# 6. Object Oriented Programming

**AlDaBi Praktikum**

Enrico Siragusa
WS 2011/12

# Summary

- Programming paradigms
- OOP in C++
- P-A5 in OOP
- Remarks for P-Aufgabe

Fundamental Programming Styles

# PROGRAMMING PARADIGMS

# Imperative Paradigm

- Computation in terms of statements changing the program state
  - Data constitutes the program state
- Programmers describe how to obtain a result by executing instructions
  - A program can be seen as a recipe
- Procedures are reusable blocks of statements
  - Procedures are also called subroutines or functions
- The result of a procedure depends on current program state
  - Problem: Program state is globally exposed
- Imperative languages abstract from machine language
  - Fortran, Pascal and C abstract from Assembly

# Declarative Paradigms

- Functional Programming Paradigm
  - Based on Lambda calculus
  - Computation as the evaluation of mathematical functions
  - Functions are stateless so their result only depends on input arguments
  - Iteration via recursion
  - Lisp, Haskell and Ocaml are functional languages

- Logical Programming Paradigm
  - Computation in terms of logical statements which have to be satisfied
  - Prolog and SQL are logical languages

# OOP Paradigm

- Computation in terms of interacting objects
  - Objects are physical or abstract entities with one precise role
  - Objects have a well defined behaviour
  - Objects hold private information
  - Objects interact with other objects exposing their functionalities
- Programmers design a set of objects modeling the problem at hand
  - Humans see the world as composed of objects
- Addresses software conception, mantainance and extensibility
  - An object is the smallest modular unity, extensible and reusable
- Modern programming languages support OOP
  - Java and C++ were explicitely designed for OOP

# Main Principles of OOP

- Abstraction
  - Functionality is provided via an interface as in abstract data types
  - Complexity of implementation is confined to the object
- Encapsulation
  - The internal state of an object is hidden to the outer world
  - An object can only be inspected or manipulated via its methods
  - Methods insure the integrity of the object's internal state
- Inheritance
  - Allows reutilization and extensibility of objects
- Polymorphism
  - Provides subtype specialization of objects

