(CS360) Fundamentals of Software Engineering

Design documentation and patterns

Lecture 2

Dr. Qobiljon Toshnazarov

Outline

New Uzbekistan University

- Documenting your design
  - CRC cards
  - UML class and sequence diagrams
- Design patterns
  - Adapter, Composite, Iterator, Singleton, Factory, Observer, and Visitor

Controlling SW complexity

New Uzbekistan University

- Software systems must be comprehensible by humans

- Which humans?
  - The other members of your team
  - The folks who will maintain and modify your system
  - Management
  - Your clients
  - and …
  - you, a week from now or 6 weeks from now

SW design is more than code

New Uzbekistan University

- Design is about how your code relates to the real world
- Design is about the organization of the code
- Design is about the relationships between different pieces of the code
- So: you need a different language to talk about your design

Remember Principle #2:

Design your data!

Shared vocabulary in a SW project

New Uzbekistan University

- You and your teammates need to have a common understanding of the things in your program.
  - What are their names?
  - What do they represent?
  - How do they interact?

You get to make up the names. But you should make them Good Names, of course

There are standard names for many of these interactions. These are called Design Patterns.

Design languages

New Uzbekistan University

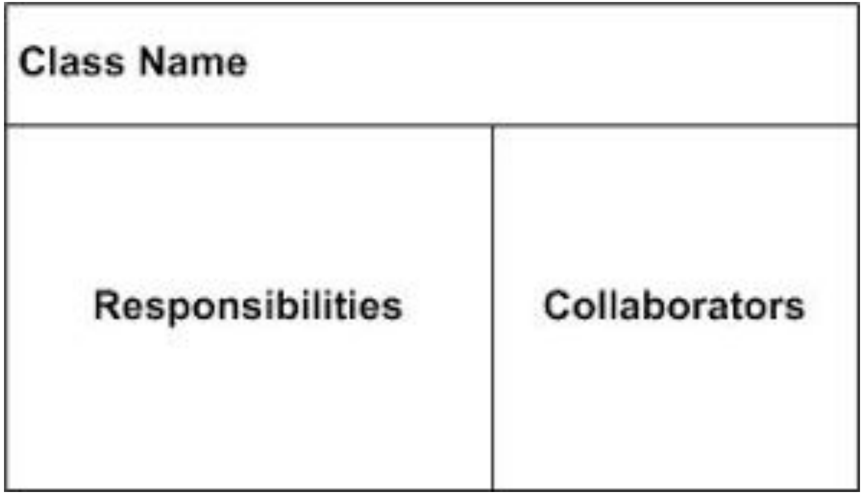
- We'll study two design languages
  - CRC cards (class-responsibility-collaboration)
  - UML diagrams (unified modeling language)

- These are very different languages for describing designs
  - Different level of formality
  - Different scope

CRC cards

CRC cards

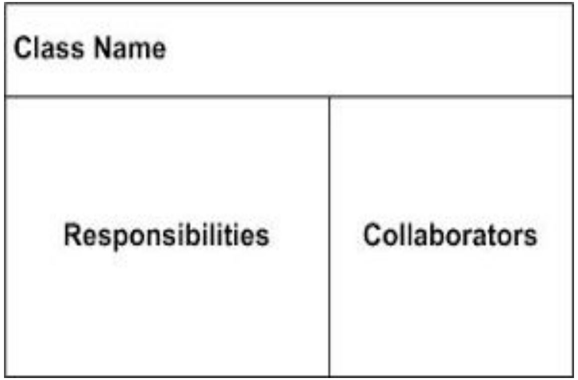
Class-responsibility-collaboration (CRC) cards look like this:



CRC cards

New Uzbekistan University

- **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 (SRP)”
- **Collaborators**
  - The other "things" with which this thing interacts
  - For us this means the things with which this thing is coupled (i.e., depends on)
  - Includes at least: all the things that this thing uses, and all the things that use this thing, at least directly



Some books say to list just the things that this thing depends upon.

Agile alliance about CRC cards

New Uzbekistan University

- CRC cards (for Class, Responsibilities, Collaborators) are an activity bridging the worlds of role-playing games and object-oriented design.
- With the intent of rapidly sketching several different ideas for the design of some feature of an object-oriented systems, two or more team members write down on index cards the names of the most salient classes involved in the feature. The cards are then fleshed out with lists of the responsibilities of each class and the names of collaborators, i.e. other classes that they depend on to carry out their own responsibilities.
- The next step is to validate – or invalidate as the case may be – each design idea by playing out a plausible scenario of the computation, each developer taking on the role of one or more classes.

https://www.agilealliance.org/glossary/crc-cards

CRC cards in practice

New Uzbekistan University

- 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

The metaphor: sketching the conspiracy

New Uzbekistan University

CRC cards in this course

New Uzbekistan University

- Assignment 5 (part of it) will ask you to use CRC cards and UML diagrams to document a SW.
- You may not be able to identify all the classes that use your class. Don't worry too hard about that.
- We will also ask you to put one more thing on your CRC cards - **the state**
  - All objects of a given class keep state information that may change over time, influencing object’s behavior or appearance in the SW

CRC card template

New
Uzbekistan
University

Class Name:

State:

Responsibilities	Collaborators

Download this template via this link:

You can print it out and write on by hand, or you can even simply use an online platform that supports drawing CRC cards.

CRC card for `TemperatureSensor`


```
// temperatures are measured in Celsius
type Temperature = number

interface TemperatureSensor {
    // return the current temperature
    // at the sensor location
    getTemperature () : Temperature
}
```

CRC cards are supposed to be *informal*, so don't get hung up on emulating the exact words or the exact layout used here.

Class name: TemperatureSensor (an interface)	
State: none	
Responsibilities	**Collaborators**
Defines interface for thermometers in the system	`RefrigeratorThermometer`
	`OvenThermometer`
	`TemperatureMonitor`
	etc.

TemperatureMonitor

New Uzbekistan University

```
class TemperatureMonitor {
   constructor(
      // the sensors
      private sensors: TemperatureSensor[],
      // map from sensor to its location
      private sensorLocationMap: SensorLocationMap,
      private maxTemp: Temperature,
      private minTemp: Temperature,
      private alarm: IAlarm,
   ) { }

   // sensor in range?
   private isSensorInRange (sensor:TemperatureSensor) : boolean
   {
      const temp: Temperature = sensor.getTemperature()
      return ((temp < this.minTemp) || (temp > this.maxTemp))
   }
.
.
.
```

Here's a slightly more elaborate `TemperatureMonitor` class

It monitors multiple sensors

And it knows where each sensor is

Better division into one method/one job than our earlier version.

TemperatureMonitor (contd.)

New Uzbekistan University

```
.
.
.

   // if the any of the sensors is out of range, sound the alarm
   public checkSensors(sensor: TemperatureSensor): void {
      this.sensors.forEach(sensor => {
         if (!(this.isSensorInRange(sensor))) {
            this.soundAlarm(sensor)
         }
      })
   }

   private soundAlarm (sensor) {
     const location = this.sensorLocationMap.getLocation(sensor)
     this.alarm.soundAlarm(location)
   }
}
```

CRC card for `TemperatureMonitor`

New Uzbekistan University

Class name: `TemperatureMonitor`	
State: Some sensors, `maxTemp`, `minTemp`, the alarm to sound, map from sensors to locations	
Responsibilities	**Collaborators**
If any of the sensors is out of range, tell the alarm to sound at its location	`TemperatureSensor`
	`SensorLocationMap`
	`IAlarm`

IAlarm

New Uzbekistan University

```
// sound alarm for issue at the given location
interface IAlarm { soundAlarm(location:Location): void }
```

Class name: IAlarm (interface)	
State: none	
Responsibilities	**Collaborators**
Defines interface for classes that will sound an alarm	`TemperatureMonitor`
	`all implementations of IAlarm`

SensorLocationMap

New Uzbekistan University

```
class SensorLocationMap {
    private locationMap : Map<TemperatureSensor, Location> = new Map()

    // get the location, if any.  If none, throw error
    public getLocation (sensor:TemperatureSensor) : Location {
        if (this.locationMap.has(sensor)) {
            return this.locationMap.get(sensor)
        } else {
            throw new Error (`sensor ${sensor} location unknown`)
        }
    }

    // methods to add and remove sensors from the map...
}
```

