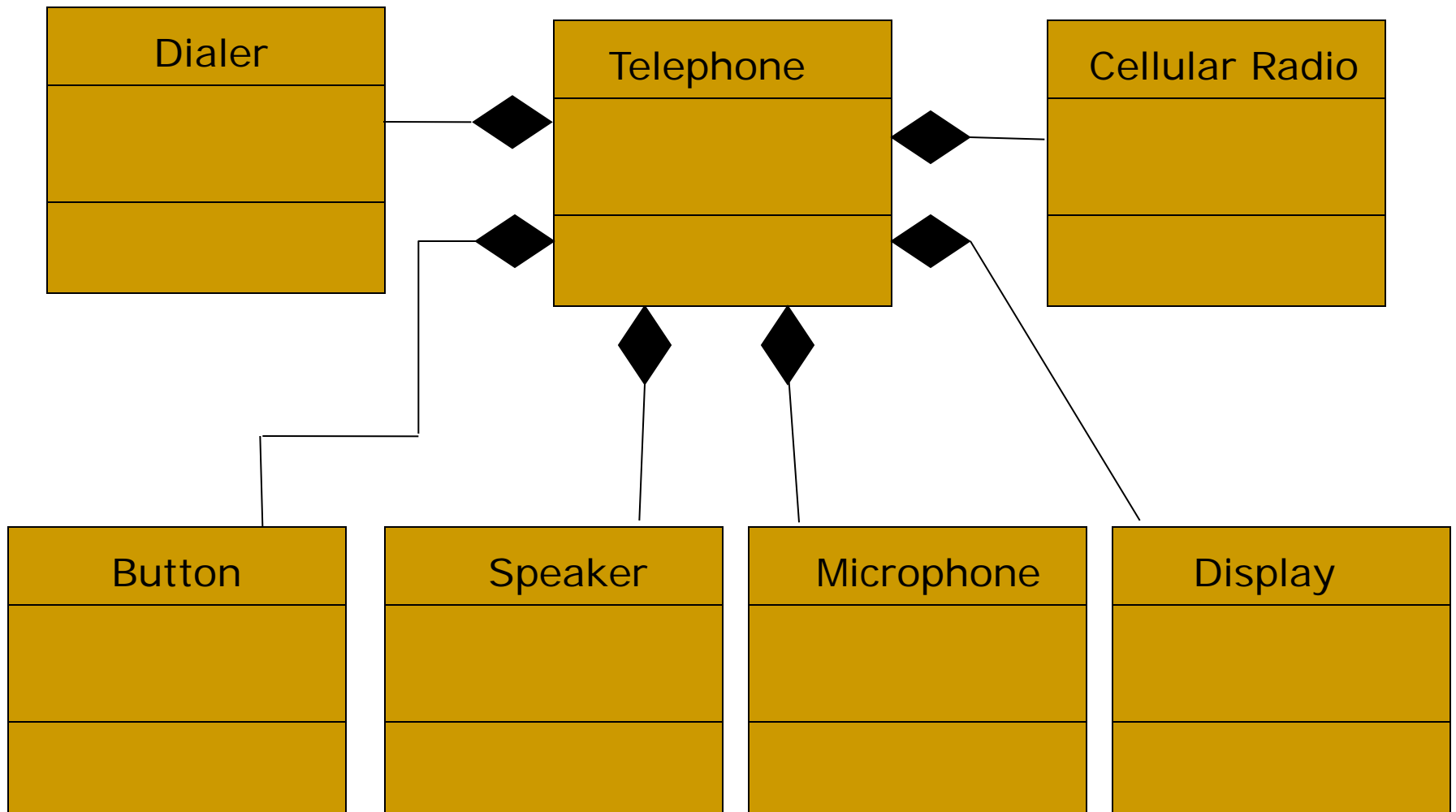
Sequence diagram example



Collaboration Diagram

- Show relationship between object messages passed between them
 - Objects as icons
 - Messages as arrows
 - Arrows labeled with sequence numbers to show order of events