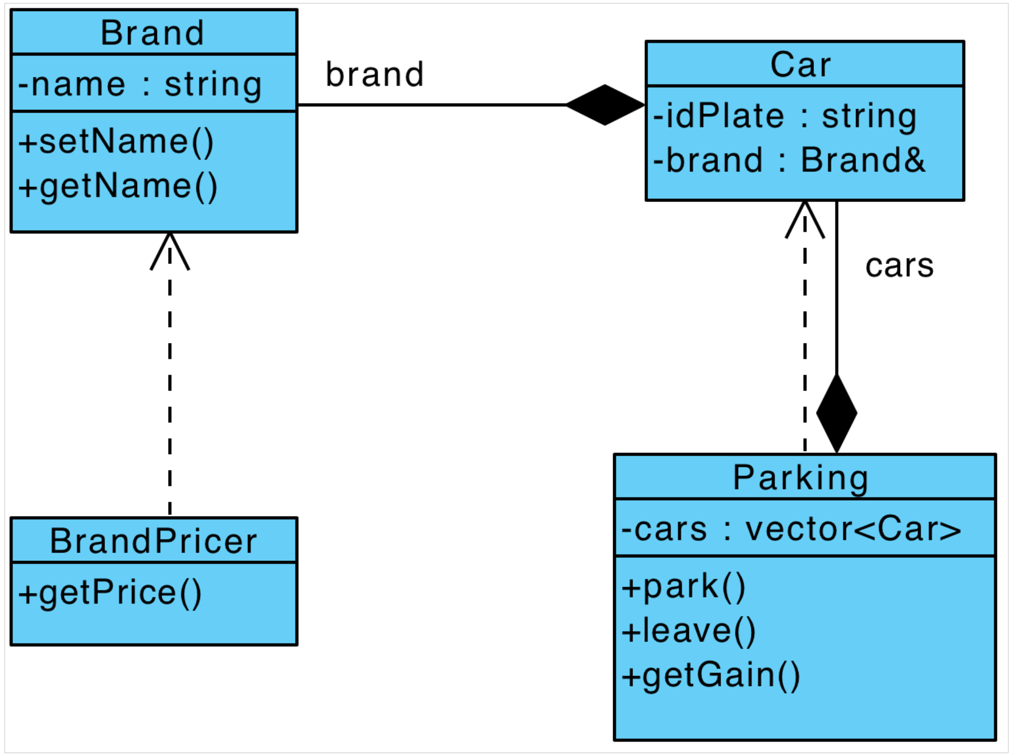
# Example of OOP Modeling

- Specifications
  - Program an application managing a parking
  - Compute at any time the total gain if all cars leave the parking
  - Parking fee only depends on car brand

- Responsibilities
  - Parking: contains vehicles
  - Car: the car itself
  - Brand: the car brand
  - BrandPricer: fixes the fee depending on the car brand

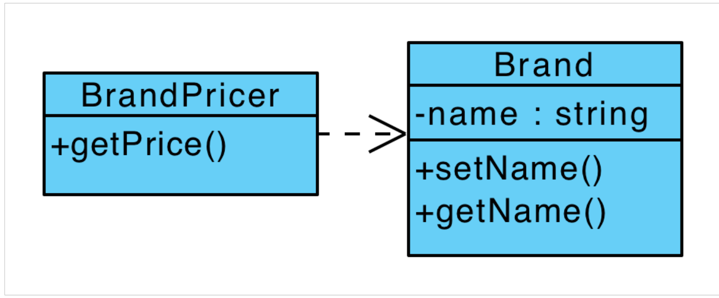
Example taken from lecture „La programmation objet “ of Rémi Forax, WS07, Univ. Paris-Est MLV

# Example of OOP Modeling (II)

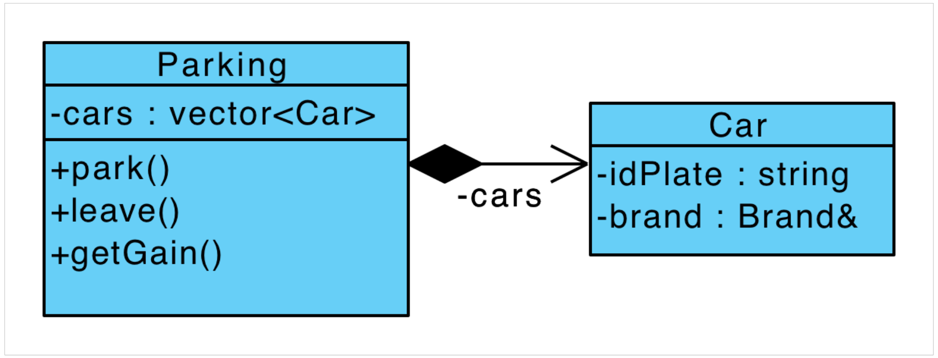


# Objects Relationships

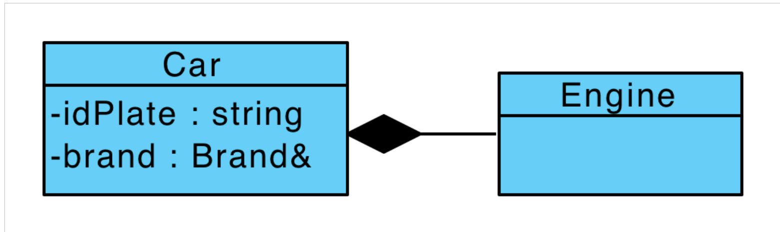
- Association
  - Generic relationship between two objects
  - Objects providing or using other objects




- Aggregation
  - „Has a“ relationship
  - Occurs in containers




- Composition
  - „Owns a“ relationship
  - Owned object does not exist outside of owner object



# OOP IN C++

# Classes and Objects

- A class defines the implementation of a set of objects
  - In C++ classes are implemented as data structures

```
class Car { };
```

```
struct Car { };
```

  - Classes have default private visibility (see next slide)

- An object is an instance of a class
  - a, b, c  are all instances of class Car

```
void main() {
  Car a;
  Car b;
  Car c;
}
```

