CS 4350: Fundamentals of Software Engineering
Lesson 2.4 The Object Scale

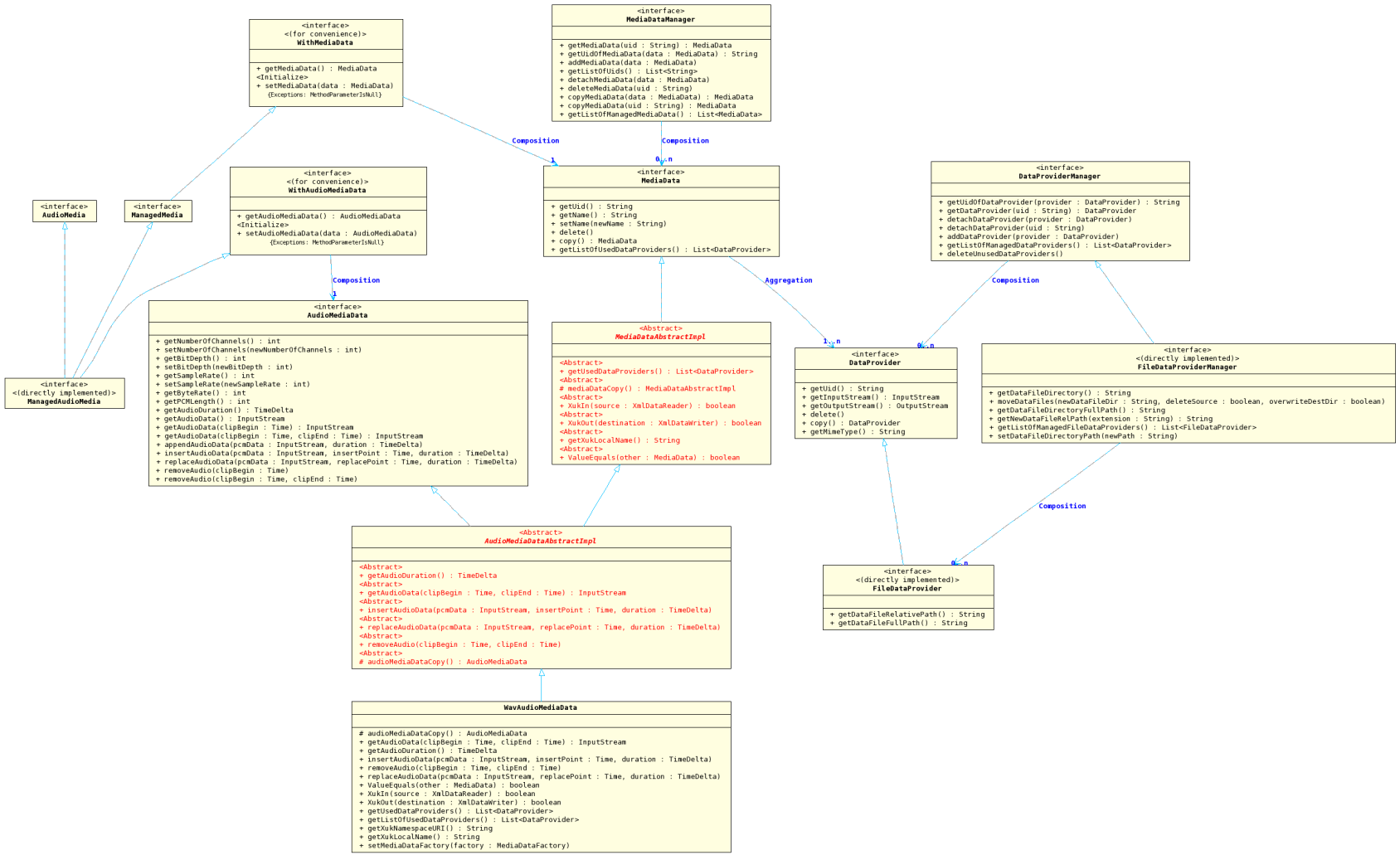
Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

- At the end of this lesson, you should be able to
  - Demonstrate the basics of UML class diagrams
  - Explain the significance of the different kinds of associations in UML
  - Explain what the reviewer of a design might want to know that UML leaves out