SensorLocationMap

New Uzbekistan University

Class name: `SensorLocationMap`	
State: Map from sensors to their locations	
Responsibilities	**Collaborators**
Maintain the map from Sensors to their Location	`TemperatureMonitor`

`FireAlarm`

A hypothetical implementation of **`IAlarm`** interface

Class name: `FireAlarm`	
State: socket for communicating with Fire Dept	
Responsibilities	**Collaborators**
When sounded, call the FireDept	`IFireDept`
When FireDept responds, turn off alarm	

CRC - mapping the conspiracy

New Uzbekistan University

Class Name:	TemperatureSensor (interface)
State:	none
Responsibilities	**Collaborators**
establish interface for thermometers in the system	RefrigeratorThermometer
	OvenThermometer
	etc.
	TemperatureMonitor

Class Name:	TemperatureMonitor
State:	sensors, maxTemp, minTemp, alarm
Responsibilities	**Collaborators**
if any of the sensors is out of range, tell the alarm to sound at its location	TemperatureSensor
	SensorLocationMap
	IAlarm

Class Name:	SensorLocationMap
State:	Map from Sensors to their Location
Responsibilities	**Collaborators**
Maintain the map from Sensors to their Location	TemperatureMonitor

Class Name:	IAlarm (interface)
State:	none
Responsibilities	**Collaborators**
Interface for classes that will sound an alarm	TemperatureMonitor
	all implementations of IAlarm

Class Name:	FireAlarm
State:	socket for communicating with Fire Dept
Responsibilities	**Collaborators**
when sounded, call the FireDept	IFireDept
when FireDept responds, turn off alarm	

UML class diagram

Unified modeling language

- UML is a general-purpose visual modeling language developed by an industry consortium in 1997.
- Based on multiple prior visual modeling languages.
- Goal was to have a single standard representation for a large number of SE tasks.
- A large language: 13 different kinds of diagrams.
- Currently, UML is at version 2.5.1 (December 2017).

See UML.org and https://www.omg.org/spec/UML/

UML in the context of this course

New Uzbekistan University

- 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.
- So 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.

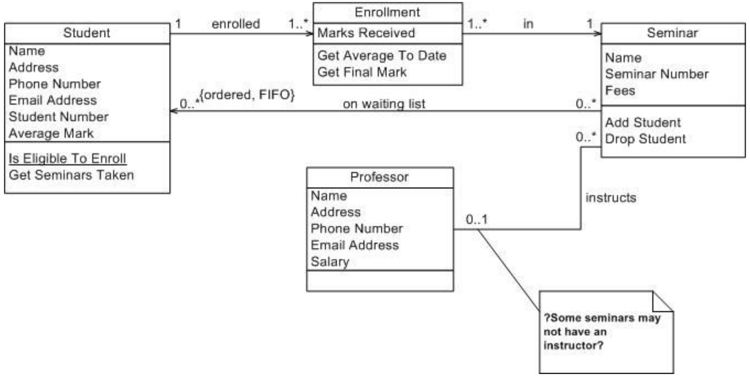
That will only demonstrate that you haven't thought hard about the problem ☺

Most common: UML **class** diagram

New Uzbekistan University

- Class diagram:
  - Which objects do we need?
  - Which are the features of these objects? (attributes, methods)
  - How can these objects be classified? (is-kind-of hierarchy, both via inheritance and interface)
  - What associations are there between the classes?

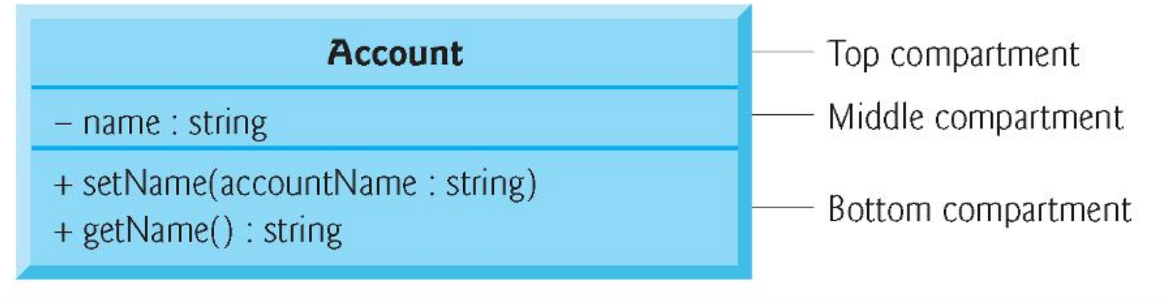
UML class diagram



- UML class diagrams summarize a class's attributes and operations.
- In industry, UML diagrams help systems designers specify systems in a concise, graphical, programming-language-independent manner, before programmers implement the systems in specific programming languages.

UML class diagram

New Uzbekistan University



- In the UML, each class is modeled in a class diagram as a rectangle with three compartments:
- The top compartment contains the class name centered horizontally in boldface type.
- The middle compartment contains the class's attributes, which correspond to the data members (i.e., state information).
- The bottom compartment contains the class's operations, which correspond to the member functions.

UML class diagram

New Uzbekistan University

- The UML class diagram lists a minus sign (–) access modifier before the attribute name for private attributes (or other private members).
- Following the attribute name are a colon and the attribute type.
- The UML models operations by listing the operation name preceded by an access modifier.
- A plus sign (+) indicates public in the UML.
- The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.
- More class diagram details: developer.ibm.com/articles/the-class-diagram

Class attributes in UML

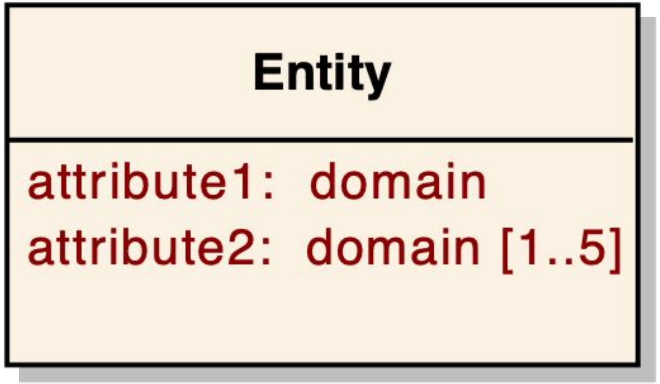
New
Uzbekistan
University

- 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.

Attribute example

New Uzbekistan University



attribute1 is simple.

attribute2 is multivalued (there can be up to five values stored on attribute2)

domain is UML terminology for “type”

UML has its own vocabulary for describing UML diagrams, with words like "entity" and "attribute" and "domain". A language like this, that describes another language, is sometimes called a metalanguage.

Relationships in UML

New Uzbekistan University

- UML has notations for 3 kinds of relationship between classes:
- Most general relationship: Association
- Special cases:
    - Generalization
    - Aggregation

Relationship #1: Association

New Uzbekistan University

- An association is a simple semantic relationship between two objects that indicates a link or dependency between them. For example:
  - 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.
- Associations can be directed, meaning there is a relationship from one object to another, or bi-directional, meaning the relationship works both ways.
- Relationships may be annotated with descriptions.
- An association may be implemented in several possible ways.

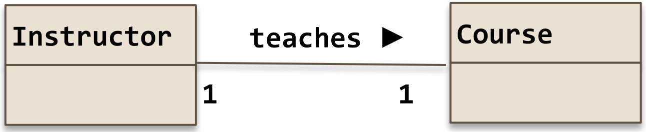
Properties of association: cardinality

New Uzbekistan University

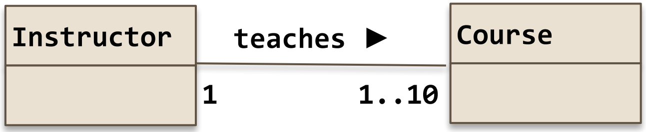
- Cardinality in UML class diagram is also called multiplicity
- The relationship between two entities has an associated cardinality or multiplicity
  - Multiplicity is expressed with specific numbers or ranges
  - e.g.: 1:1..2 or 1:1..N
- Examples:
  - A student is associated with exactly one transcript (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.

