

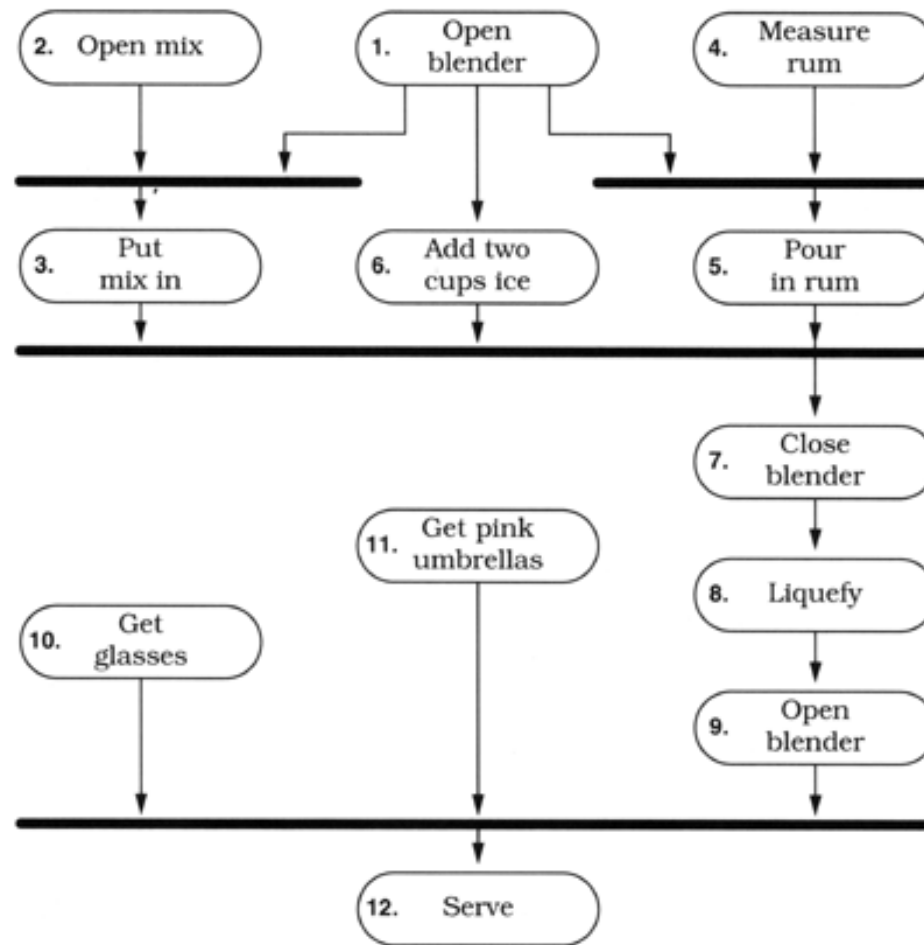
Formal Tools: introduction to UML

Elena Punskeya, elena.punskeya@eng.cam.ac.uk

Diagrams are Useful

- Can help you win a cocktail making contest!

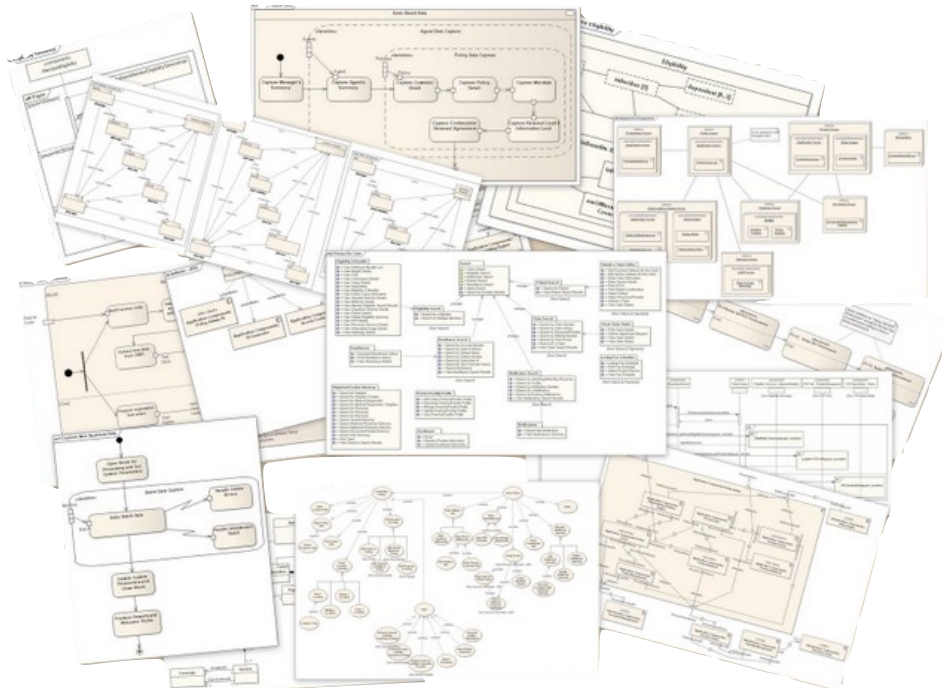
Figure 5.2. UML activity diagram: making a piña colada



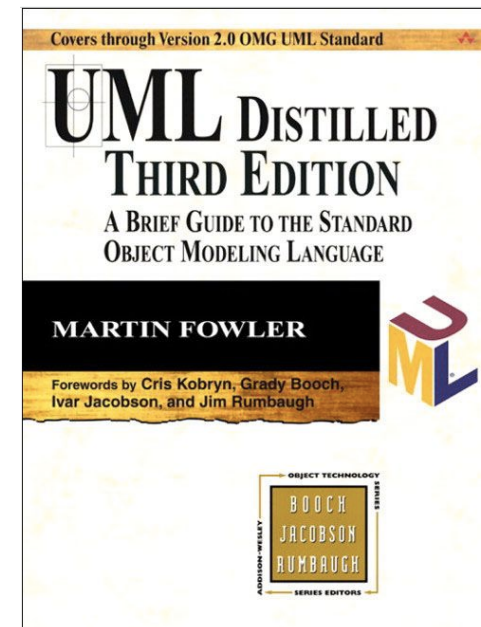
source: The Pragmatic Programmer, Andrew Hunt and David Thomas

Unified Modelling Language (UML)

- **UML** is a formal graphical language comprising a set of **diagrams for describing software systems**.
- It is used for designing, documenting and communicating various views of the software architecture and program behaviour.
- These different views of the system can be used at varying scales, presenting the key information and suppressing unimportant detail as desired.