# Members and Visibility

- Properties of an object are held inside class memberds
- Visibility of class members can be limited
  - Keyword private limits visibility to the class
  - Keyword protected limits visibility to subclasses

```
class Car {
private:
  string idPlate;
protected:
  unsigned seats;
public:
  string brandName;
};
```

```
void main() {
  Car c;
  // OK
  c.BrandName = "BMW";
  // Compile Error
  c.seats = 5;
  // Compile Error
  cout << c.idPlate;
}
```

- Visibility keyword applies also to methods (see next slide)

# Methods and Accessors

- Methods are functions having an implicit argument called this
- The keyword this provides a pointer to the owner object

```
class Car {
private:
  string idPlate;
public:
  string getIdPlate() {
    return this->idPlate;
  }
  bool setIdPlate(string &idPlate) {
    if (!idPlate.empty())
      return false;
    this->idPlate = idPlate;
    return true;
  }
};
```

```
int main() {
  Car c;
  // Returns false
  c.setIdPlate("");
  // Returns true
  c.setIdPlate("B ER 5");
  // Prints B ER 5
  cout << c.getIdPlate();
}
```

- Methods perform simple operations on the object, no monster code here!

# Static Methods and Members

- Stateless methods and perstistent members can be declared static
  - Static methods can be called at any time without object instantiation
  - Static members exist prior to object instantiation
  - Static members are shared by all object instances!

```
struct Class {
  static bool state;
  static bool getState()
    { return state }
  static void setState(bool state)
    { Class::state = state }
};

bool Class::state = false;
```

```
void main() {
  // Returns false
  Class::getState();
  Class::setState(true);
  // Returns true
  Class::getState();
  Class c;
  // Returns true
  c.getState();
}
```

- Take care while using static keyword!

# Method Overloading

- Methods (and functions) can be overloaded
  - Two functions can have the same name but different signatures
  - The compiler chooses the most adherent signature
  - Overloading is not performed on return value!

```
struct Class {
  static void m(int a)  { cout << "1" }
  static bool m(char a) { cout << "2" }
  static void m(double a, double b)
                        { cout << "3" }
};
```

```
void main() {
  // Prints 1
  Class::m(5);
  // Prints 2
  Class::m((char)5);
  // Prints 3
  Class::m(3.1, 2);
}
```

- Overload methods only if they share a common semantic
- Operators are implemented as methods and can be overloaded as well

# Constructors and Destructor

- Initial object state is set up by special methods called constructors
- Default empty constructor can be overloaded
- Eventual deallocation of any internal resources is done by the destructor

```
class Car {
private:
  string idPlate;
public:
  // Default constructor
  Car() {}
  // Custom constructor
  Car(string & idPlate)
    : idPlate(idPlate) {}
  // Default destructor
  ~Car() {}
};
```

```
void main() {
  Car c("B ER 5");
}
```

# Inheritance

- Inheritance consists of three concepts
  - Structural inheritance of methods and members
  - Subtyping
  - Method overloading

```
struct Car {
  void refill() {}
  void drive() {}
};
struct ElectricCar : Car {}
```

```
void main() {
  ElectricCar e;
  e.drive();
}
```

- The derived class ElectricCar
  - Inherits methods refill and drive from Car
  - Is a subtype of class Car
  - Has the ability to specialize and extend class Car

# Subtyping

- Problem: Can we park ElectricCar cars in a Parking for Car cars?
  – Yes we can! ☺

```
struct Parking {
  vector<Car> cars;

  Parking(unsigned places) {
    cars.resize(places);
  }
  void park(unsigned place, Car &car) {
    cars[place] = car;
  }
  Car & leave(unsigned place) {
    return cars[place];
  }
};
```

```
void main() {
  Parking p(2);

  Car c;
  p.park(0, c);

  ElectricCar e;
  p.park(1, e);

  Car & b = p.leave(1);
  b.drive();
}
```

Note: due to space constraints, class Parking is not implemented as it should be!