This is the scale of UML diagrams

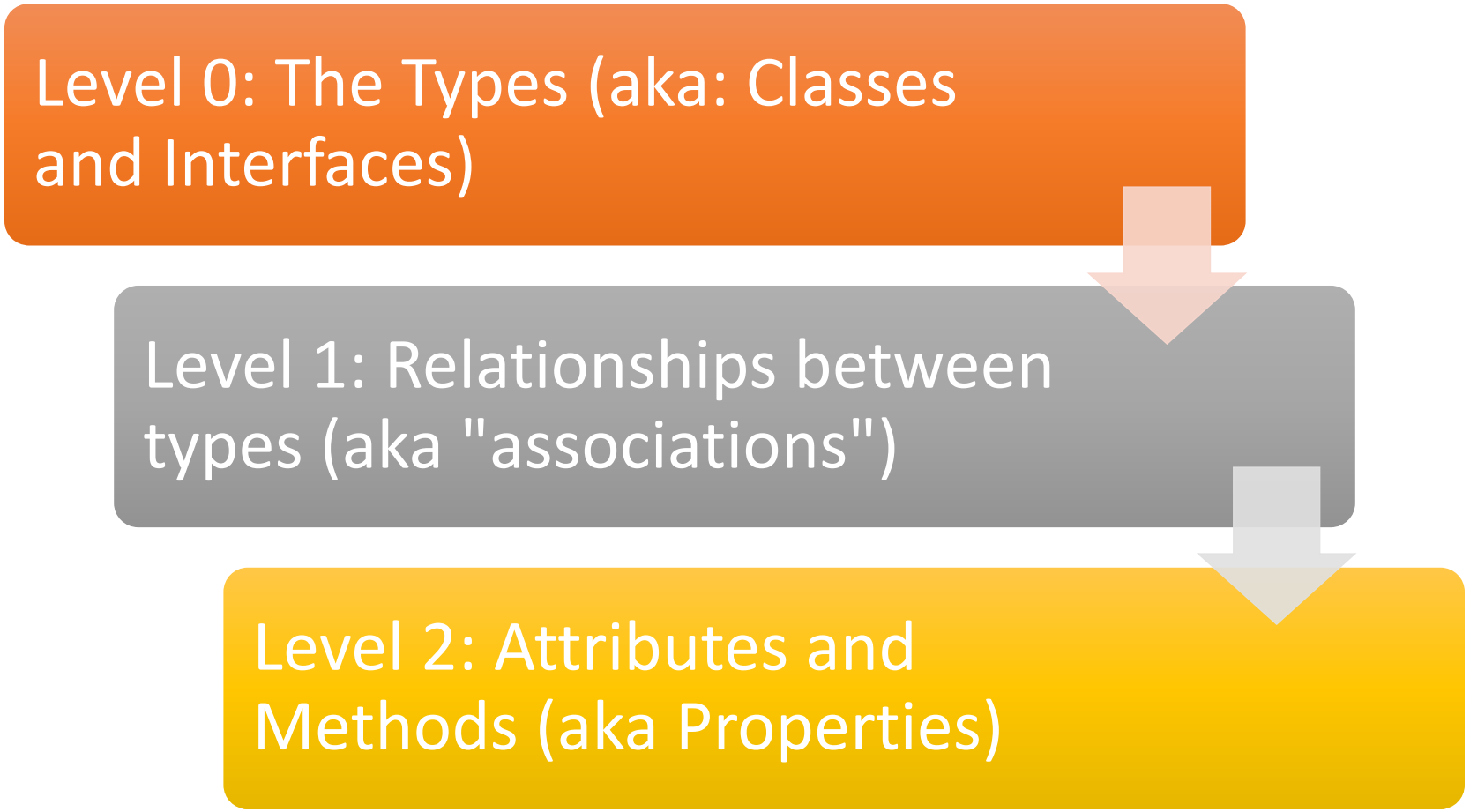


UML in the context of this course

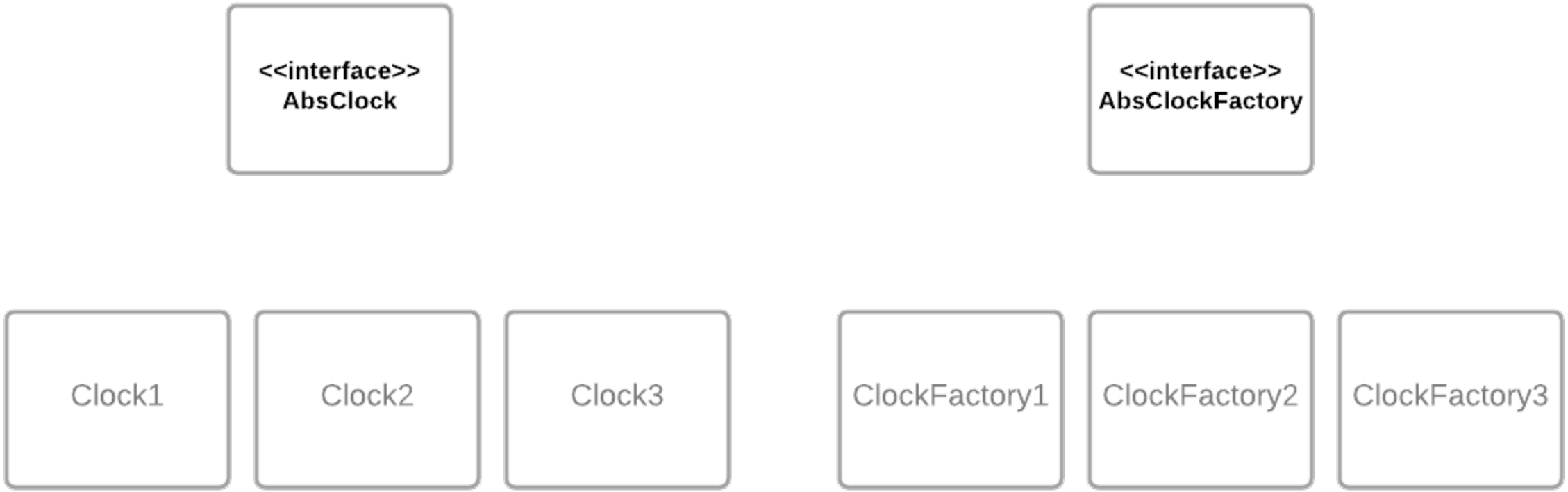
- There are numerous tools for translating from UML to code (or code fragments), and vice versa, BUT
- We are interested in UML as a human-to-human language.
- In general, we expect your UML diagrams to "look like" UML diagrams, but we are not interested in every last detail of the notation.
- We just want your diagrams to communicate the important things, with detail as necessary.