Notation of cardinality in association

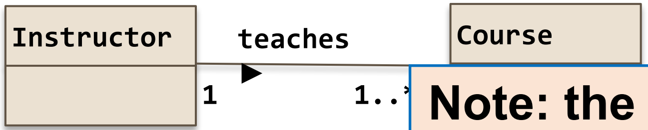
New Uzbekistan University



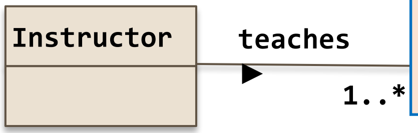
Any given instructor teaches <u>1 course.</u>
Any given course is associated with <u>one instructor.</u>



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

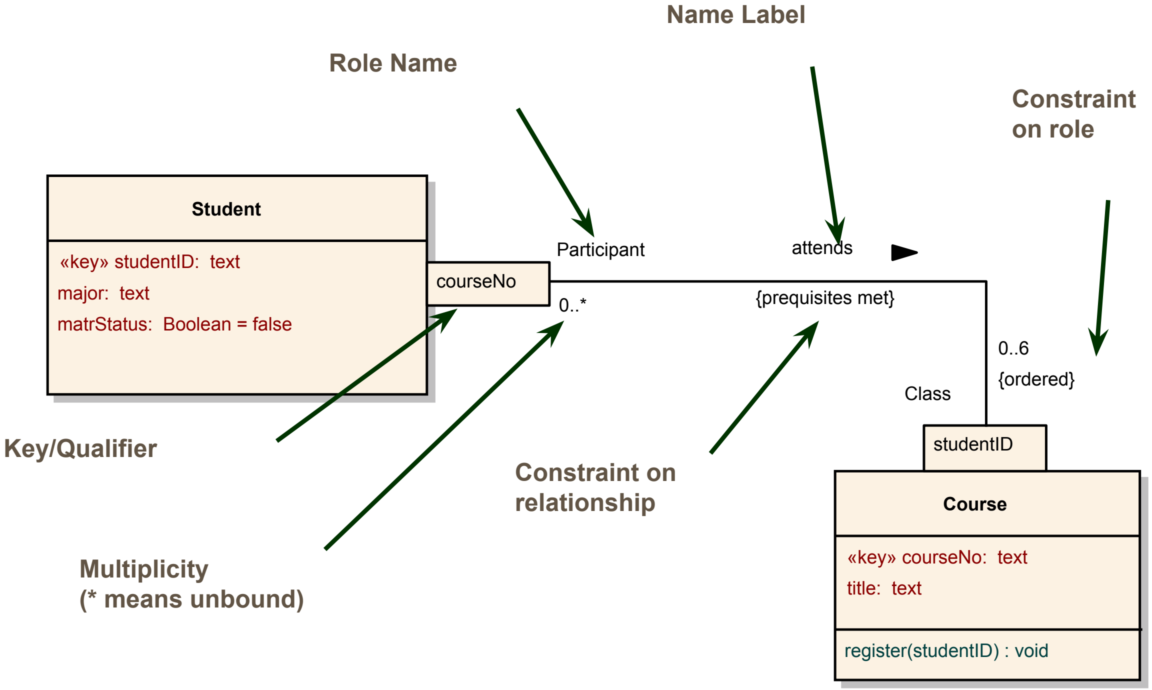


Any given instructor teaches <u>1 or more courses.</u>



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?)

Full association specification

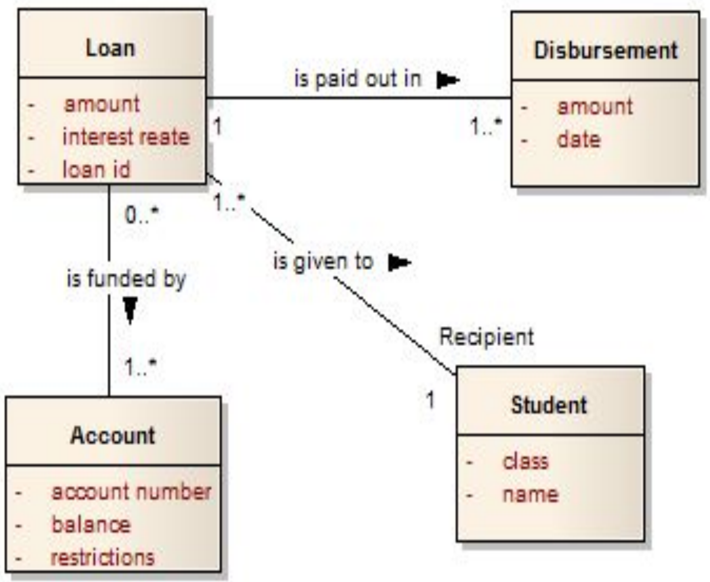


The UML folks tried to think of everything you could possibly say about an association.

Like much about SE, you only need to memorize the parts you need.

Association

Associations should reflect something about the real world



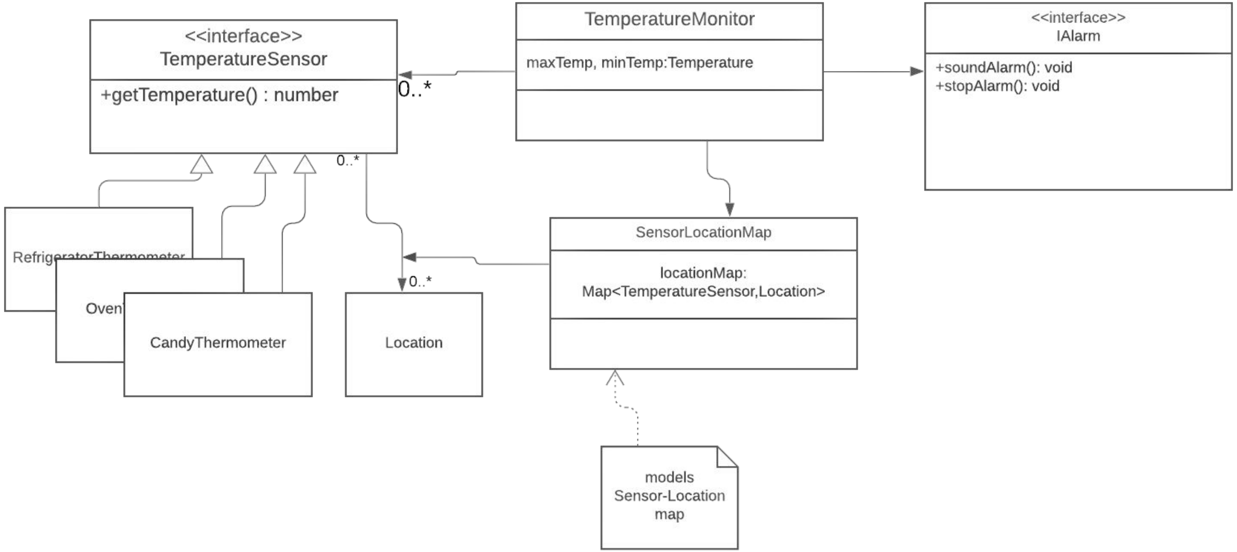
<u>Partial Translation</u>:

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?