source: Wikipedia



Brief History

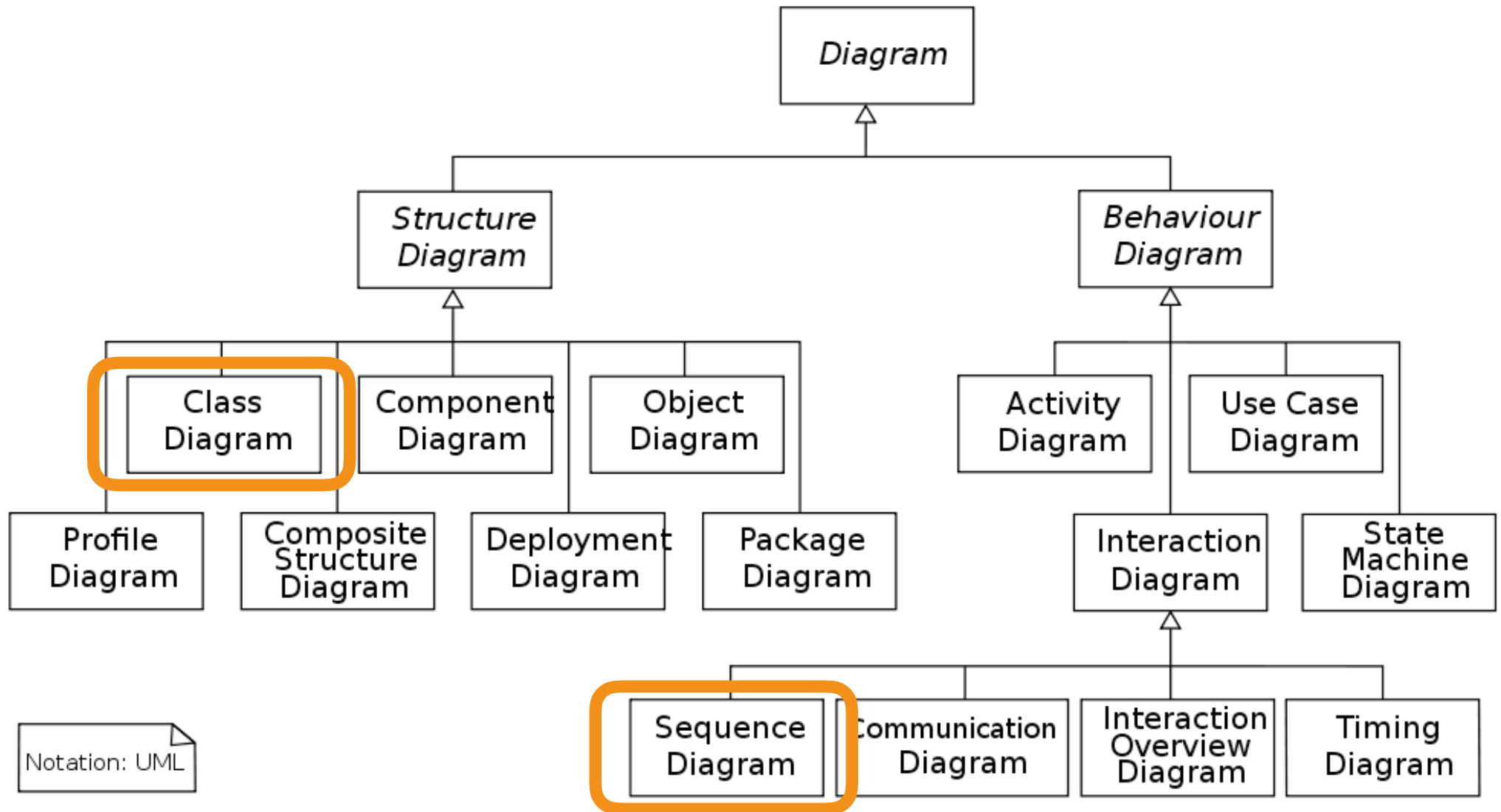
- UML evolved from **notations** used in three earlier rival approaches independently developed between 1989 and 1994.
 - **Booch method** developed by Grady Booch of the Rational Software Corporation.
 - **Object Oriented Software Engineering** (OOSE) developed by Ivar Jacobson of Objectory.
 - **Object Modelling Technique** (OMT) developed by James Rumbaugh of GE.
- All three of them ('three amigos') got involved in Rational Software – the company that produced a development framework (tools, processes and services)
- UML was developed as the notation for Rational Unified Process (RUP) and was released by the Object Management Group, a consortium that was setup to establish Object-Oriented standards and was founded by the likes of DEC, HP, IBM, Microsoft, Oracle, and Texas Instruments.
- Complete version of UML (V1.1) was released in 1997 and has been since managed by OMG
- Current version is 2.4.1 (released Aug 2011)
- "In Process" version 2.5 (released Oct 2012)

UML Diagrams and Modes

- To describe a system we need to communicate
 - **structures: items, and how they are connected**
 - **behaviour: how items interact**
- In UML, structures are described by **Class Diagrams**
- **Class diagrams** describe the software architecture of the system by specifying what classes are present in the system, and their relationships
- **Behaviour diagrams** show **interactions** between Objects, most commonly as Sequence and Communication Diagrams
- There are many more types of diagrams but these 3 types are the most practical ones
- Martin Fowler observes that UML can be used “AsSketch”, “AsBlueprint” and “AsProgrammingLanguage” with “**AsSketch**” being the most popular mode
 - <http://martinfowler.com/bliki/UmlMode.html>
- The key property of using “UmlAsSketch” is communicating parts of the system in a group of people

State of the Art

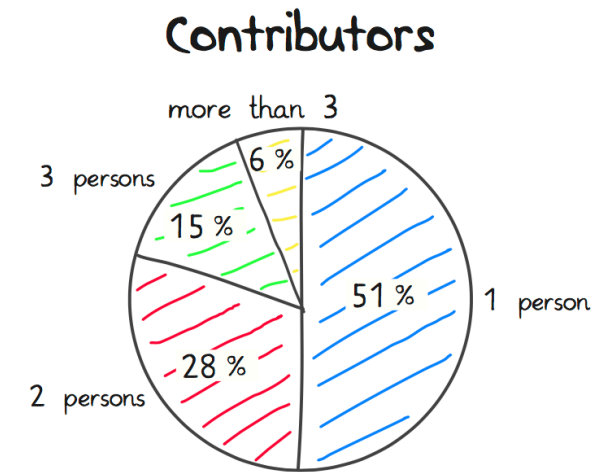
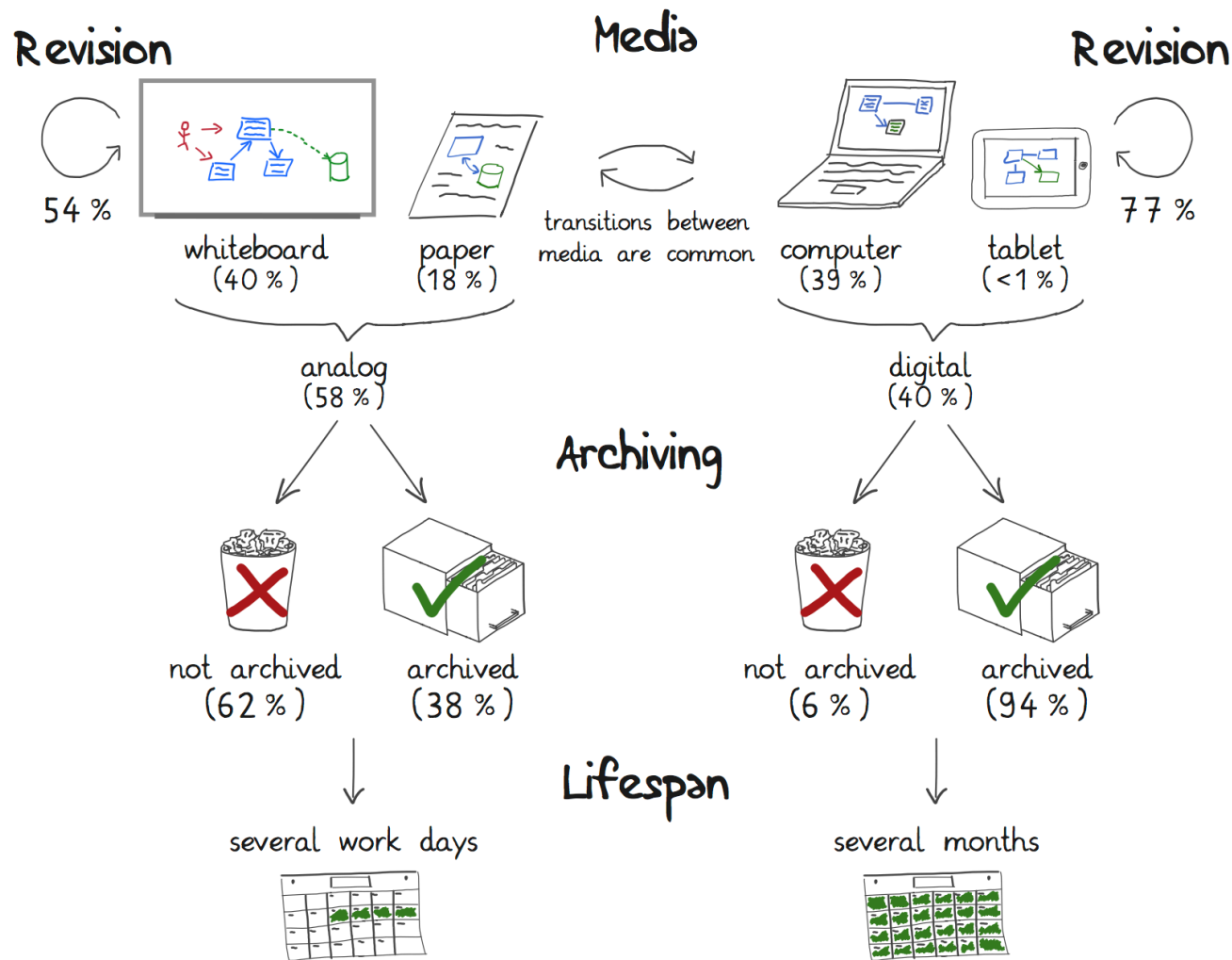
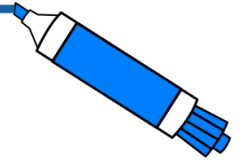
- 14 types are defined in UML 2.2



