Informatics 1: Object Oriented Programming

Tutorial 07

Week 10: Object Design

Volker Seeker (volker.seeker@ed.ac.uk)
Vidminas Vizgirda (s1750767@ed.ac.uk)

1 Introduction

This course is about object-oriented programming and today's topic is about the essence of this theme – objects! In this tutorial you will practice:

- Translating word problems into object-oriented code
- Thinking about code design alternatives
- Evaluating object-oriented code written by other people

You will have seen in lectures all about creating new classes and objects, but such implementation is only one side of the coin. The other one has... random real life things?

Figure 1: "Objects". Image by Omar Elgabry published on Medium. © 2016 Omar Elgabry

What do the things in the picture above have in common? Well, they are all objects of sorts! The spoon, the basketball, the shopping bag and even people (although not in all contexts!) can be seen as objects.

Object-oriented programming borrows the concept of objects from real life and simplifies it to models: objects are represented by just the properties we care about. For example, when writing a basketball simulation program, we might create a ball object, which has the properties: diameter, air pressure, and roughness (but we might not care about its colour, materials, atom composition and an infinite number of other things that make up a real life basketball). In another context, such as an online shopping website for basketballs,

we might include diameter, roughness, colour, and weight properties for a basketball object instead (while ignoring the air pressure, since someone could inflate/deflate the basketball after buying it). It all depends on context.

Identifying what the relevant objects are in a problem and what properties should represent them is **object design**. It happens before implementation, perhaps in your head or perhaps on a piece of paper, and it does not involve code, yet it is an essential part of programming.

Programming also extends real life objects: in code objects can be not only things you can see and touch, but also abstract concepts, like time, bank accounts or events.

1.1 Worked example: dog daycare app

Suppose we were making a smartphone app that simulates a dog home. The player is the owner of this home and takes care of the dogs that are always either eating, sleeping, sitting or running around.

Small dogs should be kept separately from large dogs for safety. The owner must also keep in mind that young dogs will likely need walking much more frequently than old ones, albeit this depends on the breed too - some dogs are more independent than others.

When the dog owners come back to pick up their dogs, they don't need to be too precise about finding the exact dog they left - as long as the breed and colour match up and they have roughly the same physical characteristics, that's good enough.

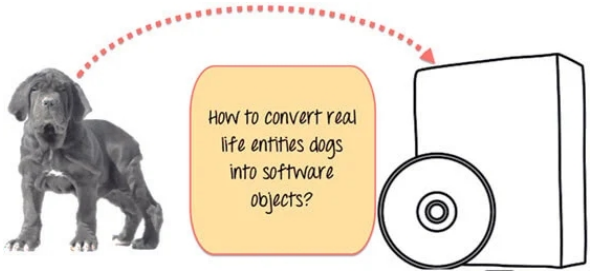

Figure 2: How to represent real life dogs by software objects? Image by James Hartman published on Guru99. © 2023 Guru99 Tech Pvt Ltd

In making such an app, we would need to identify the various objects from this description. These could be the dogs, the dog owners, the dog home and the player (perhaps some more too! Can you think of any?).

What could represent each of the objects? Let's start with dogs: we have a requirement to keep small and large dogs separately, so we must keep track of each dog's size. We need to know how often to walk the dogs, so we need the age and breed of each dog. Finally, we need the colour, breed, size, and possibly age (depending on what is meant by 'physical characteristics') to be able to pick a dog to return to the owners. So altogether, we have the properties: breed, size, age, and colour (apparently the name is not important in this evil evil dog home).

The generalisation of the properties that any dog has becomes the `Dog` class. Objects or "instances" of this class are particular dogs, e.g. a small, white, 2-year-old Maltese.

