“System Design”
– *Design Principles & Methods [02]*

1 Object-Oriented Design

For a registration family office, an object-oriented design of an (average German conservative) family has to be done.

1.1 Class Model

Model the following facts as a UML class diagram with the help of Enterprise Architect and implement the classes in Java[1]:

- Men, women and children are described by their respective names and surnames; Women additionally by birth name.
- A man can be married to one woman, a woman to one man.
- Families consist of a man, a woman and children.
- Men and women know their respective children, of course, just as, conversely, the children know their parents.
- Men and women also know their spouse.

The following methods should be provided:

- new persons should be created with first name, last name and birthday (i.e. date of birth)
- for men and women: `marries(spouse, marriageDate)` which should return a new “family” object
- for women: `newChild(first name, birthday)` which should return a new child with the given first name.
- meaningful `toString()`-methods

Pay attention to a meaningful inheritance structure – especially maintain abstraction levels and inheritance as specialization. For the date use the `java.time.LocalDate`. To create a new date use the static method `LocalDate.of(int year, Month month, int dayOfMonth)`.

A simple test class, using your classes may look like this:

```
public static void main(String[] args) {
     Man john = new Man("John", "Doe", LocalDate.of(1990, OCTOBER, 9));
     Woman jane = new Woman("Jane", "Miles", LocalDate.of(1991, MARCH, 15));

     john.marries(jane, LocalDate.of(2015, AUGUST, 1));
     Child jack = jane.newChild("Jack", LocalDate.of(2016, DECEMBER, 24));
     Child jill = jane.newChild("Jill", LocalDate.of(2018, JULY, 15));

     System.out.println(john);
     System.out.println(jane);
```

[1] You don’t have to generate the code – use EA simply as a “drawing tool”.

```
        System.out.println(jack);
        System.out.println(jill);

    }
```

With the following results:

```
John Doe, born 9 OCTOBER 1990
Jane Doe, born 15 MARCH 1991 as Jane Miles
Jack Doe, born 24 DECEMBER 2016
Jill Doe, born 15 JULY 2018
```

Add additional methods for getting the children of a mother or her husband.

2 Service-Oriented Design

Your above application should be extended with a horoscope. A method `getHoroscope()` should return todays horoscope for a given person. To do so, use Tapasweni Pathak's Horoscope-API (https://github.com/tapasweni-pathak/Horoscope-API).

For a simple access, use the given utility class `ServiceAccess`, which encapsulates the call and returns a Json-Object with the respective result (or **`null`**). The class `SampleHoroscopeAccess` gives an example, how to use the class. To get it running you have also to add the supplied `json.jar` to your build path.

A simple test class, using your classes may look like this:

```
    public static void main(String[] args) {
        Man john = new Man("John", "Doe", LocalDate.of(1990, OCTOBER, 9));
        Woman jane = new Woman("Jane", "Miles", LocalDate.of(1991, MARCH, 15));

        john.marries(jane, toMinutes(2015, AUGUST, 1));
        Child jack = jane.newChild("Jack", LocalDate.of(2016, DECEMBER, 24));
        Child jill = jane.newChild("Jill", LocalDate.of(2018, JULY, 15));


        System.out.println(john.getHoroscope());
        System.out.println(jane.getHoroscope());
        System.out.println(jack.getHoroscope());
        System.out.println(jill.getHoroscope());
    }
```

With the following results:

```
John, LIBRA: Ganesha says your nature of spending wisely …
Jane, PISCES: A highly productive day awaits you. …
Jack, CAPRICORN: It's time to prove your extraordinary skills …
Jill, CANCER: Your speech will today explode your thoughts….
```

3 Design Principles

In which way do your classes "hide" information?

All fields should be private
If possible getters/setters should be as less accessible as possible

Does your top-level Person-class fulfil Liskov's Substitution principle for the horoscope-service? If so define a top-level (Junit-)test case and apply it to your subclasses.

The private modifier specifies that the member can only be accessed in its own class. The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

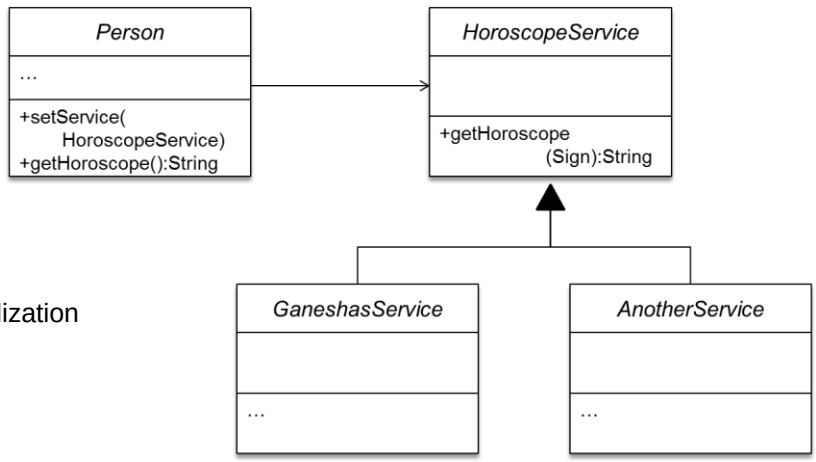
You want to add an address to your persons. One of your programmers comes up with the following code:

```
public class Address {

	protected final String street;
	protected final String city;
	protected final String code;
	protected final String country;

	public Address(String street, String city, String code, String country) {
		this.street = street;
		this.city = city;
		this.code = code;
		this.country = country;
	}
}
```

Then he makes your `Person`-class a subclass of the class `Address`. Is this acceptable? What would be an alternative solution?

It would violate the idea of using inheritance as a way of specialization

Your programmer suggest to decouple the horoscope acess in the following way:



The access to http://horoscope-api.herokuapp.com/horoscope will be encapsulated in GaneshasService, AnotherService may encapsulate a different service. According to your programmer, this realizes the Open-Closed-Principle. Which parts are open, which are closed and what is the advantage? Refactor your application in the above way.

4 Coupling and Cohesion

- What type of coupling do you have between your application and the service?
- What type of coupling do you have between you're the class `Person` and the `ServiceAccessor`?
- What type of cohesion provides the class `LocalDate`?
- What type of cohesion provides the class `Person`?
- What type of cohesion provides the method `ServiceAccessor.doRequest(…)`?

1. On syntactic level no coupling because the app will not throw any error and can be compiled if the service is removed. On semantic level - functional coupling because you are sending a request and receiving a response.
2. Stamp Coupling because we are importing another class and its functionalities.
3. Logical Cohesion because we have a sort of library
4. Abstract Cohesion because it it an abstract class
5. Sequential Cohesion because the method is executed step by step and one step uses the result from the previous step.