It will not be satisfactory to simply rely on some UML-generation tool. That will only demonstrate that you haven't thought hard about the problem ☺

3 Levels of UML



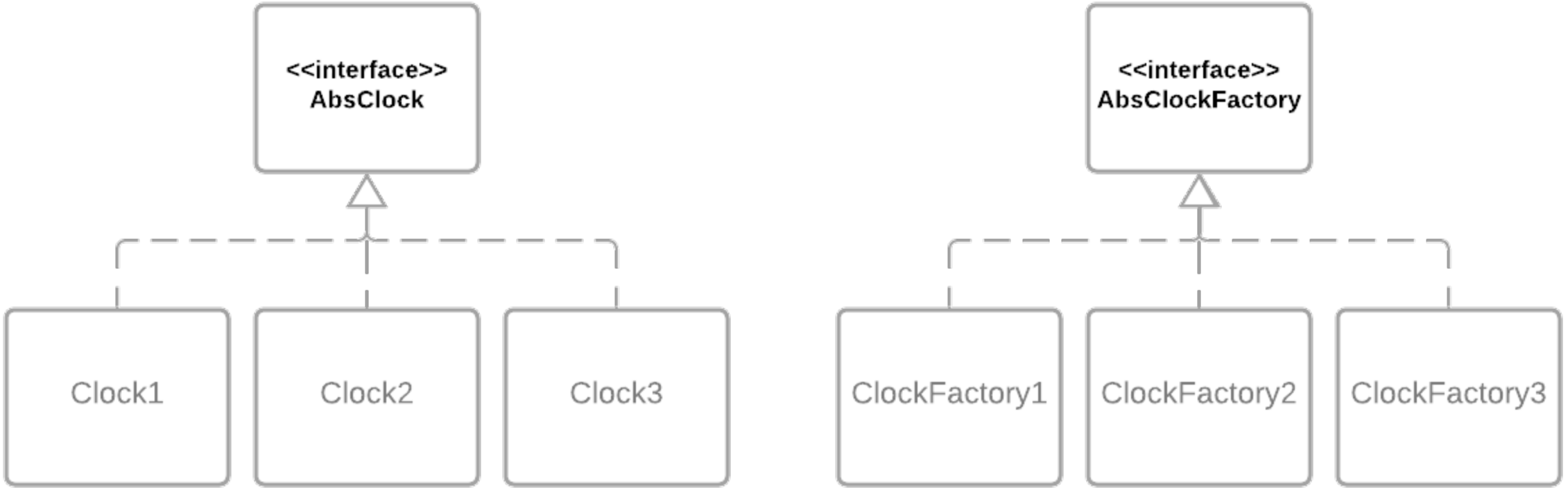
Level 0: Types (Interfaces and Classes)



