1 UML and Object-Oriented Design

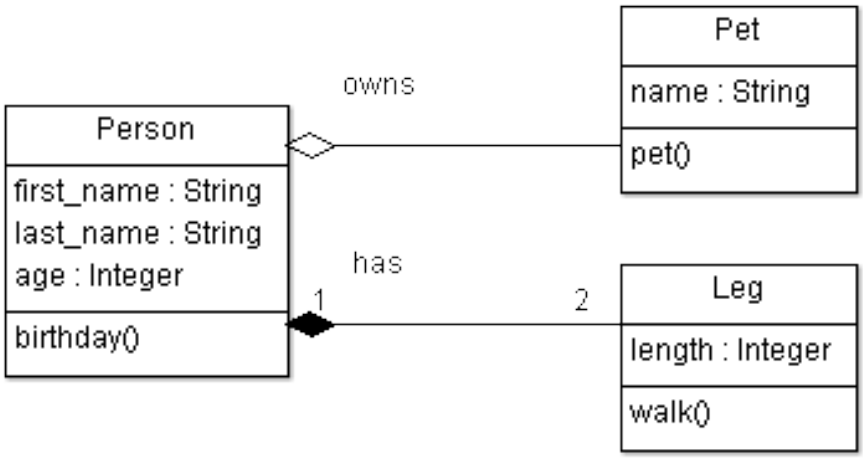
Object-oriented expand on the idea of using data structures like *struct* and *union* in order to organise logically connected data items. Instead of just associating values in logical groups, as done with *struct*s, object-oriented programming also associates operations to these data item groups. The paradigm requires that all data items modelling the same *object* need to be categorised in the same *class*. Operations manipulating these data items are also associated with the *class*. This concept is illustrated in the following example:

```
class Person {
private:
  std::string first_name;
  std::string last_name;
  unsigned int age;

public:
  void birthday() {
    ++age;
  }
};
```

We can see that all relevant information about a person is kept as a data member of the class. All operations on that persons information, such as what happens if that person celebrates a birthday, is kept in the class as well. Object-oriented design is a form of modularisation which tries to group algorithms and their relevant data items into logically cohesive *class*es.

UML is a modelling language for various software development tasks. It defines structure diagram, behaviour diagrams and interaction diagrams of large software systems. Probably the most important artefact in UML are class diagrams. They allow to model *class*es and their dependency on each other.



The diagram above illustrates relationships between a person, their pet and their body's legs. Each class in the diagram has a

set of data members and operations. The main relationships between classes are aggregation and composition. Aggregation symbolises a weak ownership. This is the case between people and their pets. This relationship is expected to change over the course of a program execution, and the existence of the owned object is not dependent on the existence of the owner object. In our diagram example this suggests that pets can be given away and that a pet continues to exist even its owner ceases to. This is not the case for the relationship between a person and their legs. In this case, we used the composition connector, indicating that from the perspective of our model, legs are an essential part of a person. We don't expect their ownership to change – current developments in medical science notwithstanding – and our model has no use for leg objects anymore once we remove the person object from our model.

Associations can also have a cardinality. Our example model suggests that one person has two legs, and that a leg belongs to exactly one person. It also states that a person owns exactly one pet, and a pet belongs to one person at a time.

Task 1 Create an employee management diagram

Using this knowledge, we create our first UML class diagram. It should model the following real-life objects:

- Employee
- Phone
- Role
- Task
- Customer
- CustomerFeedback

An employee should have a first and last name. Every employee has exactly one phone and a phone only belongs to one employee. The phone contains an extension number and gets discarded if the person leaves. Every employee is a member of a team, which is identified by its team name. A team may consist of multiple employees. Every employee may assume multiple roles in the company. A role is again identified by its name. A role contains multiple unique tasks, which only bear meaning in the context of their associated role. Tasks contain a name and an integer difficulty level.

Customers are associated with a customer id, as well as their first and last name. Customers can create CustomerFeedbacks, which are associated with exactly one customer and one employee. They contain a string comment and an integer rating. Draw a sketch of this UML model on paper.

Task 2 Use ArgoUML to create a detailed version of the diagram.

Task 3 Automatically generate C++ code from the ArgoUML diagram.

Compile the created code using the tools we used during this course. Create example objects of every class in your program.

Task 4 Design patterns

Design patterns are the basic building blocks of software architects. A design pattern is a documented way of solving a programming problem using the means of a given paradigm (procedural, object-oriented, functional). Design patterns are described by applicable usage scenarios, benefits and liabilities.

The most influential collection of design patterns for object-oriented design patterns are the Gang of Four (GoF) patterns and the General Responsibility Assignment Software Patterns (GRASP). Together these design patterns represent a useful toolkit for software engineers in many common programming dilemmas.