# The Delegation Problem

- Problem: Consider the following code:

```
struct Car {
  void refill() { cout << "Gas Please"; }

  void drive() {
    if (tank.empty())
      this->refill();
    ...
  }
};

struct ElectricCar: Car {
  void refill() { cout << "Energy Please"; }
};
```

- Class ElectricCar should overload method refill in order to specialize it

# The Delegation Problem (II)

- What happens here?

```
ElectricCar e;
e.drive();
```

- Outcome: Gas Please (!)
- Why?
  - Method drive is defined in the base class Car
  - Car does not know the derived class ElectricCar

```
void drive() {
  ...
  // this refers to a pointer of type Car
  this->refill();
  ...
}
```

# Virtual Methods

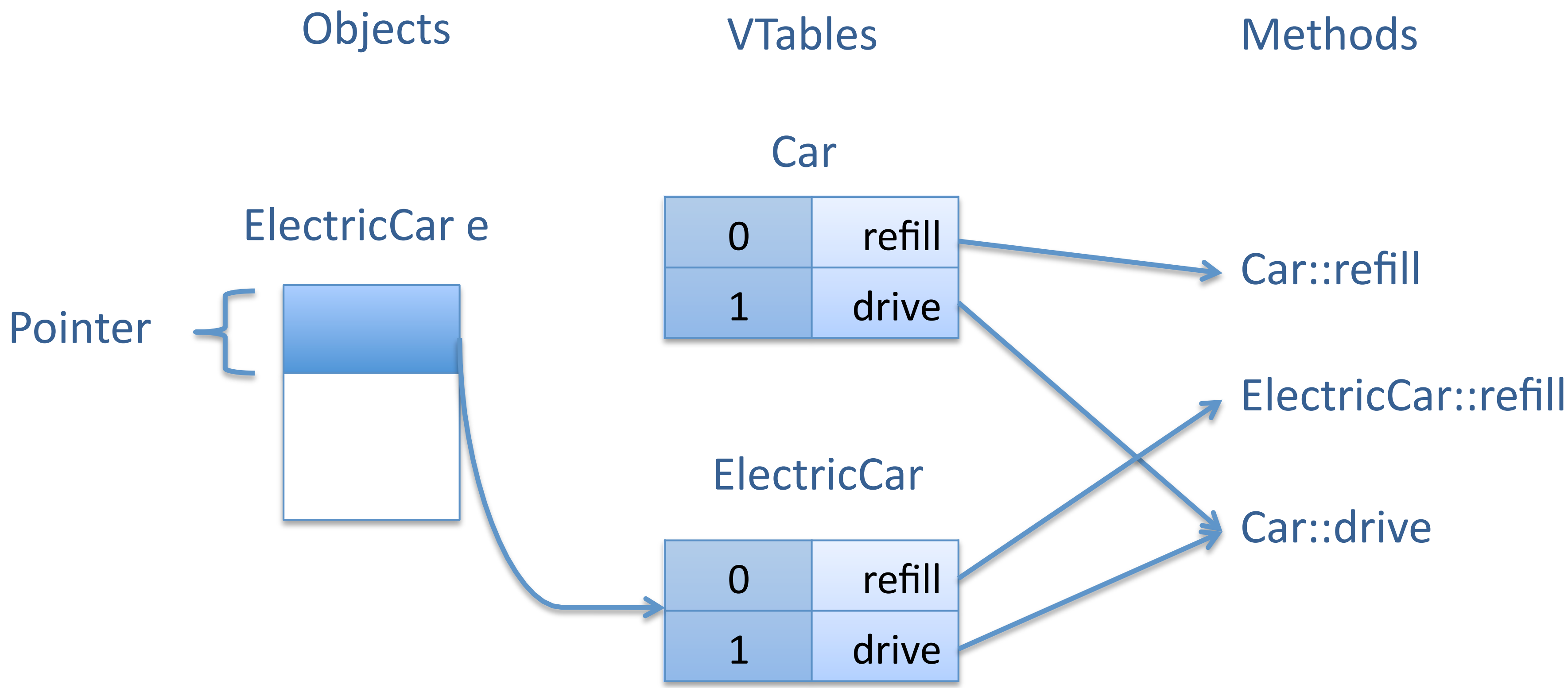
- Solution: **`virtual`**

```
struct Car {
  virtual void refill() { ... }
};

...

ElectricCar e;
// Prints “Energy Please“
e.drive();
```

- Static methods cannot be declared virtual
- Such behavior of objects is called polymorphism
  - Etymology from Ancient Greek poly (many) + morph (form) + -ism.