Level 1: Relationships

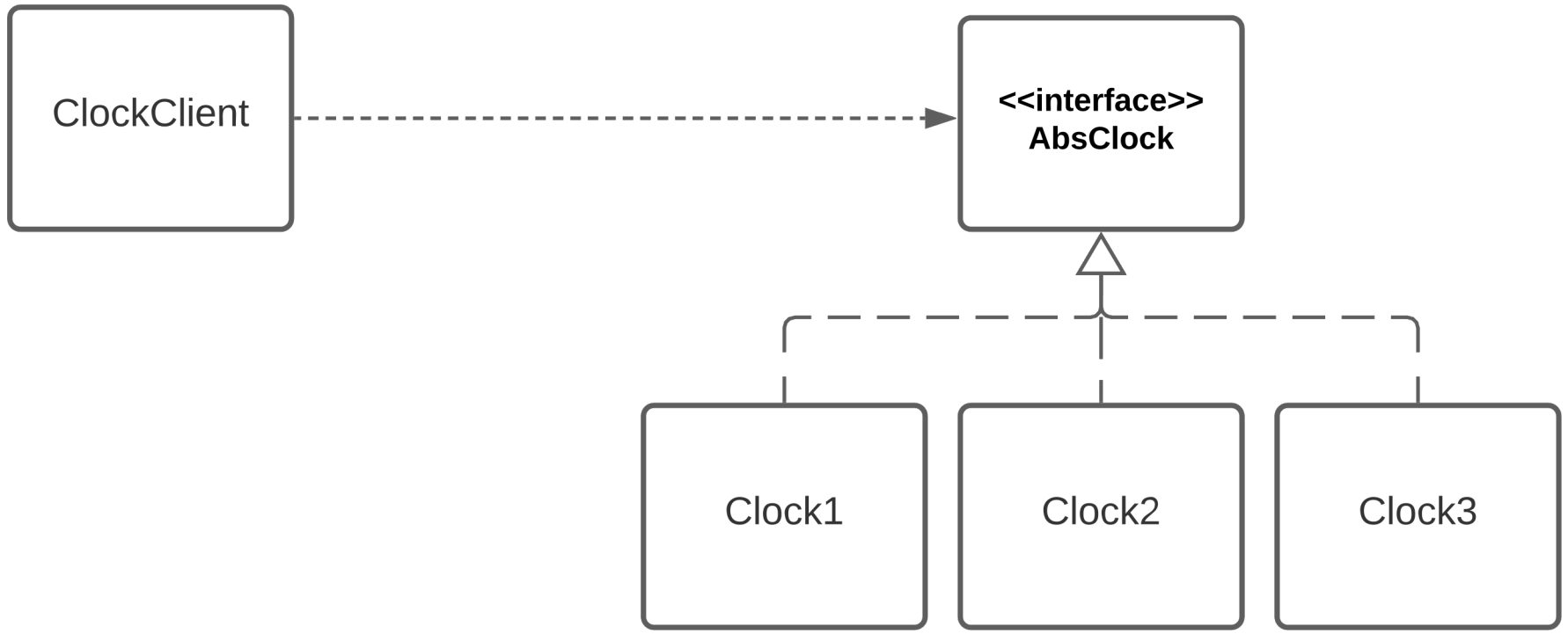
- Some different kinds of relationships
  - *implements*
    - *SimpleClock implements AbsClock*
  - *depends-on (or refers-to)*
    - *ClockClient depends-on AbsClock*
  - *subclass-of (or inherits-from)*
    - *(use alternate impl of ClockFactory)*
  - *Associations*