New Uzbekistan University

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



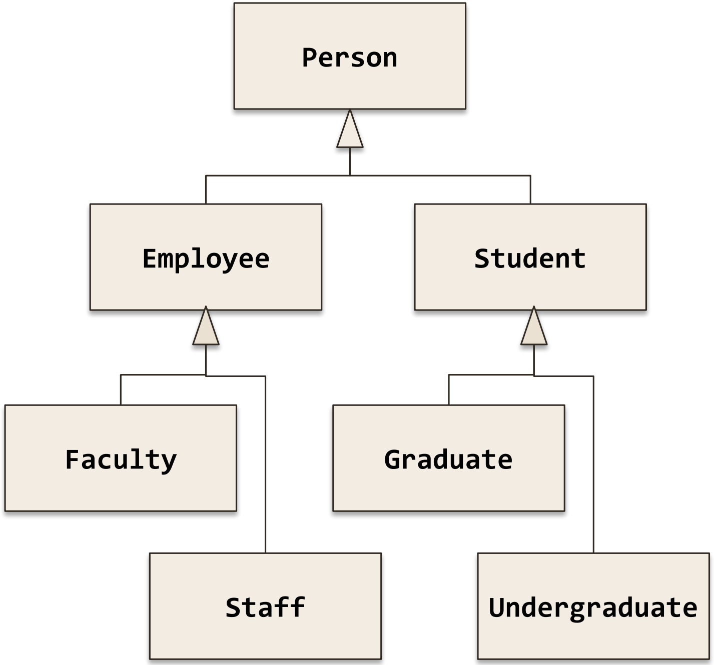
Discussion Question:

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

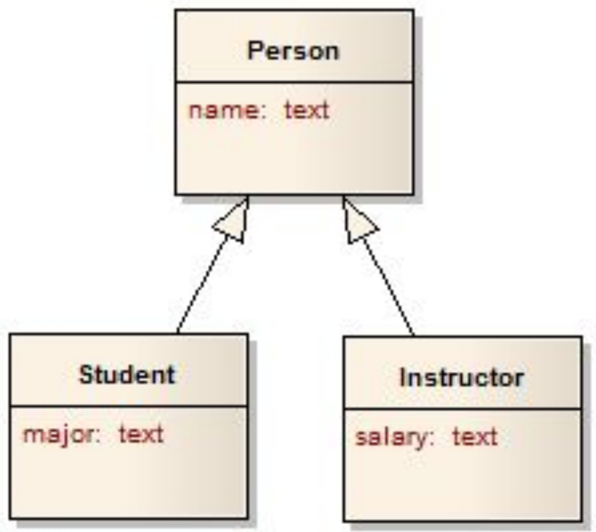
Relationship #2: Generalization

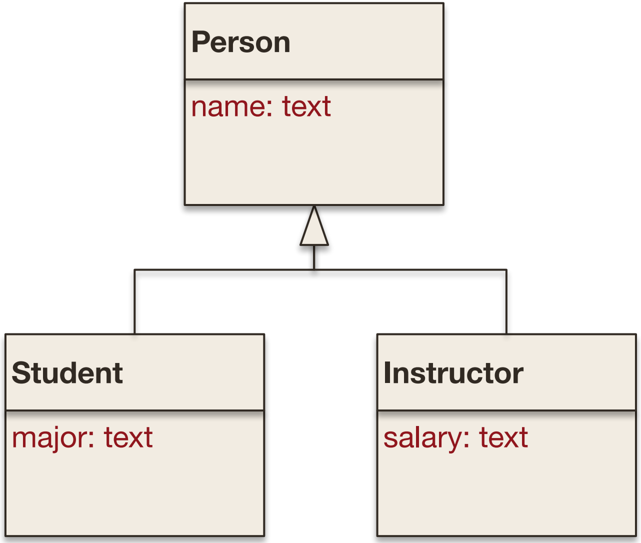
New Uzbekistan University

- Generalization is a grouping of entities based on common attributes.
    - describes an is-a-kind-of relationship between entities
    - like inheritance in OOP
- More general as you move up
- More specific as you move down
- More specific may inherit attributes and operations from the more general
    - may specialize attributes and operations



Generalization in UML

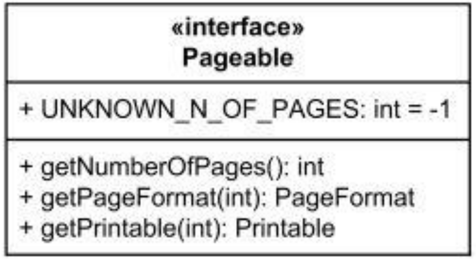




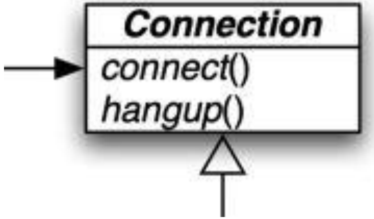
These two UML class diagrams are equivalent

Interfaces and "implements"

- In UML, the "implements" relation is generally considered to be a form of generalization.
- An interface is typically notated like a class, but with the stereotype <<*interface*>>.
  - Alternatively, the name of the interface may be given in italics.
- The "implements" relationship may be notated with a dotted or dashed line, or by an open-headed arrow.



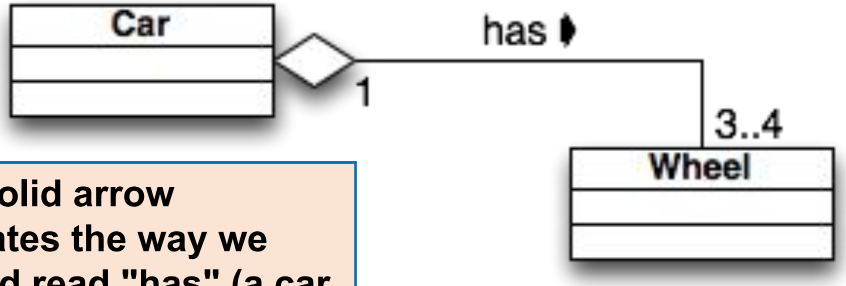
Interface *Pageable*



Relationship #2: Aggregation

New Uzbekistan University

- A car has 3–4 wheels



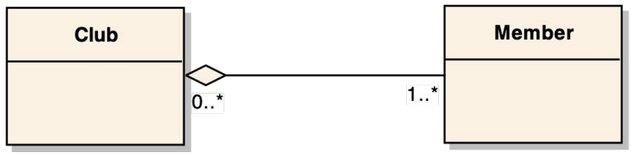
The solid arrow indicates the way we should read "has" (a car "has" wheels, not wheels "has" a car).

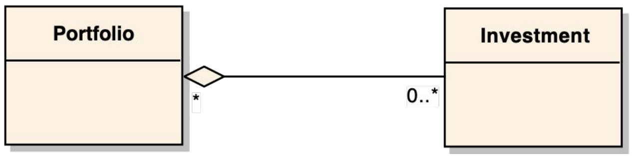
Discussion Question: What should the navigability of this association be? Should we be able to get from a Car to the Wheels that it has? Should we be able to get from Wheel to Car?

Aggregation

New Uzbekistan University

- Aggregation is an association that means a “whole/part” or “containment” relationship.
- The distinction between association and aggregation is not always clear.
- Don't stress about this: If in doubt, notate the relationship as a simple association.
- Generally, it's more important to show navigability than to worry about the diamond-shaped arrowheads.





What relation is portrayed in each of these diagrams?
What should its navigability be?