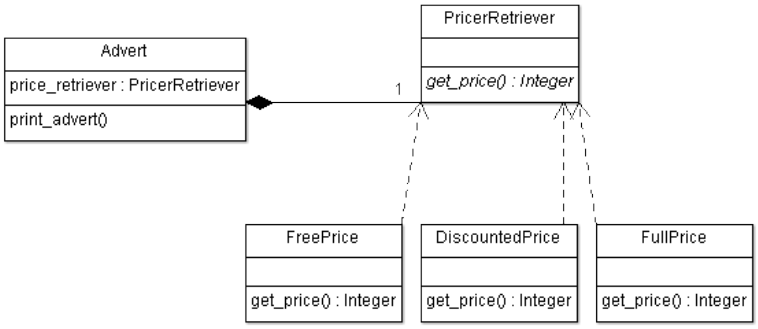
One simple example is the *Singleton* design pattern.

Task 4.1 Lookup the definition of the Singleton pattern.

Task 4.2 Implement the Singleton pattern.

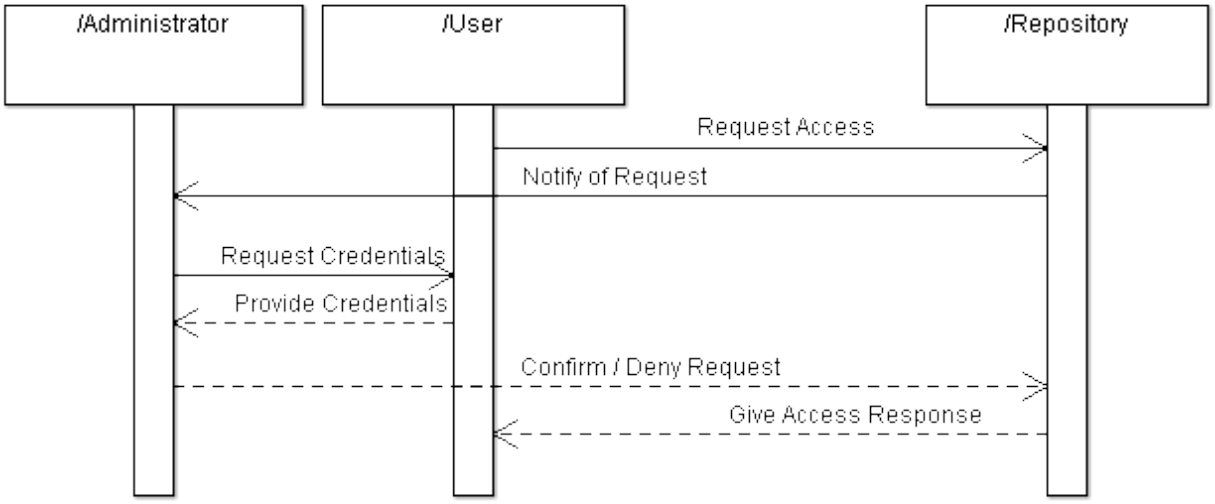
There should only be one prime-minister. Hence, a single instance of such a role should suffice. Open the `prime-minister.cpp` file and convert the class into a singleton.

Task 4.3 Lookup the definition of the Strategy pattern, and implement code according to the UML diagram



Task 5 Sequence Diagrams

UML also provides models to illustrate interactions between components. Sequence diagrams are very flexible examples of this category. They can model low-level interaction diagrams, e.g. between classes using method calls, or very abstract diagrams modelling interactions between components and actors in a system.



The above diagram illustrates a very simple sequence process. An actor called "User" requests access to a repository. The repository notifies an "Administrator" of the request, which verifies the users' credentials. He then either confirms or denies the request, which is forwarded by the repository.

Now create a sequence diagram for a challenge-response log-in system. The user interface initiates the login process by requesting a challenge from the application server. The application then queries the user to enter username and password.

The application server then queries the database server for the user's password hash. It calculates the combined response and either confirms or denies the login request. Denied or confirmed requests are logged into the database server. The application server also logs login failures and may deny a certain user access if too many failed attempts have been registered.

Task 6 Extra: Library database

Create a UML class diagram for a library management software. The software should be able to associate books in the library with their respective authors. There may be multiple copies of the same book in the library. Books are identified by title and ISBN. Author information to be stored is simply their name and their biography. User accounts are registered with specific branches in different cities of the same library company. Users have an ID, an opening date of their account and a status field, indicating whether their account is active, frozen or closed.

All books that users have borrowed or reserved, past or present, should be stored in the database. Users also have a contact person associated, who is a library employee and assists them with questions via phone. All book copies are listed in a library catalogue, which is printed every year and sent to the customers. The database should provide the information of which user has received which catalogues over the years.

Draw a sketch by hand on paper and then create the final version in ArgoUML. Add detailed methods and fields as you see fit to implement the described library system. Do not create methods in the UML classes, but instead use relations and dependencies to indicate aggregations, compositions and interactions.