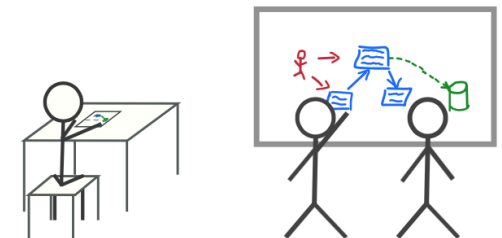
source: Wikipedia

Sketches and Diagrams in Practice

How and why do software practitioners use sketches and diagrams in their daily work?



Collaboration

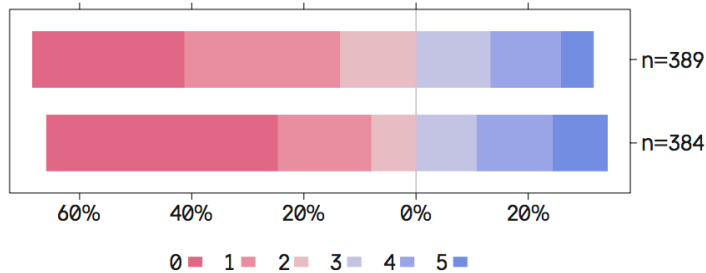


Paper is predominantly used alone, whiteboards collaboratively.

Formality and UML

Formality
0=very informal
5=very formal

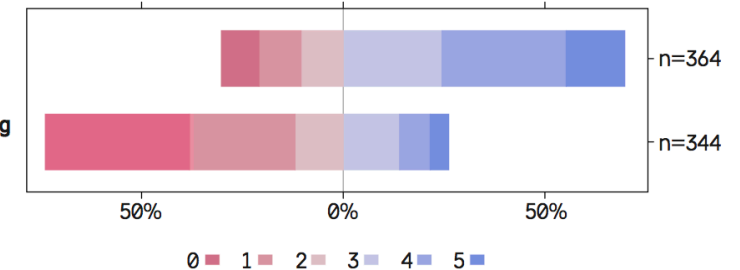
UML Elements
0=no UML elements
5=only UML elements



Agile Methods and MDE

Agile Methods
0=only agile methods
5=only other methods

Model-driven Engineering
0=never
5=always






Relation to Source Code






47 % of the sketches are rated as helpful for others to understand the related source code artifacts.

Purpose

Designing (75 %) 
Explaining (60 %) 
Understanding (56 %) 
Analyzing Requirements (45 %)

Team Size

11 % work alone 
8 % with one colleague 
19 % with two colleagues 
40 % with 3 to 9 colleagues
15 % with more than 10
5 % with more than 100



Sebastian Baltes
s.baltes@uni-trier.de





Stephan Diehl
diehl@uni-trier.de

Further information

Sketches and Diagrams in Practice
Sebastian Baltes and Stephan Diehl
FSE 2014

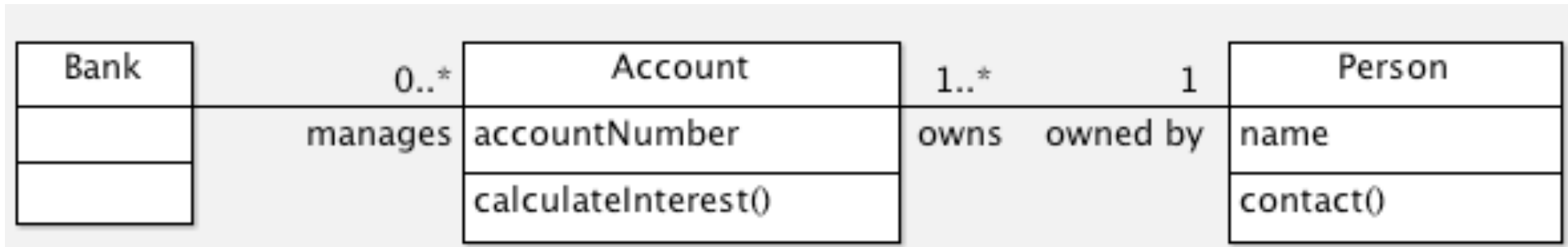
<http://st.uni-trier.de/survey-sketches>

Participants

- 394 software practitioners reported on their last sketch/diagram
- majority from Germany and USA
- 52 % software developers 
- 22 % software architects 
- median work experience 10 years
- various company sizes and sectors

Class Diagram

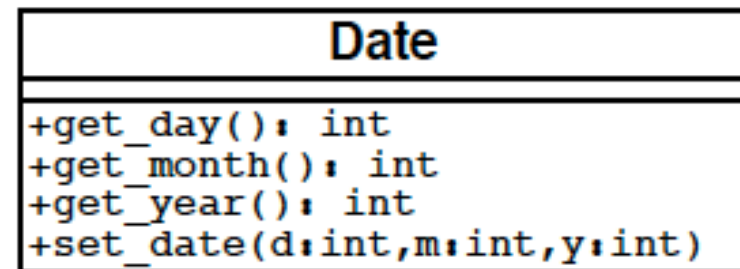
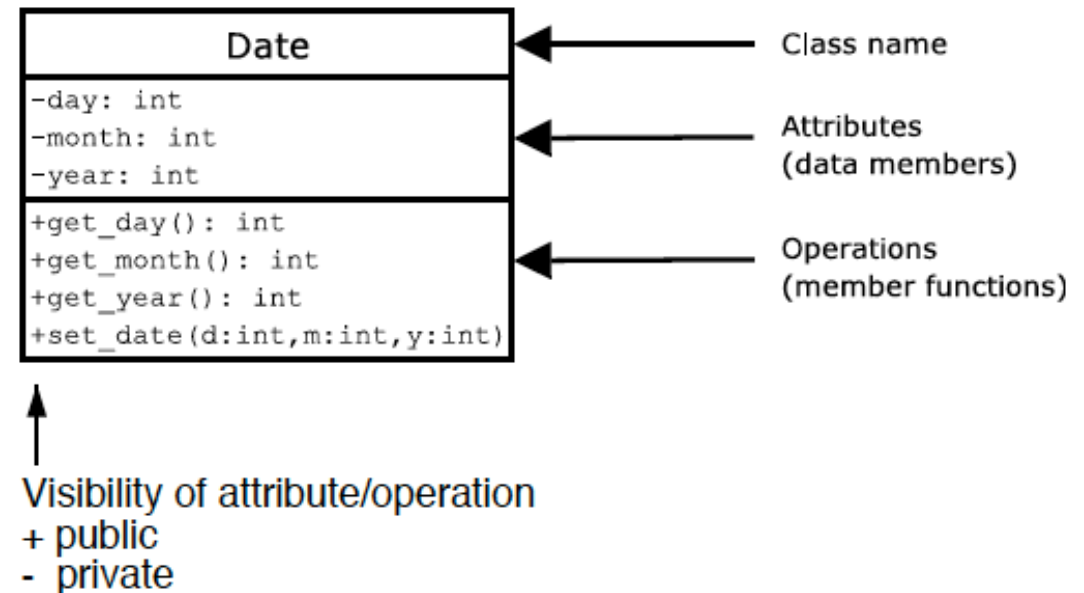
- These show the **classes** in a software system, their **attributes** and **operations** and their **relationships**