UML sequence diagram

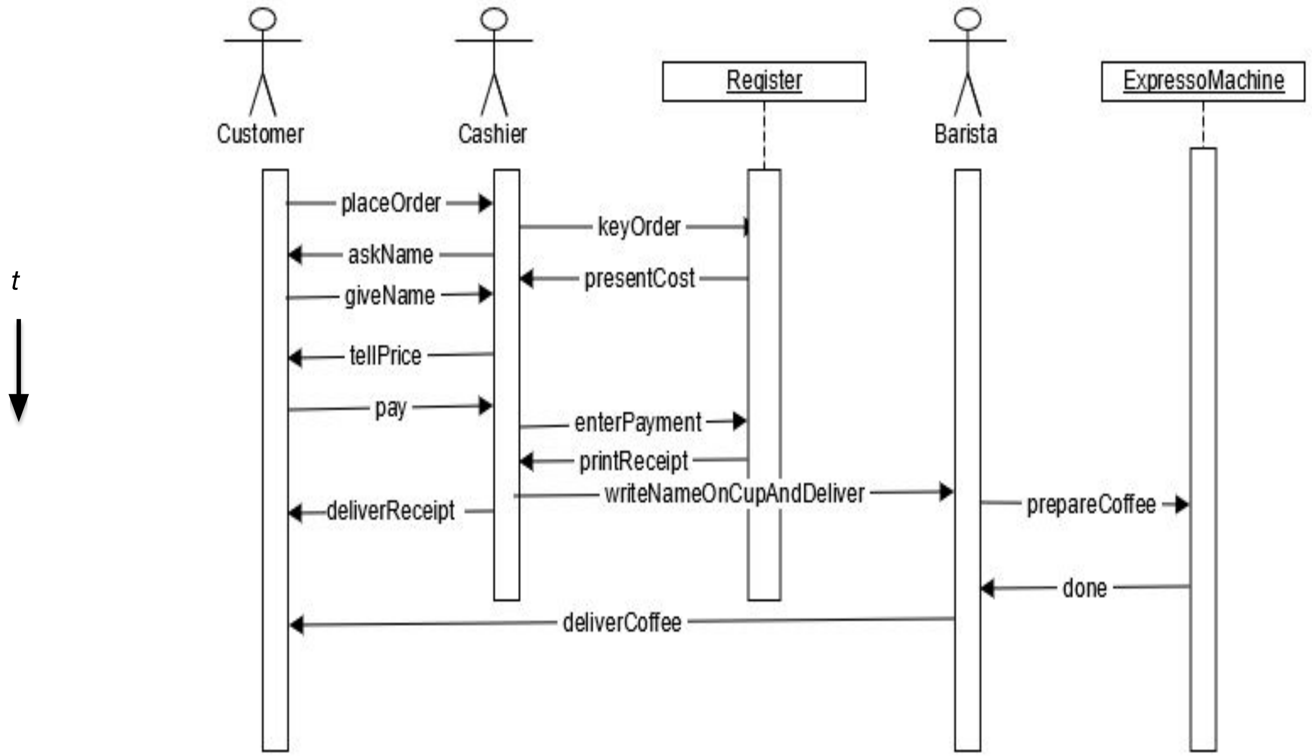
UML sequence diagram

New Uzbekistan University

- Shows the flow between elements of a system (the messaging sequence)
  - Classes (instances of classes)
  - Components
  - Subsystems
  - Actors
- Time is explicitly shown and flows from top to bottom (vertically, downwards)

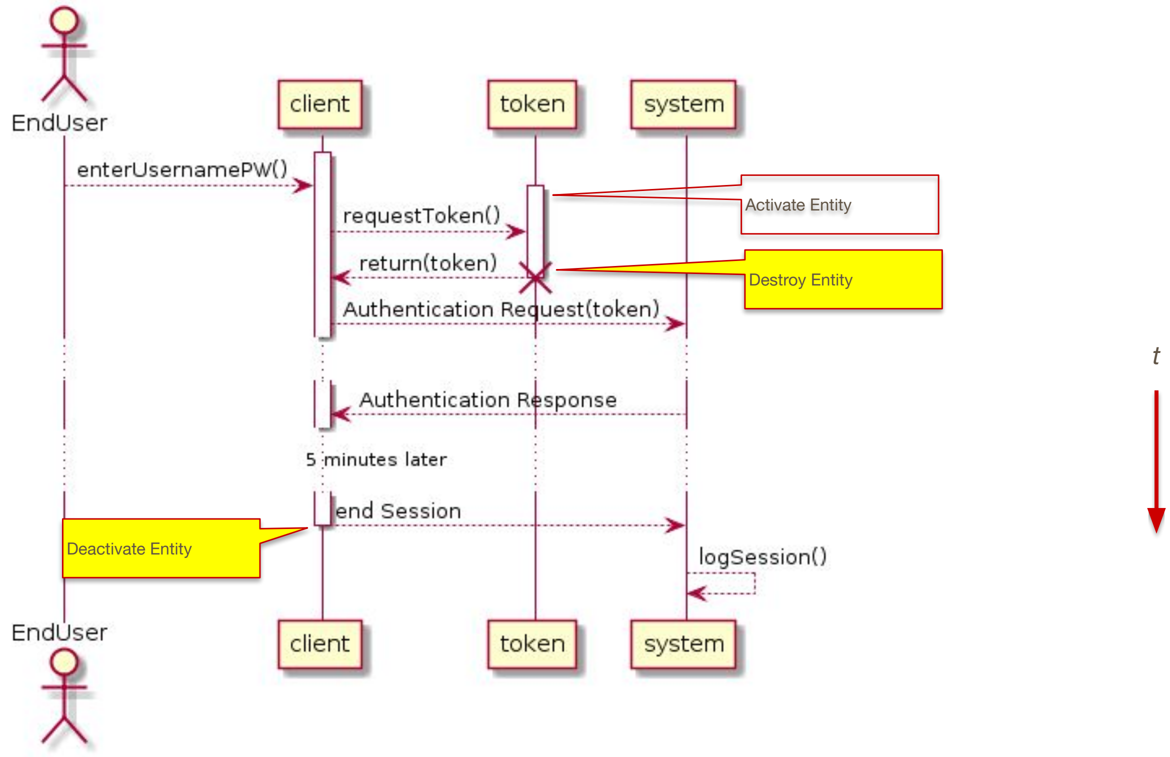
UML sequence diagram example

New Uzbekistan University



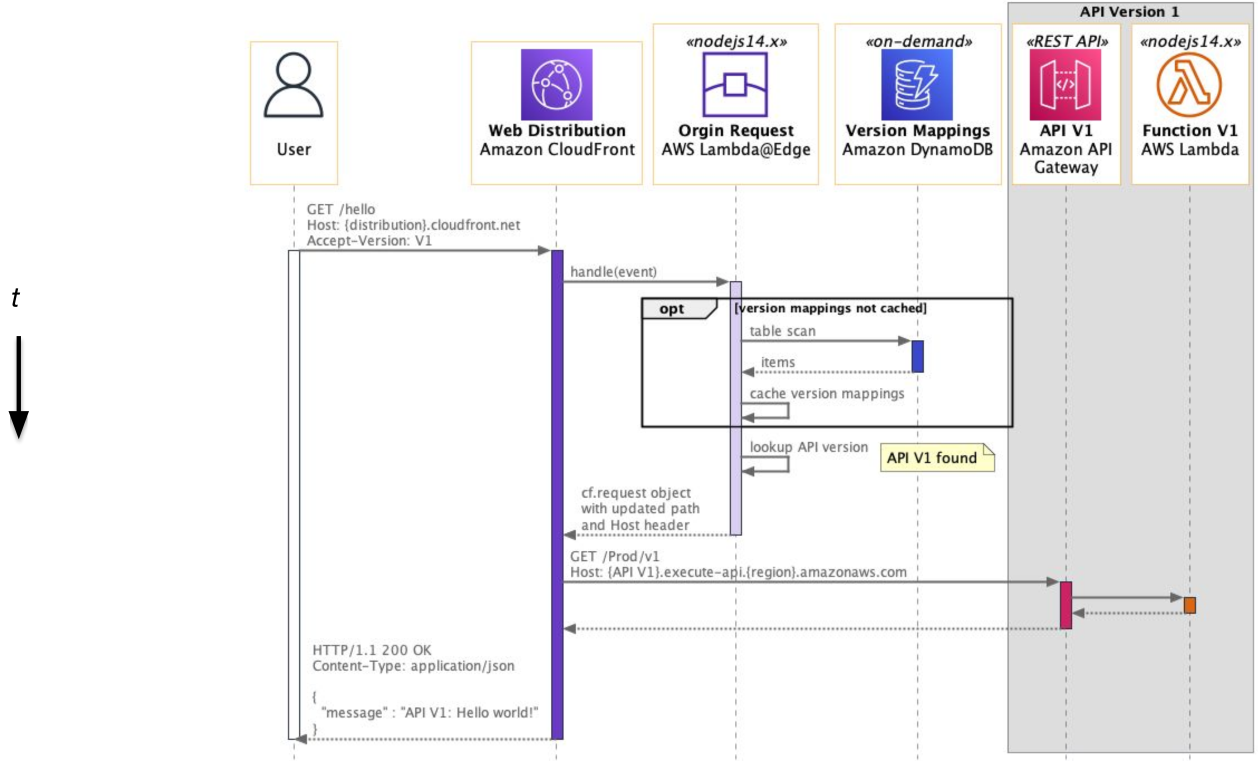
Another example

New Uzbekistan University



A more complicated example

New Uzbekistan University



https://aws.amazon.com/blogs/architecture/sequence-diagrams-enrich-your-understanding-of-distributed-architectures/

Design patterns

What is a design pattern?