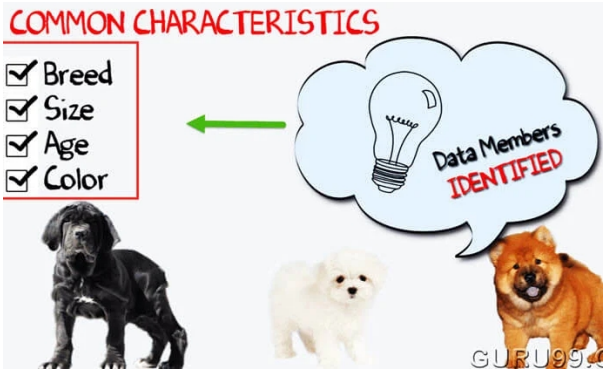

Figure 3: Dogs' common characteristics. Image by James Hartman published on Guru99. © 2023 Guru99 Tech Pvt Ltd

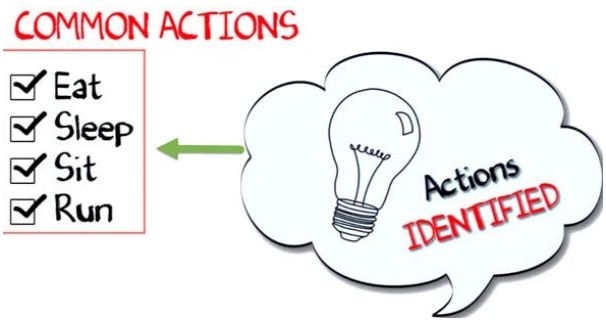

Figure 4: Dogs' common actions. Image by James Hartman published on Guru99. © 2023 Guru99 Tech Pvt Ltd

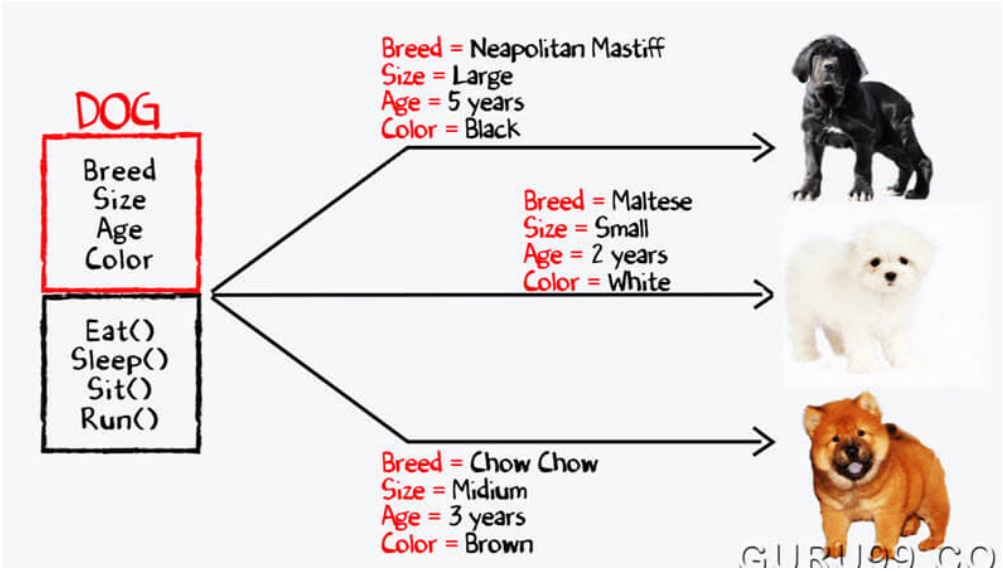

Figure 5: Resulting design. Image by James Hartman published on Guru99. © 2023 Guru99 Tech Pvt Ltd

Once we have decided on the classes and properties, we sometimes also have to think about how to represent the properties. Take the dog breed for example - what data type could represent a dog breed? There are at least 3 solutions: using a string, using an enumeration, and using subclasses for each breed.

Which is best? We just learned about classes, so should we have a subclass of `Dog` for every dog breed, e.g. `ChowChow extends Dog` ? This would work, but for this app, using subclasses would result in more code than necessary. Keeping the breed as a string in the generic `Dog` class would be much simpler. Without sub-classes, the project would also be easier to extend - when a new dog breed is introduced, we won't need to create a new class for it, simply creating a new instance of `Dog` with the appropriate string will be enough. Enumerations might work even better, but this will be a topic for another time.

Then there could be a `DogHome` class. This class would probably maintain a list of `Dog` objects and perhaps contain some methods for checking a dog in, when new owners come to leave one or finding a suitable dog when the owners come back. Similarly, we would need to design all the other classes too.

2 Exercises

For the exercises in this tutorial, let's get into groups of 4.

Task 1 - Model the universe

◁ **Task**

To start off with a warmup exercise, let's design a simulation of the entire universe (all in a day's work for our INF1B students!)