"Implements" relationship

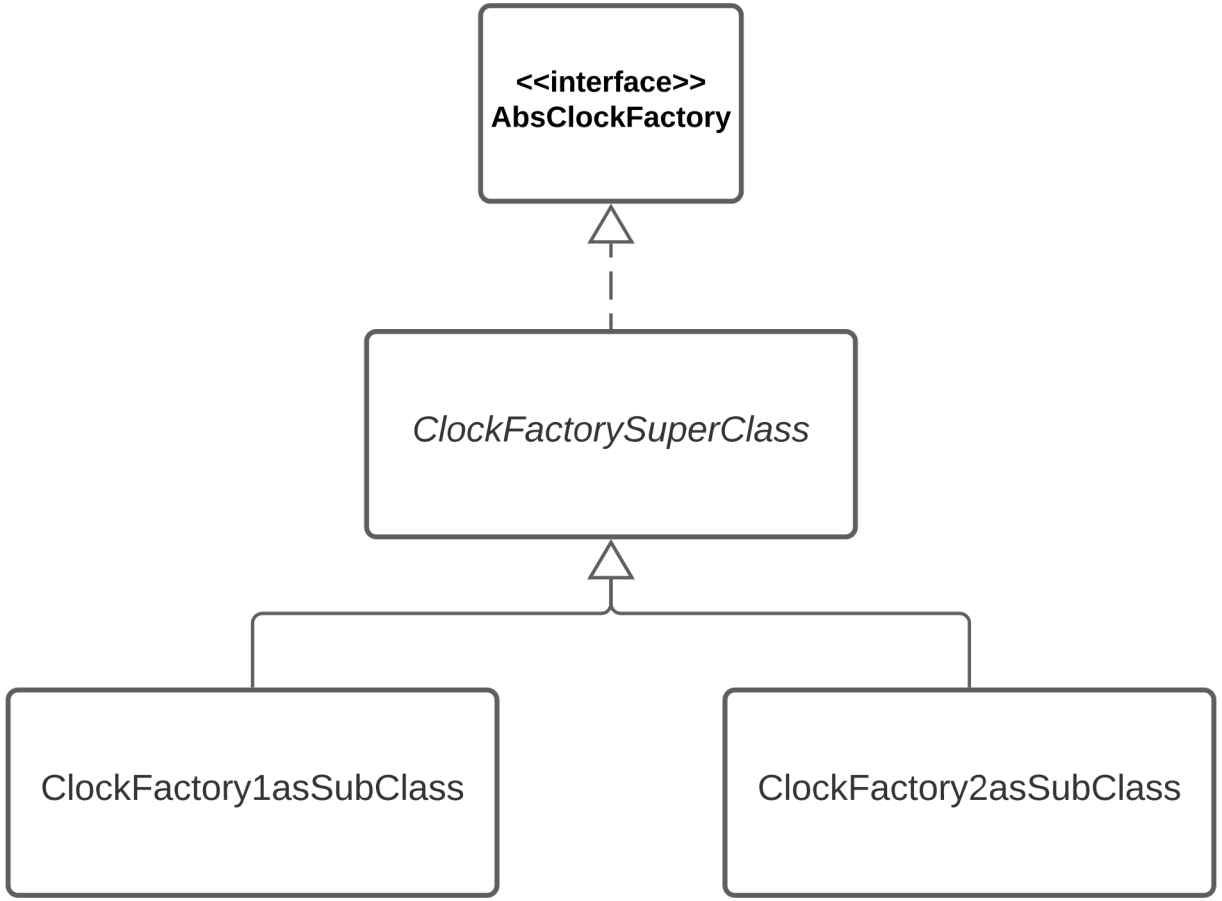


"Depends on" relationship

```
export class ClockClient {
    constructor (private theclock:AbsClock) {}
    getTimeFromClock ():number {return this.theclock.getTime()}
}
```



"inherits-from" relationship in UML



Associations

- An association is a relationship between two objects that indicates a link or dependency between them.
- Examples:
  - a portfolio is associated with an investor
  - every sale is associated with the sales representatives that worked on the sale
  - every student is associated with a transcript
- An associations typically has a name that indicate its meaning in the real world
- An association typically has a cardinality that indicates whether it is a 1:1 relation, a 1:many relation, etc.

Associations

Each teacher teaches zero or more courses.

Each course has exactly one instructor

Each student is enrolled in zero or more courses

Each course has zero or more students enrolled

Associations in Code


```
class Teacher {
    coursesTaught : Course[]
}
```

```
class Course {
    instructor : Teacher
    roster : Student[]
}
```

```
class Student {
    classesTaking : Course[]
}
```

```
// INVARIANT:
// c.instructor = t iff c is in t.coursesTaught
// s in c.roster iff c is in s.classes
```

Properties of Associations: Cardinality (or Multiplicity)

- The relationship between two entities has an associated cardinality or multiplicity
  - multiplicity is expressed with specific numbers or ranges,
  - e.g.: 1:1..2 or 1:1..N

- Examples:
  - A student is associated with exactly one transcript (1:1)
    - One student, one transcript.
  - Every course is taught by a professor, but a professor must teach at least one course (1:1..*)
    - One course, one professor. One professor, one or more courses.
  - An address may have a zip code (1:0..1)
    - One address, zero or one zip code

Notation for Cardinality in Associations

Any given instructor teaches 1 course.
Any given course is associated with one instructor.

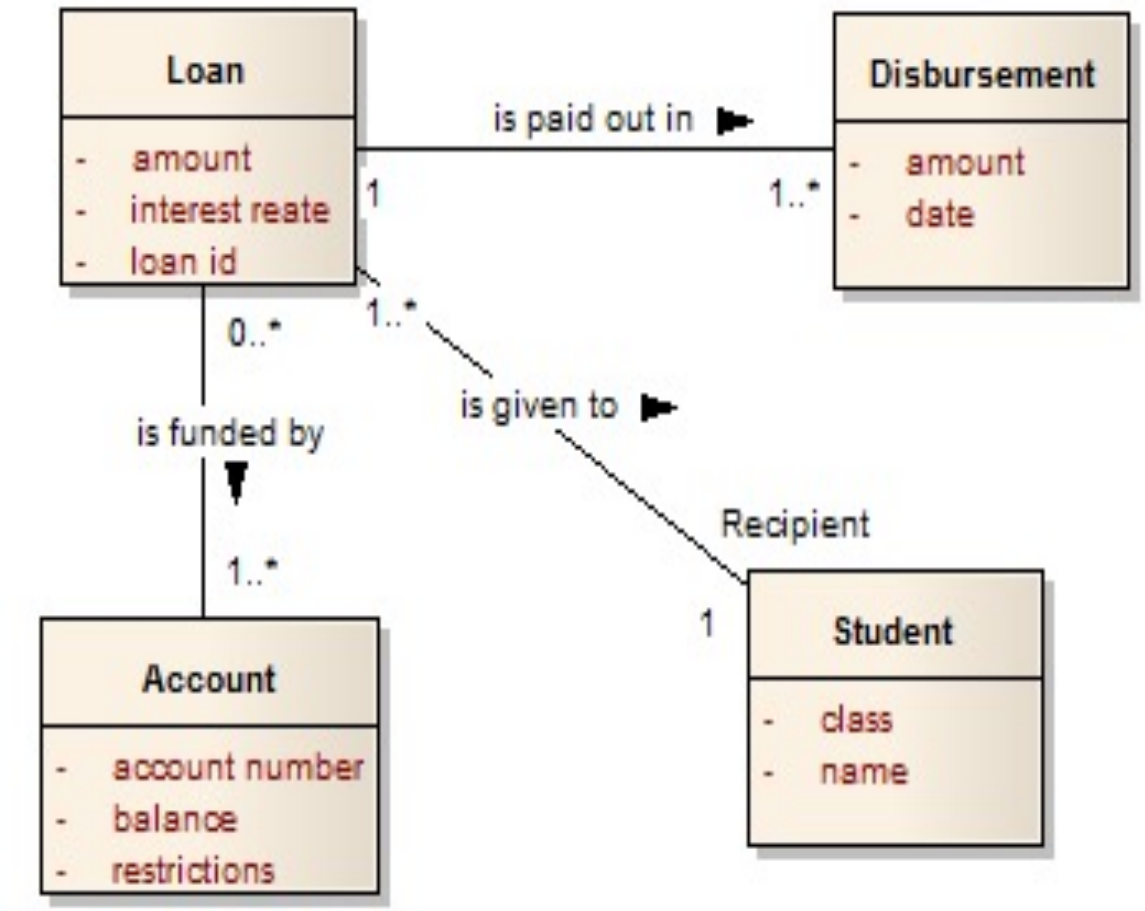
Any given instructor teaches at least 1 and up to 10 courses.
Any given course is associated with one instructor.