# How `virtual` works?

- Principle: Objects hold a pointer to a virtual table which has pointers to the overloaded methods



# Abstract Classes

- Classes only having virtual methods are called abstract
  - They serve as interface and base type for different concrete classes
  - They cannot be instantiated

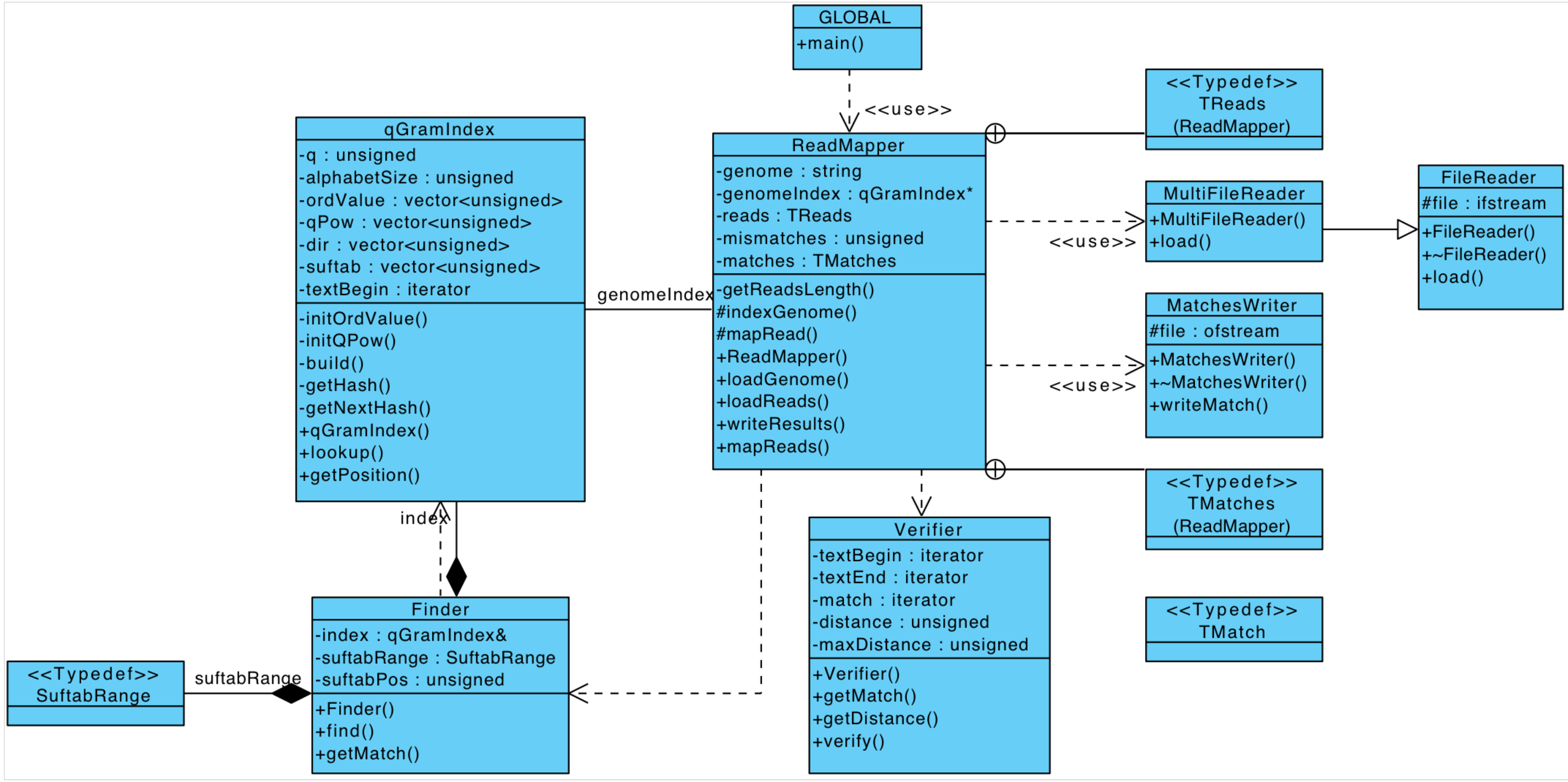
```
struct Abstract {
  // 0 or NULL indicate a null pointer
  virtual void method() = 0;
};

struct Derived : Abstract {
  void method() { /* Implemented */ }
};

// Compile Error
Abstract a;
// OK
Derived d;
```