First, in your groups watch the "Cosmic Eye" on YouTube (https://www.youtube.com/watch?v=8Are9dDbW24). You don't need to take notes or remember what exactly is shown in the video, so relax and enjoy the show.

Afterwards, discuss with your teammates: what objects might there be in a simulation of the universe? What properties would they have?

You can draw pictures or diagrams to show your model. Summarise your design in the next 5 minutes as best you can, then swap with another team's design to compare and discuss - what is different? Is one design better than another?

There is no one solution to such an open-ended problem. Anything that shows the universe at different scales (e.g. Universe, Galactic System, Galaxy, Star System, Star, Planet, Continent, Ocean, Country, City, Insect, Cell, Electron) is correct. A possible fun implementation to explore is Orteil's "Nested" project - http://orteil.dashnet.org/nested

Task 2 - Navigating a zoological maze

◁ **Task**

You and your team have a new project at your hands!

Edinburgh city has an impressive city zoo. The Royal Zoological Society of Scotland (RZSS) want to provide visitors with a navigation program, that features an interactive map of the zoo with live information about events, like animal feeding or shows.

The previous developers recently left the society before the program was finished and you have been hired to take over the project. Before you get to bug fixing, refactoring or implementing the missing features, you might want to consider the design of the system first.

Do you think it is well designed? If so, what makes it a good design? If not, how would you design it? To evaluate this question, it might help to draw out a rough sketch of what classes there are and how they interact with each other.

RZSS gave you the following diagrams and description:

Figure 6: A section of the Edinburgh zoo map. Image from maps-edinburgh.com. © 2024 Newebcreations

The current zoo navigation program can maintain the state of the zoo and print out what is within it. The zoo itself is represented by an all-encompassing `ZooNavigator` class that keeps a list of objects in the zoo:

Some of these objects are instances of the `AnimalPen` class, some are instances of the `Museum` class, and some of the `Restaurant` class.

`AnimalPens` can contain one or two kinds of animals. The animal kinds can be seen on the map to the left. There are public feeding events for brilliant birds, monkeys, koalas and sun bears.

For now, the program just prints out a list of the things in the zoo and when feeding events take place.

✎ Download the **ZooNavigator.zip** from the Inf1B course materials page and open it in IntelliJ.

This project is not well designed because:

* The similar naming of `Animal`, `ZooAnimal`, `ZooAnimalClass` classes is super confusing. It can make sense to have a generic `Animal` enum if only the animals' names are important. Since we also care about their kind (e.g. monkey or brilliant bird), an enum is not the best choice. It would be better to have a parent `Animal` class that contains a name and have sub-classes `Monkey`, `BrilliantBird` and `OtherAnimal` that store the appropriate feeding time for the animal.

* The `ZooAnimal` interface is practically useless, it is only implemented by `ZooAnimalClass` and has no other purpose than to enforce a getter for name, which could simply be defined in a parent `Animal` class.

* The `ZooNavigator` class does not need a list of generic objects, which forces use of `instanceof` and makes the code very error-prone. Instead, it could keep lists of animal pens, museums and restaurants.

* The `Restaurant` and `Museum` classes are identical - they could be collapsed into one, such as `Building` (although the distinction might be useful if different behaviour was introduced later on, such as shows at museums, it still might be good to have a parent `Building` class to inherit from).

* The methods `isBrilliantBird` and `isMonkey` don't belong in `AnimalPen` - their behaviour is closely tied to the `Animal` enum. The way they are designed, comparing names as strings, also could become a source of bugs. Instead, animals could contain a field indicating its category, or, even better, use inheritance so this would not be needed at all.

* For the purposes of this project, it would suffice for each `AnimalPen` to contain a map of animal kinds and counts instead of a list of individual animals. This would simplify the other methods a lot.

Besides design mistakes, there are some bugs (e.g. feeding times are printed twice for animal pens containing two brilliant birds). There is also lots of refactoring to do - encapsulation is not followed at all, most things are declared public when they don't need to be. However, these are not the focus of the tutorial, part of the goal is to not get lost in the details, and instead think about the overall architecture.