Any given instructor teaches 1 or more courses.
Any given course is associated with one instructor.

If no cardinality is

Note: the solid triangle indicates how a human should interpret the relationship ("Instructor teaches Course"). It does not indicate navigability (from an instructor, can you find the list of courses they teach?)

Associations should reflect something about the real world

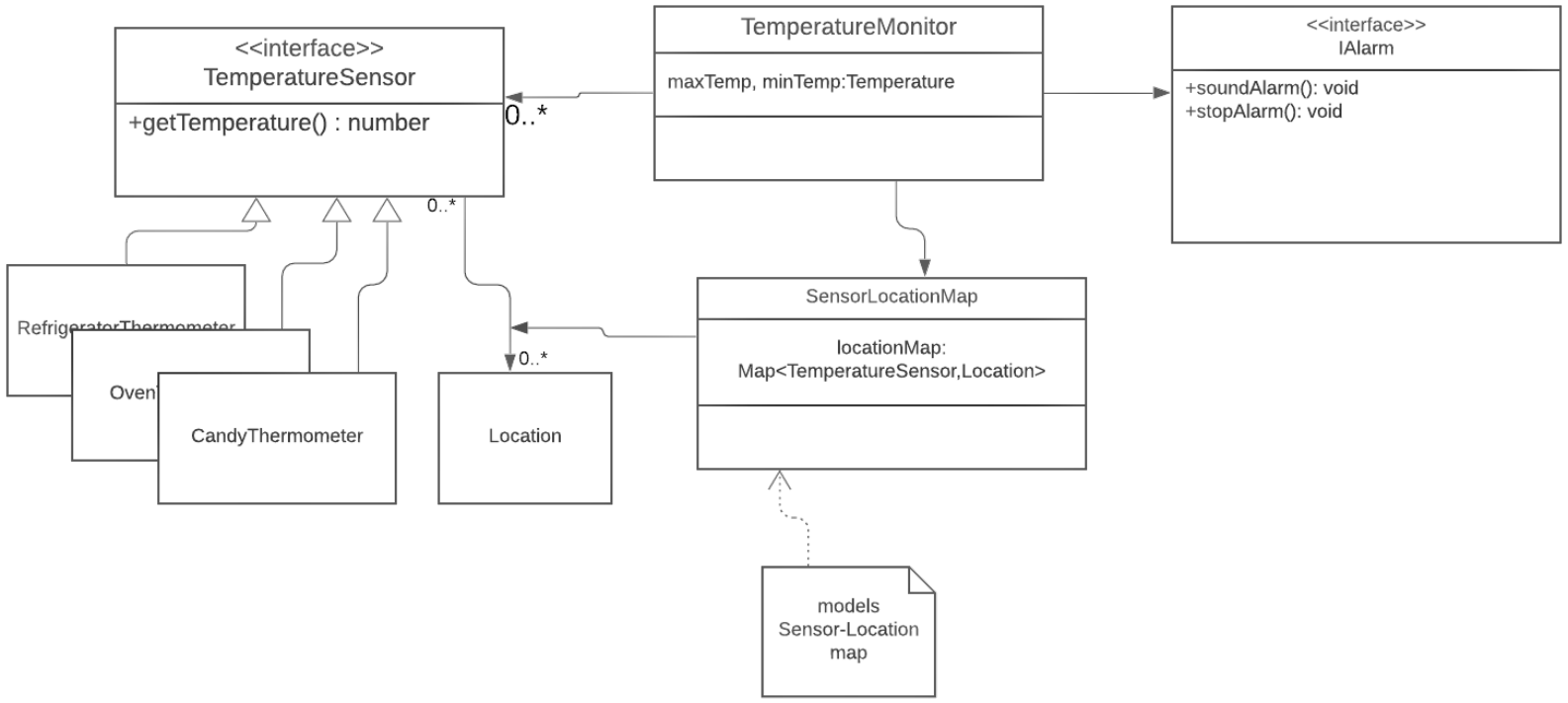


Partial Translation:

We have discovered that a loan can be paid out in multiple disbursements. There does not appear to be any limit to the number of disbursements. In addition, each loan is given to a single student. Apparently, students cannot share loans.

What world are we modeling?

- Sometimes the world we are modeling is not the real world, but the world of entities in our program



Discussion Question: Which parts of this chart represent things in the real world, and which parts represent things that only live in our computers?

Associations in Code, again

Discussion Question: What real-world things do these classes and associations represent?


```
class Teacher {
    coursesTaught : Course[]
}
```

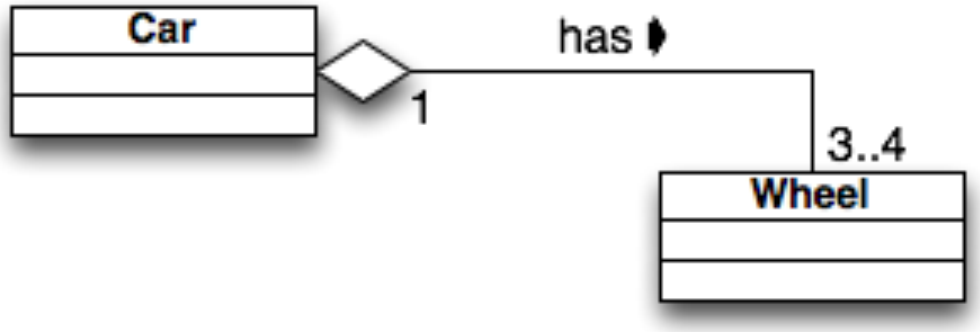
```
class Course {
    instructor : Teacher
    roster : Student[]
}
```

```
class Student {
    classesTaking : Course[]
}
```

```
// INVARIANT:
// c.instructor = t iff c is in t.coursesTaught
// s in c.roster iff c is in s.classes
```

Discussion: what do Car and Wheel represent?

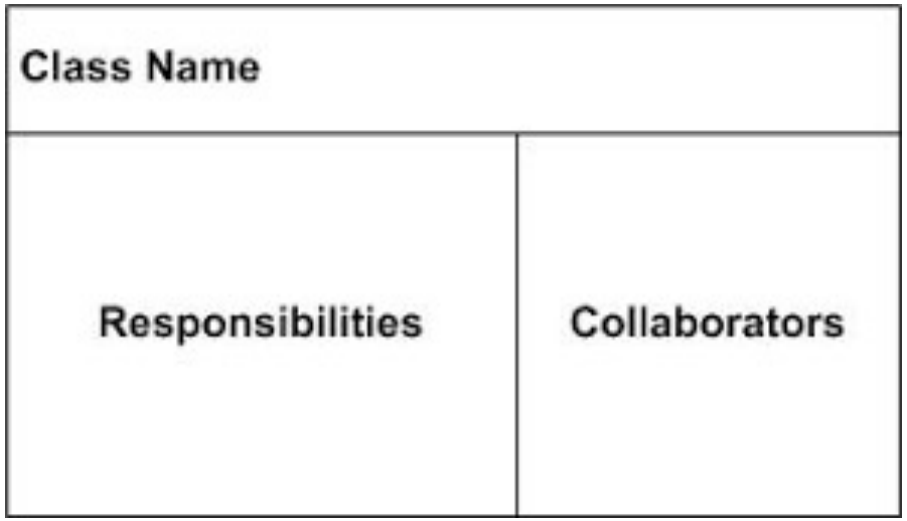
- A car has 3–4 wheels



The solid arrow indicates the way we should read "has" (a car "has" wheels, not wheels "has" a car).

Discussion Question: What should the navigability of this association be? Should we be able to get from a Car to the Wheels that it has? Should we be able to get from Wheel to Car?