# P-A5 IN OOP

# Entities and Responsibilities

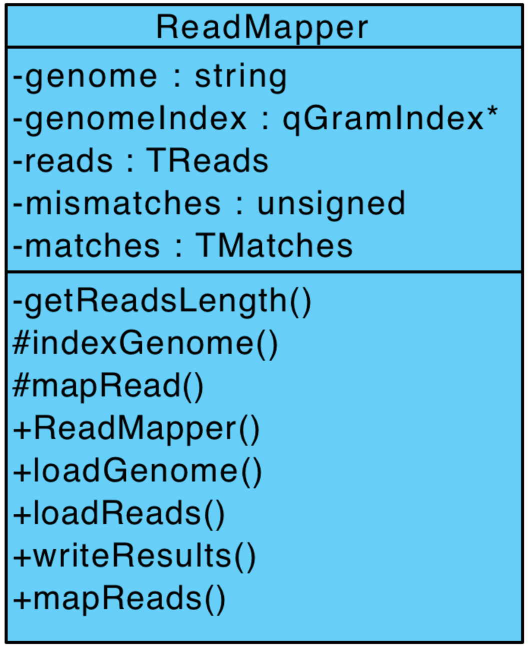
- ReadMapper
  - Maps reads sequentially
- qGramIndex
  - Indexes the genome
- Finder
  - Finds pieces in the genome
- Verifier
  - Verifies hits
- FileReader and MultiFileReader
  - Read input files
- MatchesWriter
  - Writes results

# Class Diagram



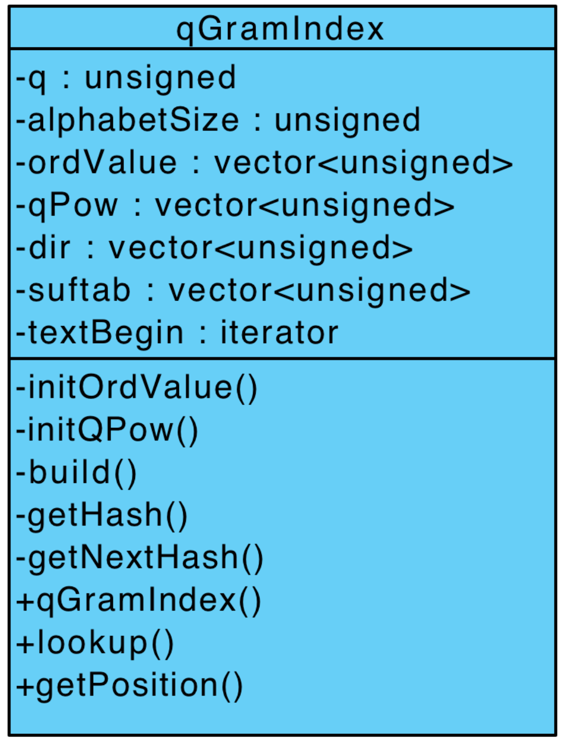
# ReadMapper Class

- Members
  - Genome
  - Genome Index
  - Reads
  - Matches
- Methods
  - Load genome using FileReader
  - Load reads using MultiFileReader
  - Index genome using qGramIndex
  - Map reads using Finder and Verifier
  - Write results using MatchesWriter