Example: Cell phone class diagram

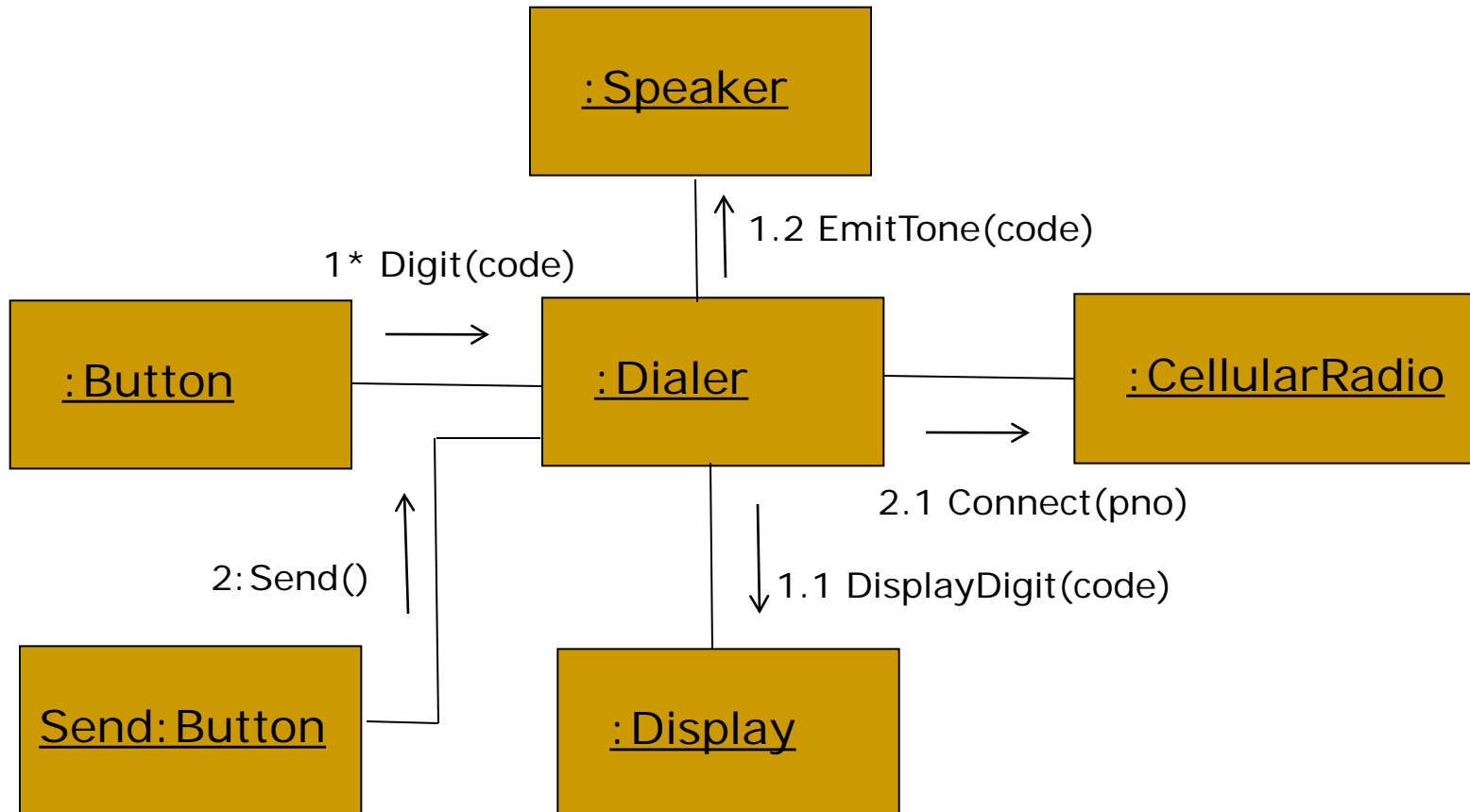


Source: Robert C. Martin, "UML Tutorial: Collaboration Diagrams"

Cell phone use case: Make call

1. User enters number (presses buttons)
2. Update display with digits
3. Dialer generates tones for digits – emit from speaker
4. User presses “send”
5. “In use” indicator lights on display
6. Cell radio connects to network
7. Digits sent to network
8. Connection made to called party

Collaboration diagram: Make call



Summary

- Object-oriented design helps us organize a design.
- UML is a transportable system design language.
 - Provides structural and behavioral description primitives.
- **Example: Model train set** (Section 1.4)