Interlude: CRC Cards are a lighter-weight alternative to UML for initial design



- Class
  - the name of a "thing" in your program
  - could be a class, interface, type, etc.
- Responsibilities
  - the main job of this "thing" in the program
  - should be simple: Remember the Single Responsibility Principle
  - Might be “Manages <some piece of state>”
- Collaborator
  - Any class or interface that this class needs in order to fulfill its responsibilities
  - Goal is to make sure that each class has enough information to fulfill its responsibilities.
- Good for early-stage planning and design

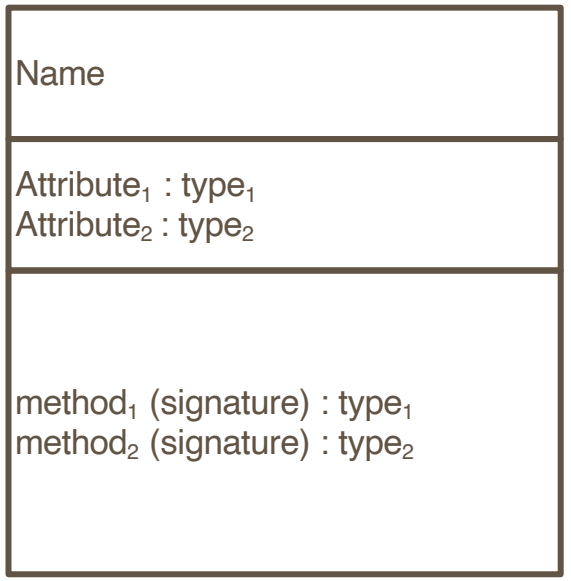
CRC Cards in Practice

- Typically used during early analysis, especially during team discussions.
  - Low-tech
  - 4x6 index cards
  - They aren't pretty.
  - They aren't something you ever want to show your customers or even your own upper management.
- Each card is a concrete symbol for a thing in the program during discussion
- Kind of like thinking on a whiteboard, but...
- Cards can be stacked, moved, etc. to illustrate proposed relationships
  - If you come out of a group meeting and your CRC cards aren't smudged, dog-eared, with lots of scratched-out bits, you probably weren't really trying.

https://www.cs.odu.edu/~zeil/cs330/live/website/Slides/crc/page/crc.html

Back to UML:
Level 2: Attributes, Methods, and Names

- A Class is drawn as a three-part box containing:
  - class name (required)
  - list of attributes with names and types (optional)
  - list of methods with argument lists (optional)
- Attributes and methods may be annotated with "+" for public and "-" for private.
- Components with special roles may be annotated with "stereotypes", which are written with `<<...>>`.



Attributes

- The attributes of a class are roughly those members (or "instance variables" or "properties", depending on what language you are writing in) whose values are either
  - scalars ("simple" attributes)
  - arrays or lists of scalars ("multivalued" attributes)
  - simple structs (e.g. dates or names)
- Class members whose values are full-fledged objects (of this or some other class) are usually represented in UML as relationships.

In TypeScript, functions are values, so for us an attribute could have a value that is a function. Your real-world boss may or may not agree.

Methods

- If there are many methods, they may not fit in a UML diagram.
- Alternatives:
  - Tables
  - Javadoc, etc.

This is the level where we specify names

- If you are going to write code, you need to know the **names** of the classes, methods, etc.

JSDoc

```
22    * @callback TilemapFilterCallback
23    *
24    * @param {Phaser.GameObjects.GameObject} value - An object found in the filtered area.
25    * @param {number} index - The index of the object within the array.
26    * @param {Phaser.GameObjects.GameObject[]} array - An array of all the objects found.
27    *
28    * @return {Phaser.GameObjects.GameObject} The object.
29    */
30
31   /**
32    * @callback TilemapFindCallback
33    *
34    * @param {Phaser.GameObjects.GameObject} value - An object found.
35    * @param {number} index - The index of the object within the array.
36    * @param {Phaser.GameObjects.GameObject[]} array - An array of all the objects found.
37    *
38    * @return {boolean} `true` if the callback should be invoked, otherwise `false`.
39    */
40
41   /**
42    * @classdesc
43    * A Tilemap is a container for Tilemap data. This isn't a display object, rather, it holds data
44    * about the map and allows you to add tilesets and tilemap layers to it. A map can have one or
45    * more tilemap layers, which are the display objects that actually render the tiles.
46    *
47    * The Tilemap data can be parsed from a Tiled JSON file, a CSV file or a 2D array. Tiled is a free
```

- You put structured comments in the code

And the tool turns it into web pages

Here’s our design again, with names.

My file imports **ClockFactory** from **'./ClockFactory.ts'**

When I create an object that needs a clock, I get a copy of the master clock by calling the static method **ClockFactory.instance()** , and the new object registers itself with the master clock by calling **c.register(this)** .

Whenever the master clock changes, it updates my object by sending it a **notify(t:Time)** message. It also sends my object a similar update message when anyone registers with it, so my object will always have the latest time.

Pat is responsible for **ClockFactory.ts** . Pat and I agreed on this protocol; Pat agreed that their clock factory will have an **instance()** method that returns the master clock, and that the master clock will have a **register()** method that I can use to register my object.

In return, I agreed that my object will have a **notify(t:Time)** method that the master clock can use to notify it about the updated time.

Review: Learning Objectives for this Lesson

- At the end of this lesson, you should be able to
  - Demonstrate the basics of UML class diagrams
  - Explain the significance of the different kinds of associations in UML
  - Explain what the reviewer of a design might want to know that UML leaves out