- This diagram says the following
- The system consists of a **Bank** that manages “0 or more” **Accounts**
- Each **Account** can be owned by 1 **Person** and each **Person** can own “1 or more” **Accounts**
- An **Account** has *account number* and can calculate *interest*
- A **Person** has a *name* and can be *contacted*

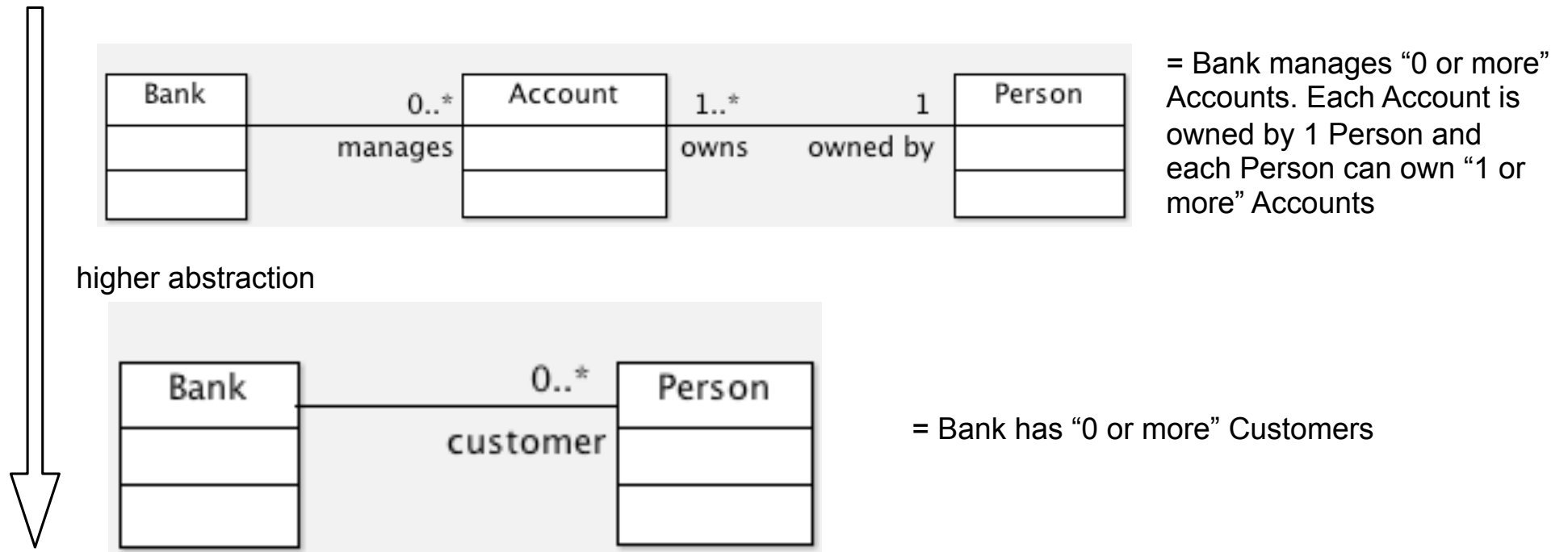
Class Diagrams

- Each class is represented by a box split into three sections
 - name
 - attributes
 - operations/methods
- The visibility of the attribute or method is noted by
 - “+” for public, i.e. accessible by any other object
 - “-” for private, i.e. internally used attributes and methods
- Mostly only public features of the class are useful to represent as they define interactions and relationships with other classes



Point of View

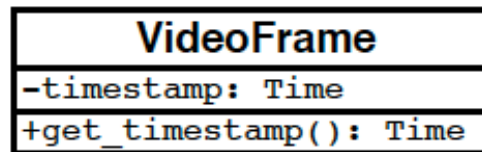
- In design and communication of ideas the right level of abstraction should be used in order to keep diagrams useful



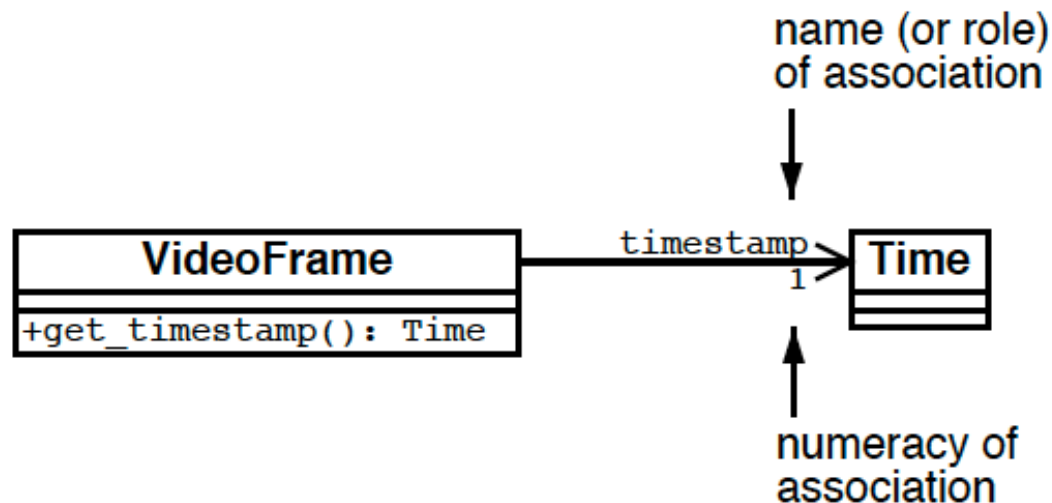
- It is common to drop the visibility indicator and assume that all shown attributes and methods are public
- Omitting any methods/attributes all together allows to focus on class relationships

Class Relationships

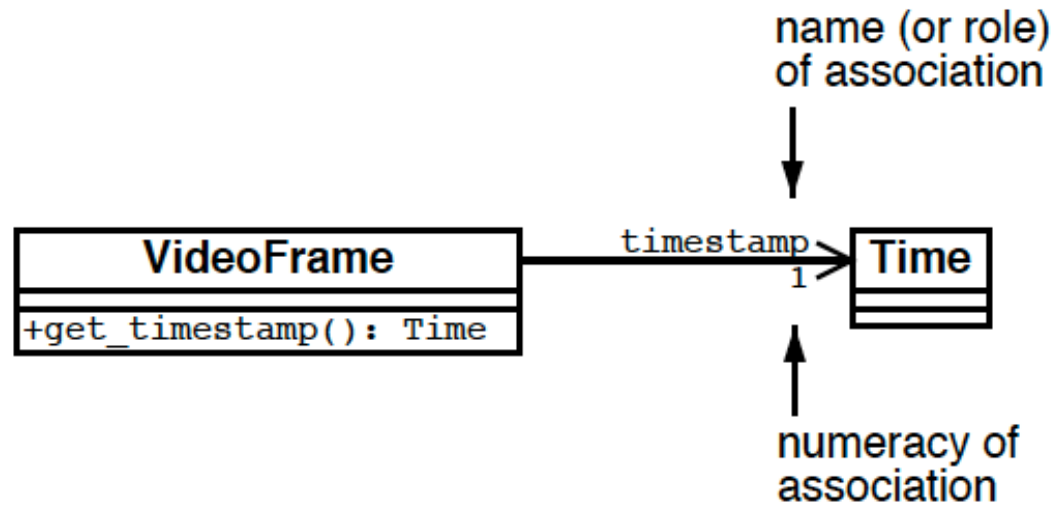
- Objects will often contain references (connections) to other objects. For example, a VideoFrame might contain a reference to a Time object rather than just an int to represent the timestamp
- This could be shown in a class definition