New Uzbekistan University

- Think of it as advice from a master to a novice.
  - A master chef may advise a novice on knife technique.
  - A golf pro may advise a novice about their swing.
  - A piano teacher may advise a student about their posture, or how to interpret a piece.
- Often these pieces of advice are stylized and recorded.
  - e.g.: "keep your elbow straight" (golf) "use the tip of your knife as a fulcrum" (knife technique).
  - Maybe in a book of "technique"
  - Maybe on YouTube
  - etc.

What is in such a piece of advice?

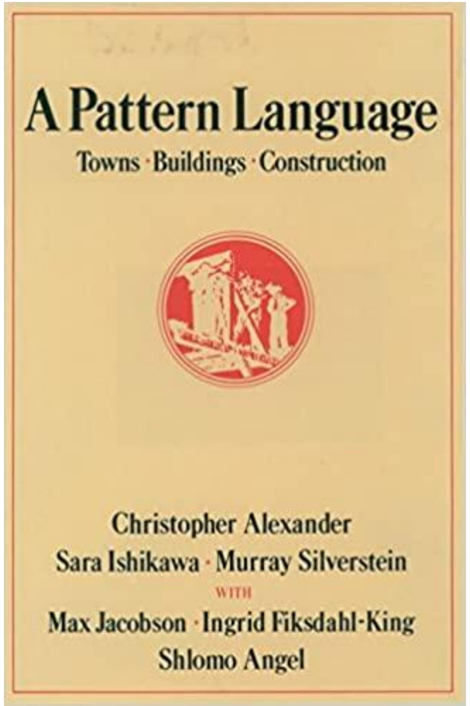
New Uzbekistan University

- A problem to be solved
  - "The golf ball keeps flying off to the side"
  - "It's taking too long to chop the carrots"
- A technique or method for solving the problem
- The technique always needs to be adapted to the problem at hand
  - Is the golf ball lying on a slope? what kind of slope?
  - Do you have a proper chopping board? What kind of knife are you using?

Design patterns in architecture

New Uzbekistan University

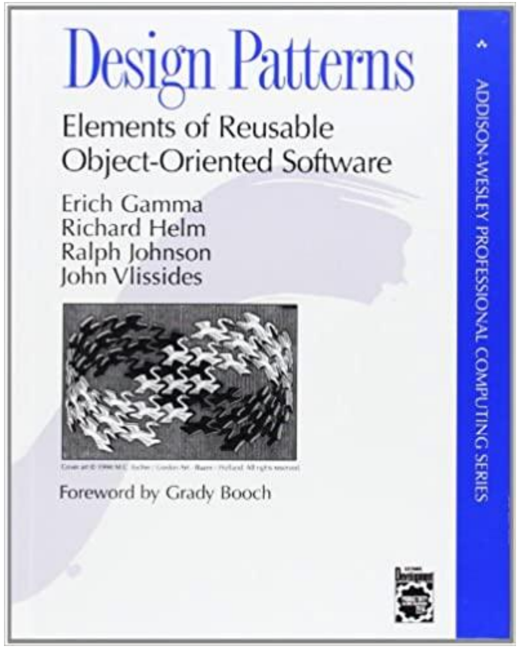
- "A Pattern Language: Towns, Buildings, Contruction" by Christopher Alexander (1977)
- Introduced this idea to a wide community beyond architects



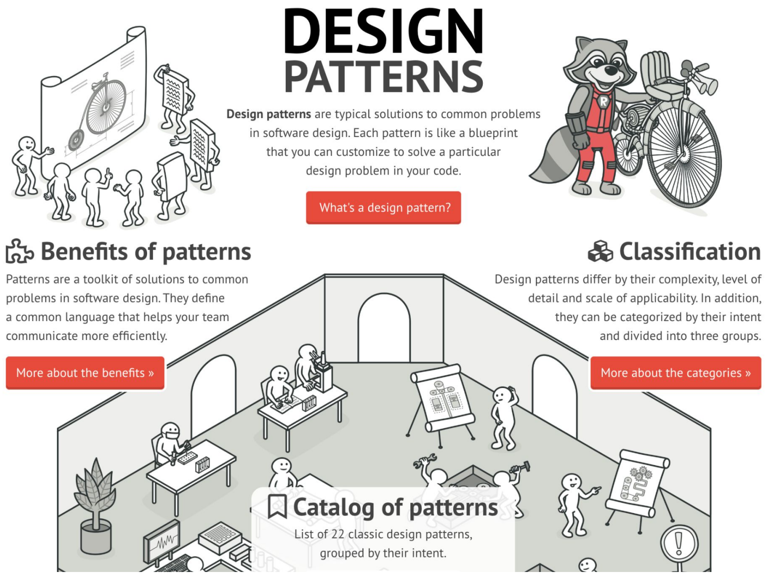
“Gang of four” book

New Uzbekistan University

- First (and only!) edition 1994
- Introduced this idea to object-oriented design
- Started the "Software Patterns" movement
- Still #1 on Amazon in Object-Oriented Software Design



Very good reference: refactoring.guru



https://refactoring.guru/design-patterns/

Design patterns - definition

New Uzbekistan University

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

According to Christopher Alexander (1977) - author of the book “A Pattern Language: Towns, Buildings, Contruction”

Elements of a design pattern

New Uzbekistan University

- The pattern name
- The problem (when to apply the pattern)
- The solution (describes the elements that make up the pattern)
- The consequences (the results and trade-offs of applying the pattern)

This is the official definition, taken from the GoF book.

But, when they get around to describing patterns, their descriptions rarely match this outline ☹

Design patterns are controversial

New Uzbekistan University

- For the last 25 years, software experts have lined up either as pattern fans or pattern skeptics
- Sometimes there are endless debates about whether a given piece of code is or is not an instance of a particular pattern.
- We are just not going to get into that. Chill.
- These patterns are tools in your toolbox.

Design patterns are everywhere

New
Uzbekistan
University

- Everytime you read a blog post or web page with some code illustrations, you are using a design pattern:
    - A piece of code to solve a particular problem
    - And which needs to be adapted to your particular situation.
- But some patterns are classics that have names that you should be familiar with.

Problem #1

New Uzbekistan University

- Suppose we need to implement a stack class with following interface:

```
// the usual stack operations
interface IStack<T> {
  push(t: T): void
  pop(): T
  size(): number
}
```

- But we have a class **List** that implements **IList:**

```
interface IList<T> {
  // add to end of list
  add(t:T): void
  // remove last element of the list
  remove(): T
  // returns the number of elements in the list
  size(): number
}
```

Of course, in Typescript you'd never do this, because in Typescript we almost always use arrays to represent lists.

Solution: Adapter

```
class Stack<T> implements IStack<T> {

  // top of stack is at end of list
  constructor (private payload: IList<T>) {}

  public push(t: T): void {
    this.payload.add(t);
  }
  public pop(): T {
    return this.payload.remove();
  }

  public size(): number {
    return this.payload.size()
  }

}
```

Important: if you do something like this, be sure to explain how the list should be interpreted as a stack. (Remember Design Principle 2!)