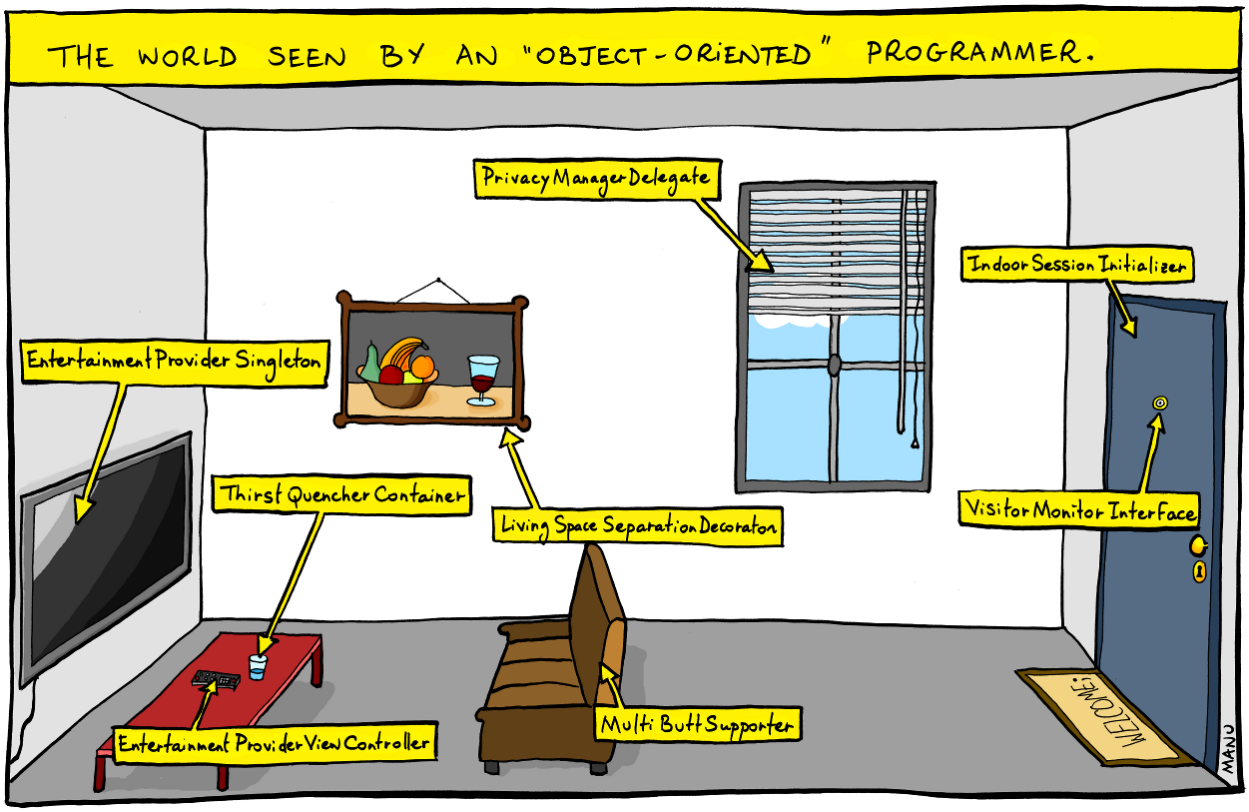

Figure 7: There are lots of keywords in this comic, like "Singleton" and "Initializer". They usually have special meanings in Object-Oriented Programming languages. "Object World" by Manu Cornet is available under the CC BY-NC-ND licence.

3 Further learning

In this tutorial we have only scratched the surface of what object design involves. There is a lot more to see about various design practices. Here are some places that can help you learn more about it:

The SOLID principles that were mentioned in the refactoring tutorial are like a standard ruleset for good class design. They are explained here, together with encapsulation, delegation, and other useful details: `https://www.javaguides.net/2018/08/object-oriented-design-principles-in-java.html`

This is an article on when to (not) use interfaces and abstract classes: `https://dzone.com/articles/when-to-use-abstract-class-and-intreface` (also see further reading links on this website)

Interfaces in more detail (e.g. why might a completely blank interface be useful): `https://beginnersbook.com/2013/05/java-interface/`

A topic that often comes up in Java design discussions is composition vs inheritance (should you keep a member class field or derive from a parent class). See for reasons why and when should you use which: `https://www.artima.com/designtechniques/compoinh.html`

The second year course "INF2-SEPP: Software Engineering and Professional Practice" will teach in-depth about designing a software system from a set of requirements, which is a lot like what we were doing today, but more focused on the architectural decisions.