- However, more usefully, it maybe shown as an association



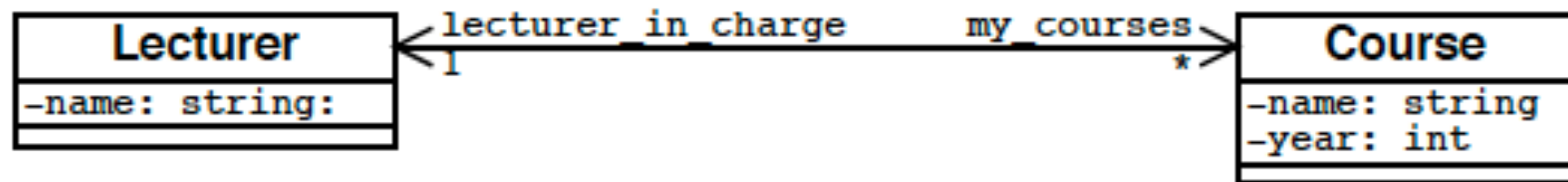
Association



- The line shows an association between **VideoFrame** and **Time**, which is
 - **directional**: a **VideoFrame** knows what its timestamp is but the **Time** object doesn't know which **VideoFrame** it linked to (indeed not all **Time** objects do belong to **VideoFrames**)
 - **meaningful**: a name for the role that the **Time** object plays within **VideoFrame**, in this case "timestamp"
 - **constrained**: a number defines the multiplicity/cardinality of the relationship, i.e. stating how many **Time** objects a **VideoFrame** can be in this relationship at the same time, in this case one. This can be a fixed number or a range, e.g. '0..1' or '*' or '1..*')

Navigability

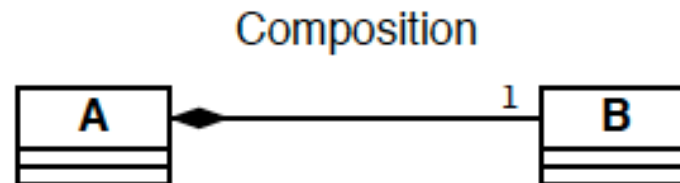
- Associations can also show bidirectional navigability. For example, consider the relationship between lecturers and lecture courses in a timetabling application



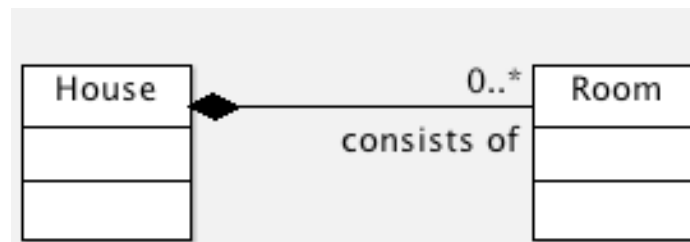
- This shows that each Lecturer object knows about all the courses they are teaching and that each Course knows which lecturer is in charge. The `*` indicates that a lecturer may be in charge of any number of courses (including none)
- The direction of associations is often not shown if it does not add to the meaning of the diagram, e.g. at the conceptual diagram abstraction level

Composition

- UML also allows a class association to be adorned with a diamond. Not all users of UML employ this, however, it is useful to know the meaning.
- A filled diamond denotes **Composition**



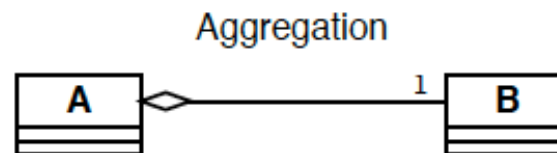
- If A is a House and B is a Room, then **Composition** relationship means that a Room cannot exist without a House, i.e. it will be destroyed if the House is destroyed



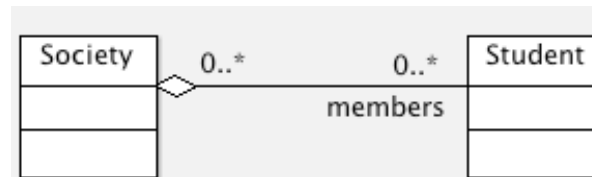
- Composition is potentially useful to emphasise the strength of the relationship between classes, e.g. an exclusive ownership

Aggregation

- An empty diamond denotes Aggregation and indicates that objects of class A contain objects of class B, i.e. B is a part of A
- A does not however control the lifecycle of object of class B



- A College Society can have a number of member Students, but it would be unreasonable to “destroy” students if the Society was to be closed

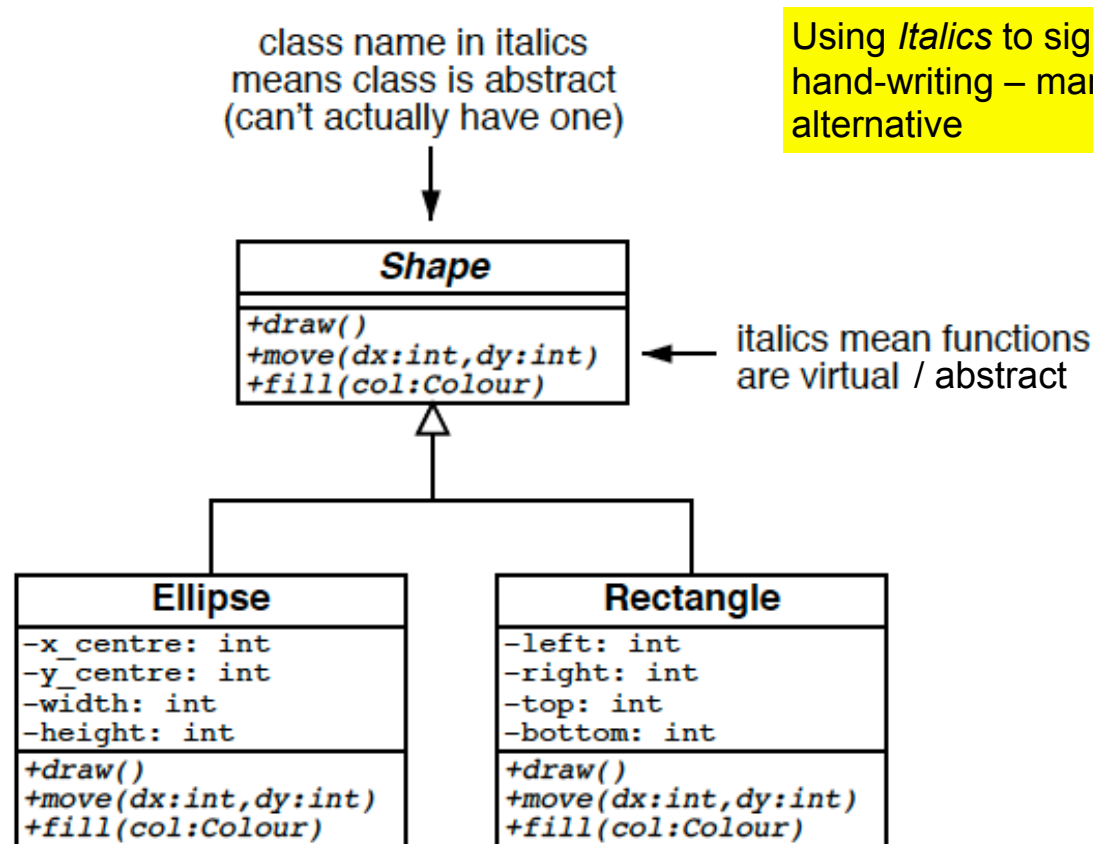


“Aggregation is strictly meaningless; as a result, I recommend that you ignore it in your own diagrams.”