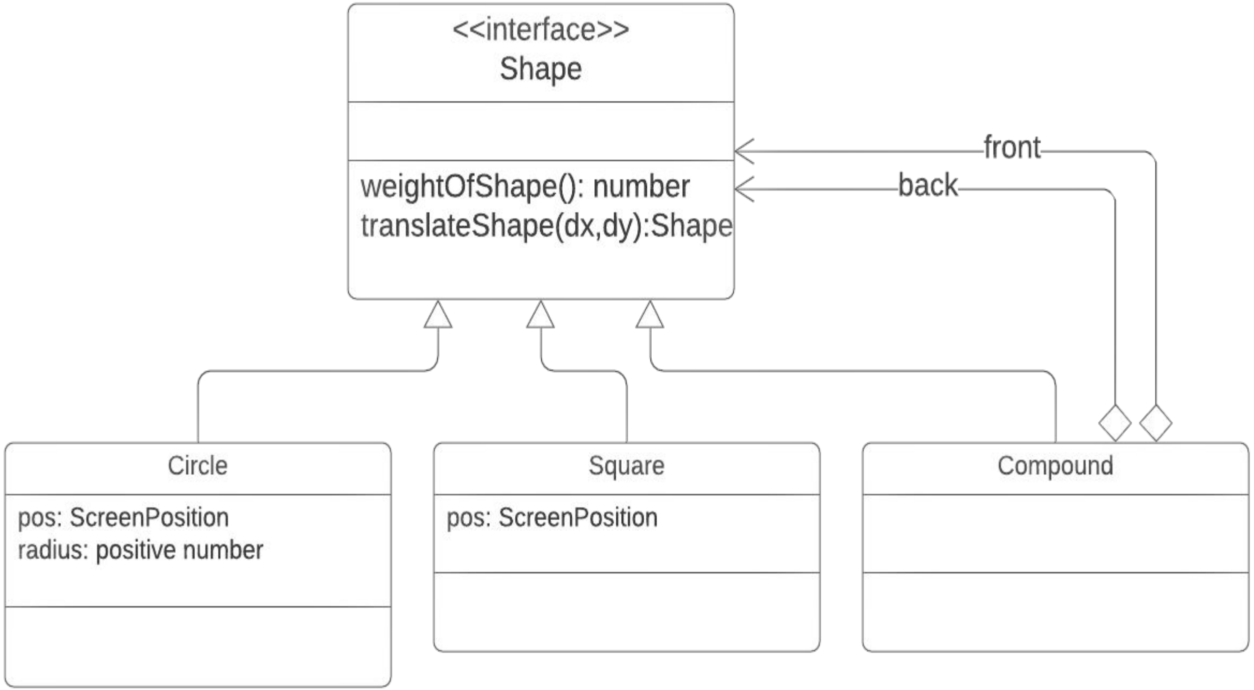
Problem #2:

New Uzbekistan University

- You need to represent data that is tree-like
- For instance, representing shapes
- A shape is either:
    - A square
    - A circle
    - Or a compound of two shapes: a front shape and a back shape.

Solution: the Composite pattern

New Uzbekistan University



Giving one class a reference to an object of another class (or interface) is sometimes called Composition.

Notice the circular dependency between Shape and Compound. That comes along with hierarchical (tree-like) data.

There's no avoiding it.

Problem #3

New Uzbekistan University

- You need to systematically go through the elements of some collection.
- Solution 1: Implement your collection using a type with native support for iteration.
  - In TypeScript, this typically means an array (a list) or Map
  - These are called *internal iterators*

```
const mylist : Shape[] = ...
mylist.map(shape => ...)
mylist.filter(shape => ...)
mylist.forEach(shape => ...)

for (s in mylist) {...}
```

The function that you apply to each element of the array is called the callback.

Internal iterators like these replace almost all loops in TypeScript.

Note: TS also allows iterators over Maps!

New Uzbekistan University

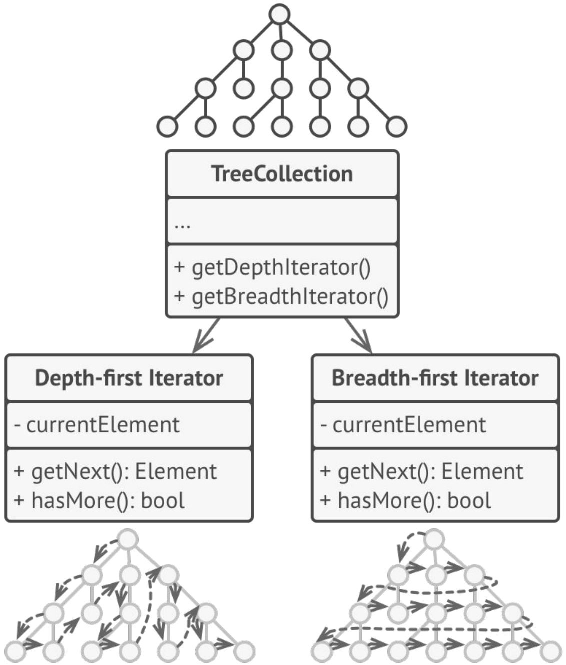
```
type StudentTableOut = Map<StudentId, StudentDataOut>

function countAllBins (studentMasterTable: StudentTableOut) {
    let histo = [0,0,0]   // a histogram with 3 bins
    for (let student of studentMasterTable.keys()) {
        let data = studentMasterTable.get(student)
        for (let question of data.keys()) {
            let questionData = data.get(question)
            let bin = questionData.bin
            histo[bin] += ...
        }
    }
    return histo.map(n => n/(histo[0]+histo[1]+histo[2]))
}
```

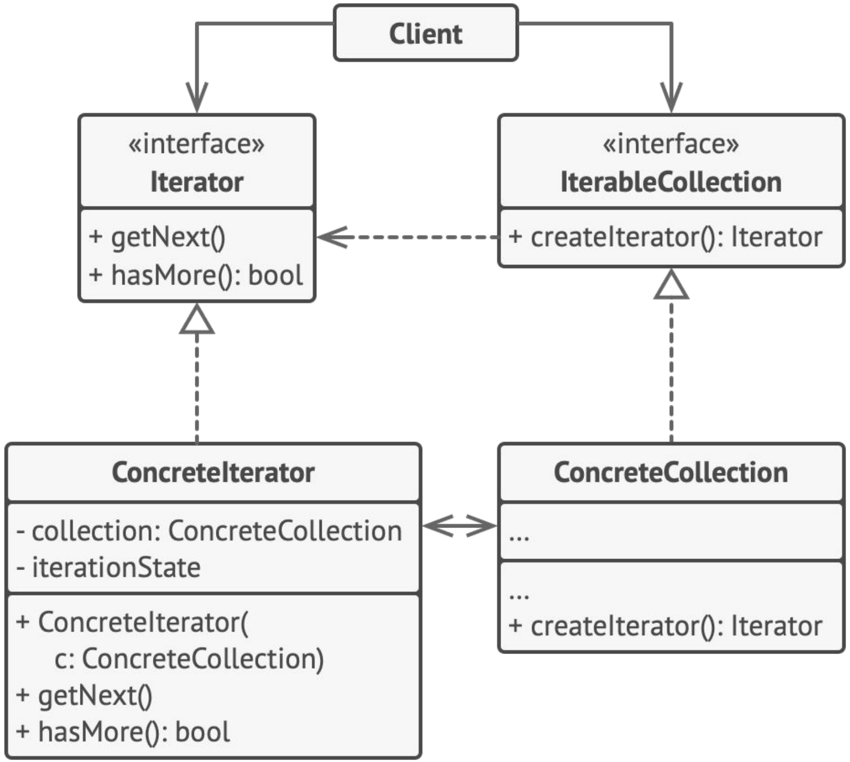
Solution: the Iterator pattern

New Uzbekistan University

- Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
- The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.



Example structure



Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).

Problem #4

New Uzbekistan University

- We have to make sure there is only one instance of a particular class
- Primary examples where only one instance is allowed:
  - A clock
  - A storage allocator
  - A generator for unique identifiers

Solution: the Singleton pattern

New Uzbekistan University

We'll do this in stages…

A simple clock

New Uzbekistan University

```
export interface IClock {
    // reset the tick counter to 0
    reset(): void
    // increment the tick counter
    tick(): void
    // returns the number of ticks since the last reset.
    currentTime(): number
}

class Clock implements IClock {
    private ticks = 0
    public reset():void { this.ticks = 0 }
    public tick():void { this.ticks++ }
    public currentTime():number { return this.ticks }

}
```

A clock factory

New Uzbekistan University

```
class FactoryMadeClock implements IClock {
    private ticks = 0
    public reset():void { this.ticks = 0 }
    public tick():void { this.ticks++ }
    public currentTime():number { return this.ticks }
}

// no need to instantiate ClockFactory
// just say ClockFactory.getClock()
export default class ClockFactory {
    public static getClock():IClock {return new FactoryMadeClock()}
}
```

calling ClockFactory.getClock() returns a new clock

Note that getClock is static, so you don't need to instantiate ClockFactory.