UML Distilled , Martin Fowler

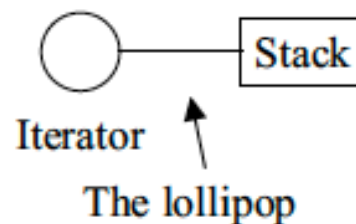
Inheritance

- As we discussed before, **Inheritance** and **Polymorphism** are the key features of Object Orientation that allow us to design extensible systems
- In UML, Inheritance relationship between classes is represented as follows

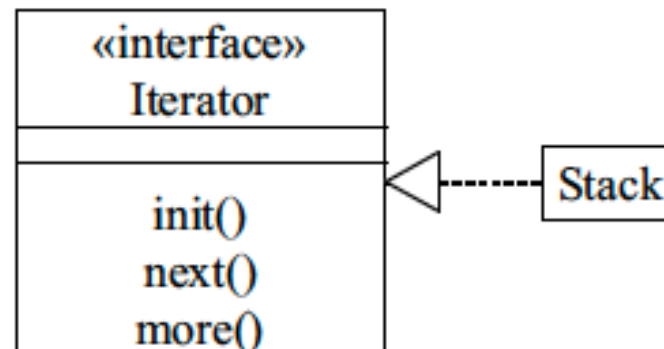


Interfaces

- There are two cases when Inheritance is useful
 - **Implementation inheritance** – the superclass implements some functionality that can be re-used by all of its subclasses
 - **Behaviour inheritance** when a class can expose a certain set of functionality
- Interfaces are used to define behavioural inheritance and denoted in UML as

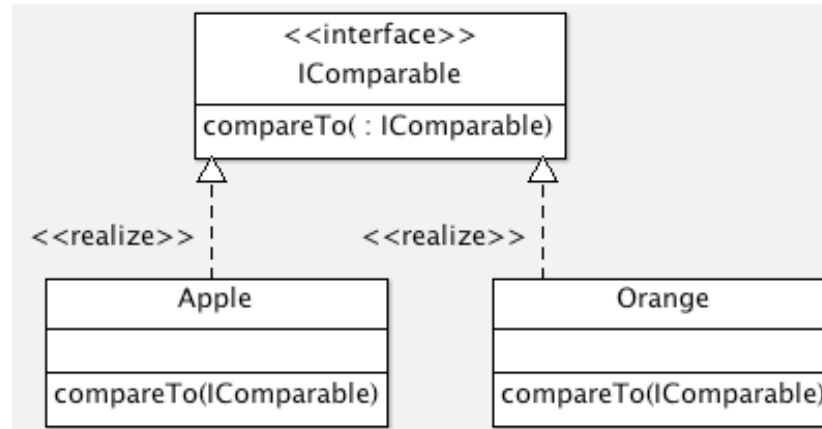


- or, more fully



Interfaces Example

- Defining the same functionality/ behaviour to very different concepts
- Example: can we compare Apples and Oranges?



- Yes, we can! – if Apples and Oranges “know” how to compare

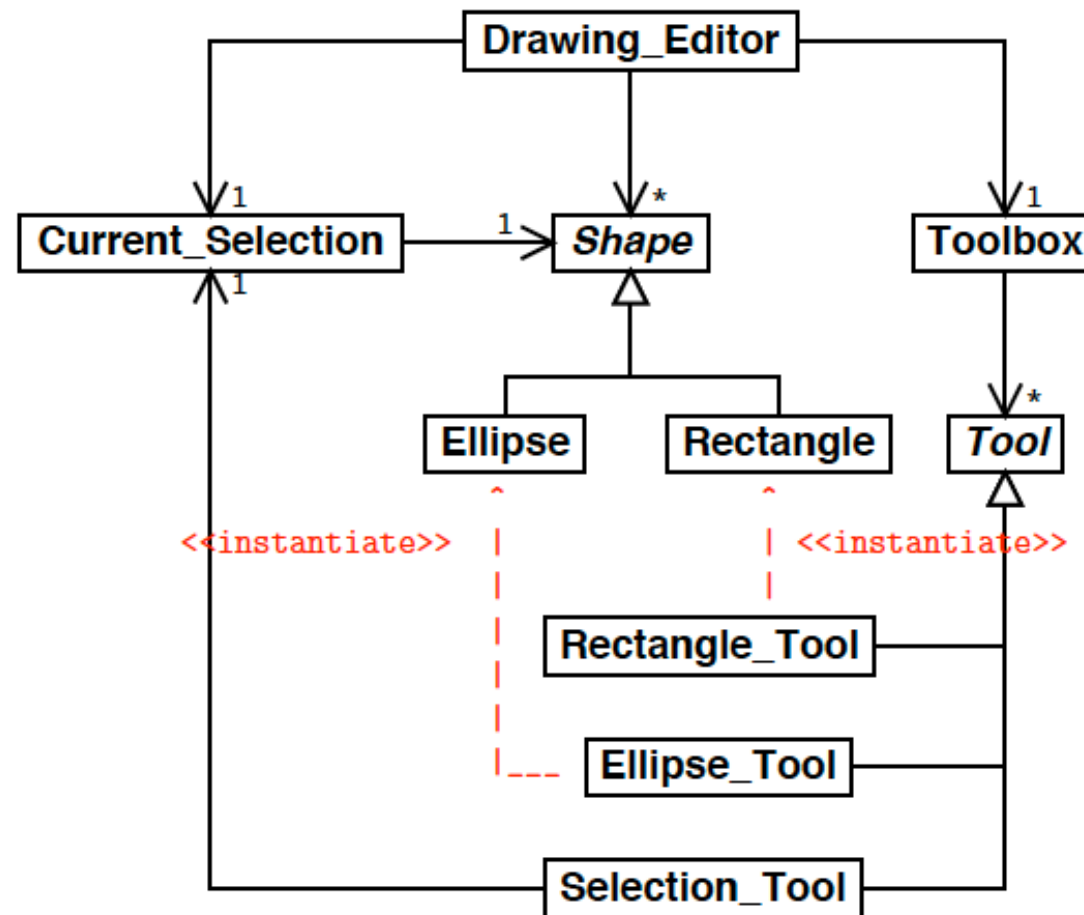
```
IComparable apple = new Apple();
IComparable orange = new Orange();

apple.compareTo(orange);
```

- Of course, the tricky part is how the comparison is made :)

Class Diagram Example

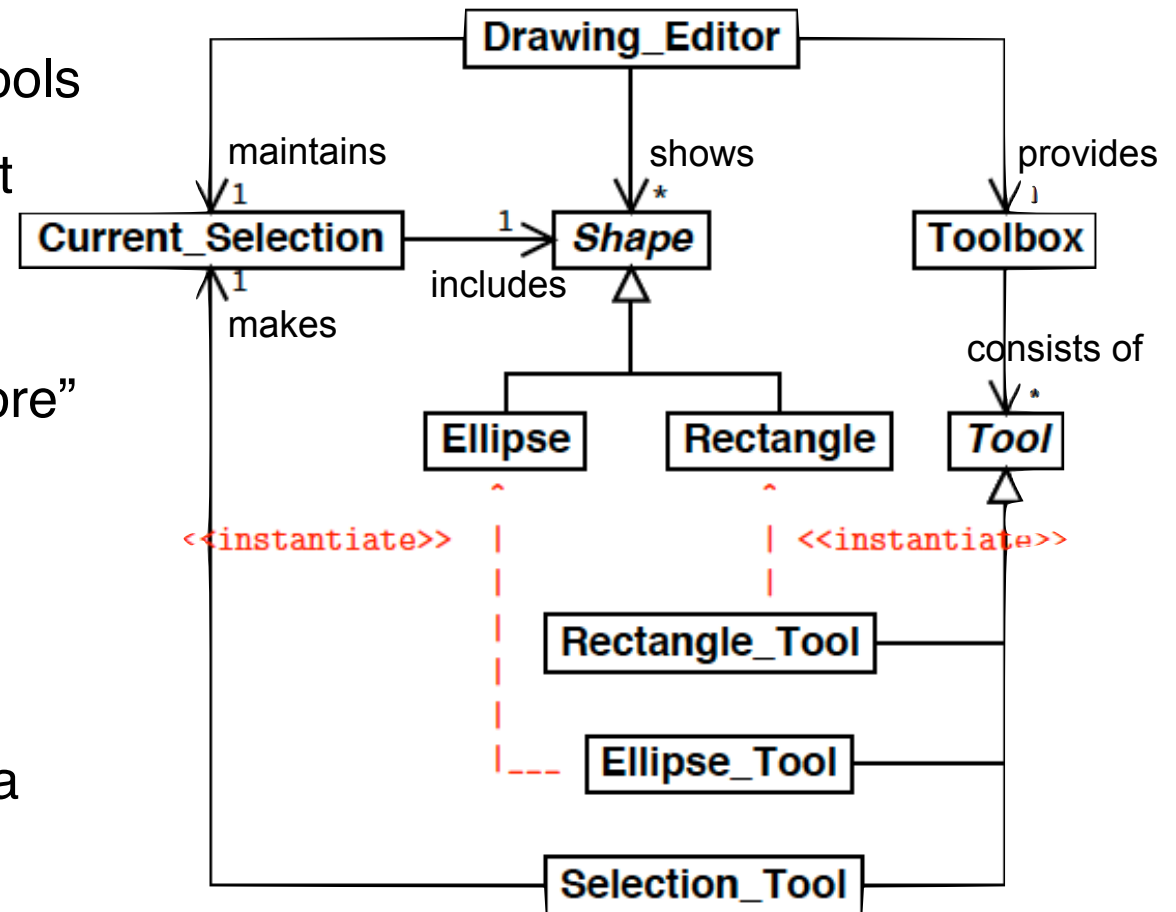
- Drawing Editor can be described as the following



- What can we understand from reading this diagram?

Class Diagram Example

- Drawing Editor provides 1 (and only 1) Toolbox
- Toolbox consists of “0 or more” Tools
- There are three types of tools that are used to create Rectangles, Ellipses and make a Selection
- Drawing editor can show “0 or more” Shapes. There are two types of shapes: Ellipse and Rectangle
- Drawing Editor maintains the Current Selection
- Only 1 shape can be selected at a time. Selection Tool makes the Current Selection which includes Shape

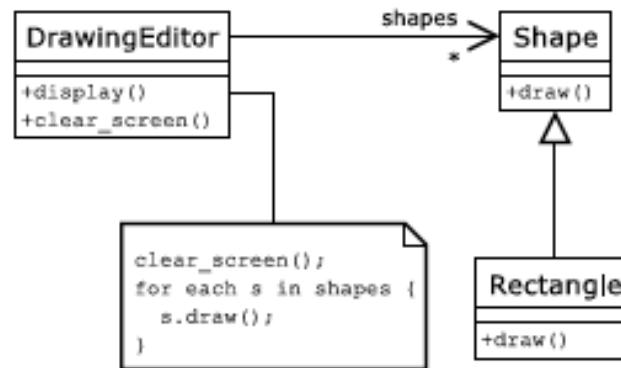


Dynamic View

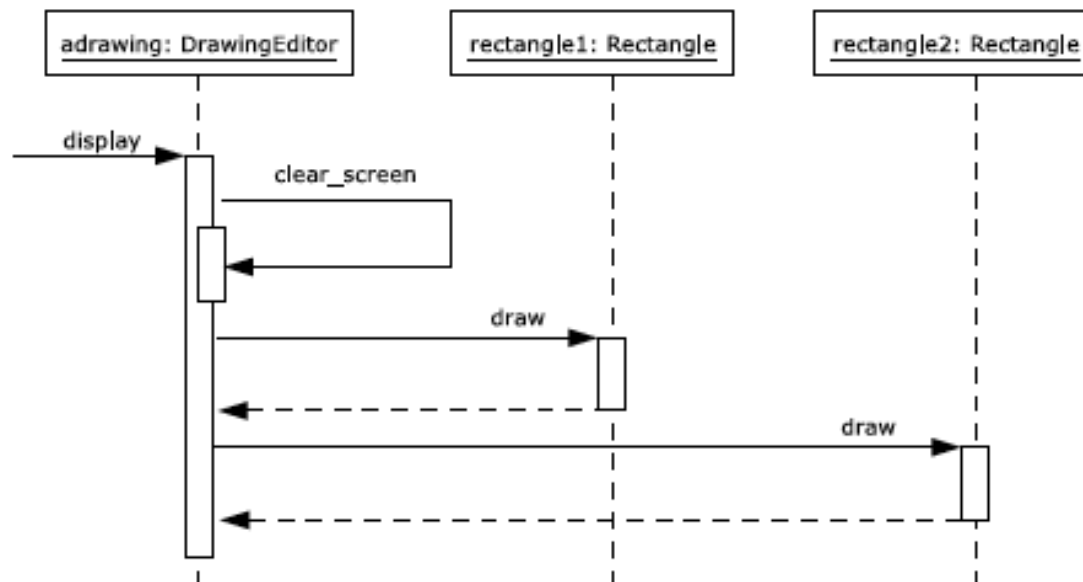
- Class diagrams are very useful to capture the **relationships**, but this is only one part of a story – this is only a list of ingredients from a recipe, but to make a dish we need to know how to put them together
- To understand how the system works (and validate the design) we need to understand how **objects interact with each other** for a purpose of a particular action
- There are two main types of UML diagrams that capture interactions
 - **Sequence** Diagrams
 - **Collaboration**/Communication Diagrams
- Both could be used for exactly same purposes
- **Sequence** Diagrams present a more clear view of the timeline, therefore could be particularly helpful to capture the logic flow
- **Communication** Diagrams can focus more on the nature of collaboration between the object to perform an action

Sequence Diagram

- Consider the drawing Editor example



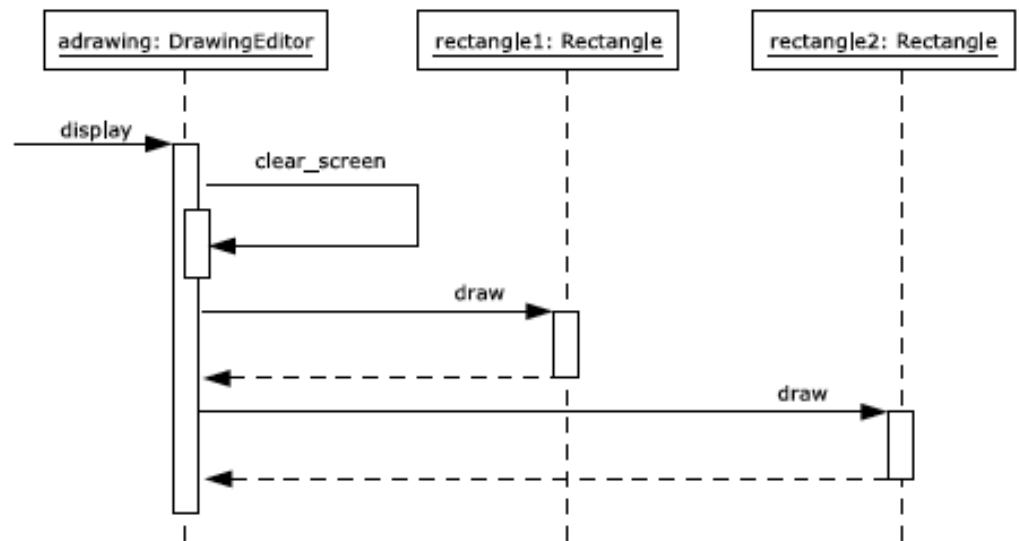
- Suppose the drawing consists of two rectangles and the *display()* method is called on the DrawingEditor. The series of calls that take place can be shown in a sequence diagram:



Sequence Diagram

- This diagram shows three objects. These are: one object of type Drawing Editor called **adrawing** and two objects of type Rectangle called **rectangle1** and **rectangle2**. The vertical axis of this diagram corresponds to time (traveling downward). The white boxes show the duration of each call. This diagram explicitly shows the return from the two draw() calls with dashed lines although these are often omitted

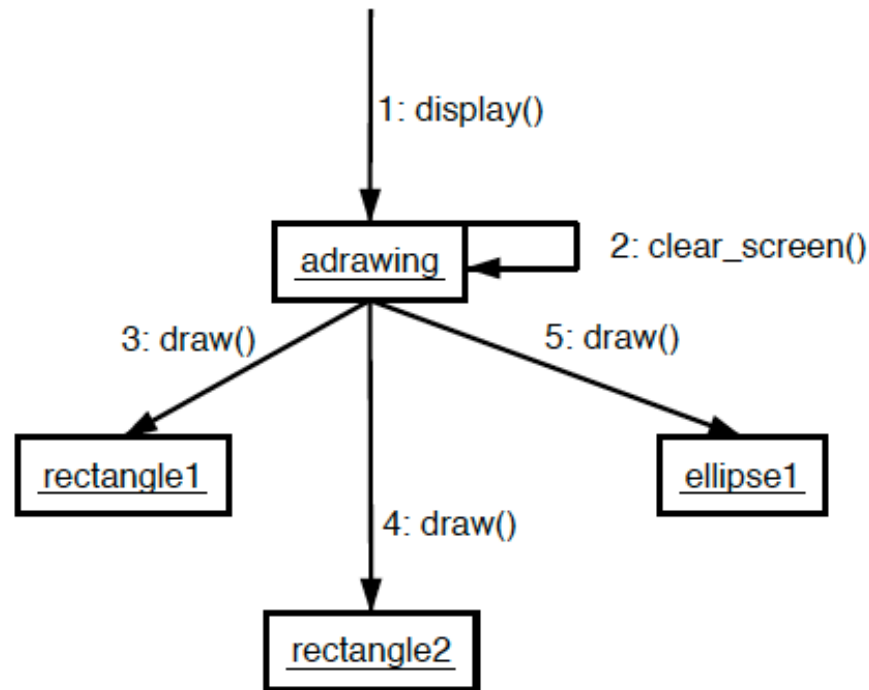
1. display() is called in adrawing
2. adrawing calls the clear screen() method in itself
3. This method returns
4. adrawing calls the draw() method in rectangle1.
5. This method returns.
6. adrawing calls the draw() method in rectangle2
7. This method returns.
8. The display() method is completed and returns



- UML defines additional notation (e.g. object deletion/creation) – rarely used

Communication Diagram

- **Communication** diagrams show the method flow between objects by numbering method calls

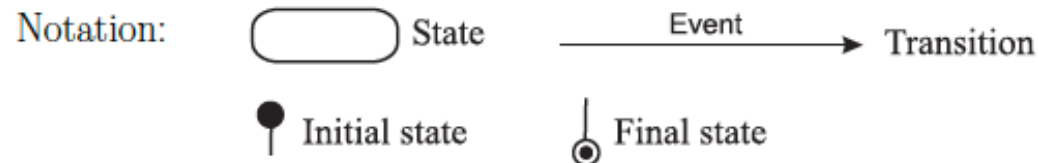
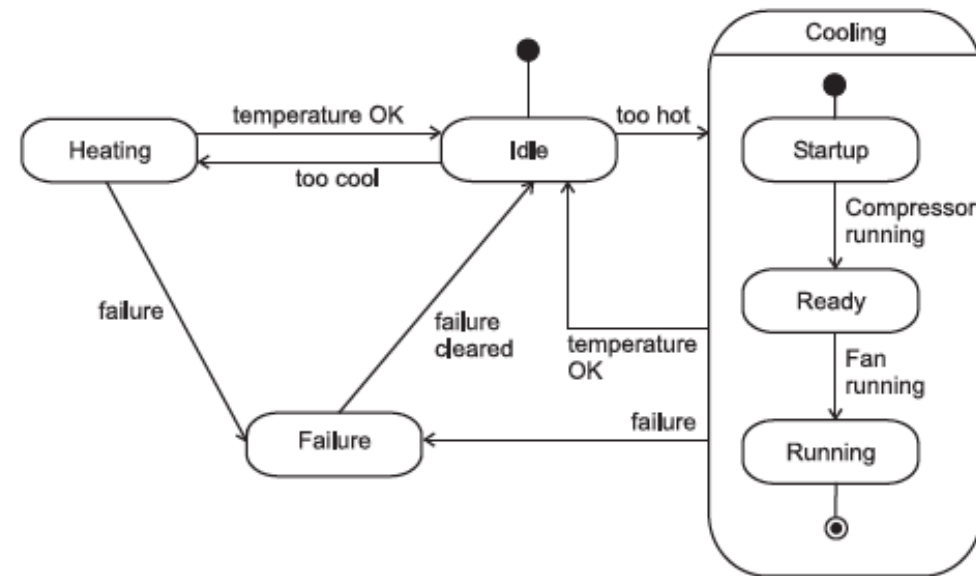


- Both diagram types capture the dynamic behaviour of the system in a specific scenario
 - Sequence diagrams are easier to read when there are a few objects with potentially a lot of calls
 - Communication diagrams are easier to read when there are a few method calls between potentially many objects

State Diagrams

- Sometimes the behaviour of a single object depends on its state, UML offers **State** Diagrams to capture it

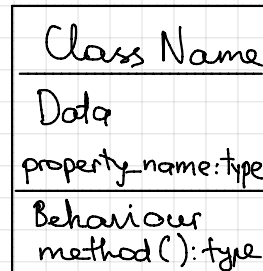
Example: Air conditioning unit



- The need for such diagrams is more common in e.g. Embedded Software development

Quick UML Summary

Class

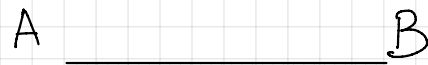


Abstract

{ Class Name }
or
Italics

Interface

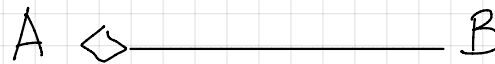
<< interface >>
I Adjective



Association (A and B call each other)



Uni-directional association
(A can call B but not vice versa)



Aggregation



Composition



Interface: A implements B



Inheritance: A inherits from B

UML Summary

- UML is a notation that helps to capture, share and validate design concepts
- It can be used in conjunction with formal software processes (e.g. RUP), where even source code can be generated from UML diagrams
- The main strength is however in the ability to describe the static and dynamic view of the system in the common way
 - **Class** and **Sequence** (Communication) diagrams are by far the most used parts of UML
- Using UML as a basis, project teams can adapt and extend it to their need, but creating new types of diagrams, modifying the notation etc. is OK, as long as:
 - everyone on the project can understand it
 - everyone's understanding is the same ;)