This is an instance of the Factory Pattern (yet another pattern whose name you should know).
This pattern doesn't add much value here, but it would be helpful if you were building something more complicated, e.g. an Amazon product listing.

A Singleton clock factory

New Uzbekistan University

```
class Clock {
    ..same as before..
}

export default class SingletonClockFactory {
    private constructor() {}

    private static _theClock: Iclock

    // have we initialized the clock?
    private static _isInitialized : boolean = false

    public static getClock() {
        if (!this._isInitialized) {
            this._theClock = new Clock()
            this._isInitialized = true  // it's initialized now
        }
        return this._theClock
    }
}
```

Like the ClockFactory, but this one cheats and only makes a clock once. Then it returns that same clock every time.

Make the factory's constructor private, so that no one can create another one

Use a first-time-through switch

Let's test this...

New Uzbekistan University

```
import {assert} from 'chai'
import SingletonClockFactory from './SingletonClockFactory'

function test1 () {
    let clock1 = SingletonClockFactory.getClock()
    let clock2 = SingletonClockFactory.getClock()
    clock1.tick()
    assert.equal(clock1.currentTime(),1)
    clock1.tick()
    assert.equal(clock1.currentTime(),2)
    assert.equal(clock2.currentTime(),2, "clock2 should see clock1's ticks")
    clock2.tick()
    assert.equal(clock1.currentTime(),3, "clock1 should see clock2's ticks")
}


describe ('check that clock is a singleton', () => {
    it('test1', test1)
})
```

Problem #5

New Uzbekistan University

- You have an object that changes state, and there are many other objects in the system that need to know this.
- But you don't know who they are
  - They may even be created after the object that is being watched.
- Example: we have a master clock, and other objects need to know the current time.

Solution: the Observer pattern

New Uzbekistan University

- Also called "publish-subscribe" (or pub-sub)
- The object being observed (the "subject") keeps a list of the people who need to be notified when something changes.
- When a new object wants to be notified when the subject changes, it registers with ("subscribes to") with the subject.

Interfaces


```
export interface IPublishingClock {
    // reset the tick counter
    reset(): void
    // increment the tick counter
    tick(): void
    // subscribe a new observer
    subscribe(obs: ClockObserver) : void
}

export interface ClockObserver {
    // action to take when clock ticks
    onTick(time: number): void
    // action to take when the clock resets
    onReset():void

}
```

No 'getTime' method! The clock *pushes* information to the observers

The protocol is:

1. **When the clock ticks, it sends an onTick message with the current time to each subscriber (observer)**
2. **When the clock resets, it sends an onReset message to each subscriber.**
3. **When a new subscriber registers, the clock responds by sending it an onTick message**

Names like 'onTick' are typical for methods in the Observer pattern

The Clock


```
class Clock implements IPublishingClock {

    // clock functionality
    private clockTime = 0
    public tick () {this.clockTime++; this.publishTickEvent()}
    public reset() {this.clockTime=0; this.publishResetEvent()}

    private observers : ClockObserver[]

    // register responds with the current time, so the observer
    // will be initialized
    public subscribe(obs: ClockObserver): void {
        this.observers.push(obs);
        obs.onTick(this.clockTime)
    }
    private publishTickEvent() {
        this.observers.forEach(obs => {obs.onTick(this.clockTime)})
    }

    private publishResetEvent() {
        this.observers.forEach(obs => {obs.onReset()})
    }
}
```

Push vs Pull

New Uzbekistan University

- In the simple model (like the one in singleton), a client pulled information from the clock.
- In the observer model, the clock pushes information to its clients

Exercise: Draw UML sequence diagrams for the simple clock and for the publishing clock.

Problem #6 (last one)

- You have a hierarchical structure, and there are many operations that will need to traverse it.
- You don't know in advance what those operations will be.
- But each operation can be implemented imperatively, perhaps by accumulating the answer in some variable.
- Also, you'd like to keep the internal organization of each node in the structure hidden from the operation.

Solution: the Visitor pattern

- Behavioral design pattern for separating behaviors from objects on which they operate.
- First thing that comes to mind is polymorphism with method overloading, but since the exact class (data type) of a node object is unknown in advance, this does not work.
- Visitor pattern resolves this issue using a technique called Double Dispatch - where the objects themselves (since they know their own classes) pick a proper method on the visitor by “accept”ing a visitor and redirecting to a visiting method to be executed.
- To invoke the operation / behavior, create a new object of the Visitor class.

This is called the Visitor class.

Let's call that the visitor (with a small v).

Visitor pattern example

New Uzbekistan University

```
// operates on a node
// the node itself is responsible for invoking the
// visitor on its descendants, if any.
interface ShapeVisitor {
    visitCircle(c: Circle): void
    visitSquare(sq: Square): void
    visitCompound(c: Compound): void
}

// a Shape is any class that will accept a Shape Visitor
interface Shape {
    // calls back the appropriate method of the visitor.
    // also sends the visitor to each child of the shape
    accept (v: ShapeVisitor) : void
}
```

The name 'accept' is not fixed. It is up to you how to call it, but it's what everybody calls it So if you see a method called 'accept' or 'acceptVisitor' in a codebase, that probably means that there's a visitor pattern here.

Compound + Visitor: In-order traversal

New Uzbekistan University

```
class Compound implements Shape {
    public accept (v: ShapeVisitor) {
        // apply the visitor using in-order traversal
        this.back.accept(v);
        v.visitCompound(this);
        this.front.accept(v)
    }
    constructor(private front:Shape, private back: Shape){}
    public getFront() : Shape { return this.front }
    public getBack() : Shape { return this.back }
}
```

When a Compound accepts:

1. **Passes the visitor onto its back shape.**
2. **Sends itself to the appropriate method of the visitor for local processing**
3. **Passes the visitor on to its front shape.**

The front and back properties are private to preserve encapsulation.
We need getters to make their values available to v.visitCompound(). Or you could make them public if you wanted to allow the visitor (or anybody else) to change them.

It's up to the node to decide the order in which these operations happen. This order is called in-order traversal. Other possible orders are called pre-order, and post-order.

Visitor pattern example (detailed)

New Uzbekistan University

Shape class hierarchy with a shape visitor:

```
export interface Shape {
    accept(visitor: ShapeVisitor): void;
}

export class Circle implements Shape {
    constructor(public radius: number) {}

    accept(visitor: ShapeVisitor): void {
        visitor.visitCircle(this);
    }
}
```

```
export class Square implements Shape {
    constructor(public sideLength: number) {}

    accept(visitor: ShapeVisitor): void {
        visitor.visitSquare(this);
    }
}

export class Triangle implements Shape {
    constructor(
        public base: number,
        public height: number
    ) {}

    accept(visitor: ShapeVisitor): void {
        visitor.visitTriangle(this);
    }
}
```

Visitor pattern example (detailed)

New Uzbekistan University

The shape visitor interface:

```
export interface ShapeVisitor {
   visitCircle(circle: Circle)
      : void;

   visitSquare(square: Square)
      : void;

   visitTriangle(triangle: Triangle
      : void;
}
```

A concrete visitor behavior/implementation

```
class AreaCalculator implements ShapeVisitor {
   totalArea: number = 0;

   visitCircle(circle: Circle): void {
       this.totalArea += Math.PI * …
   }

   visitSquare(square: Square): void {
       this.totalArea += square.sideLength…
   }

   visitTriangle(triangle: Triangle): void {
       this.totalArea += 0.5 * triangle.base…
   }
}
```

Visitor pattern example (detailed)

An example sample main.ts:

```
let shapes: Shape[] = [];
shapes.push(new Square(10));
shapes.push(new Circle(3));
shapes.push(new Triangle(4, 6));

let calc = new AreaCalculator();
shapes.forEach(shape => shape.accept(calc));

console.log(`Total Area: ${calc.totalArea}`);
```


Redo
Undrin
Q
&
A
J
Find
C
V
B
N