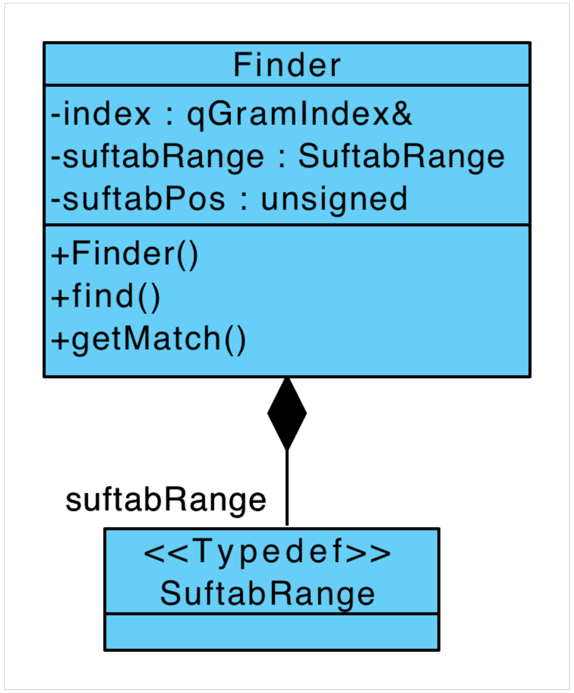
# qGramIndex Class

- Members
  - Values of q, alphabet size
  - Tables dir and suftab
  - Precomputed values for:
    - Ordinal value
    - Powers of q
- Methods
  - Constructor building the index
  - Lookup a q-gram
  - Getter for text position from suftab position
  - All other methods are private!



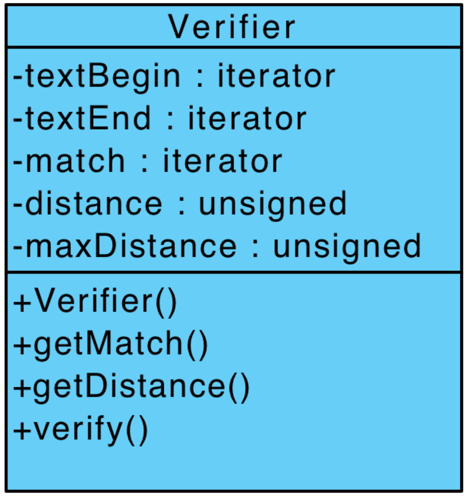
# Finder Class

- Members
  - qGram Index
  - Current suftab range
  - Current position in suftab
- Methods
  - Find a pattern
  - Get a match for the found pattern



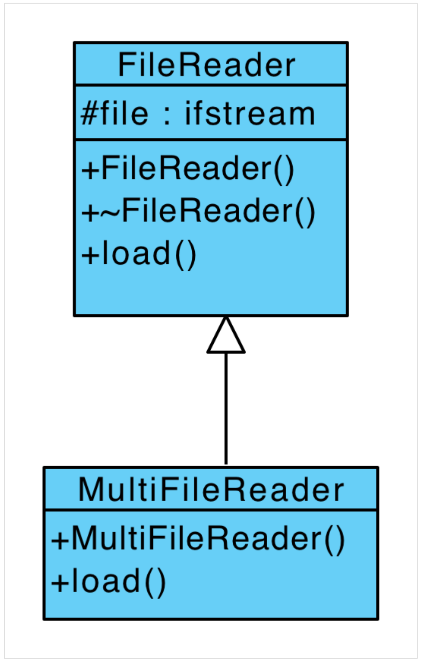
# Verifier Class

- Members
  - Text boundaries
  - Maximum distance
  - Current distance
  - Current match position
- Methods
  - Verify a hit stopping as soon as possible
  - Getters for
    - Current distance
    - Current match position



# FileReader Classes

- Members
  - Input stream is protected
- Methods
  - Constructor takes file name
  - Load loads the file
  - MultiFileReader specializes file loading



# REMARKS FOR P-AUFGABE

# Tips for Aufgabe 6

- A DFA *A* is a 5-tuple ($Q$, $\Sigma$, δ, $q_0$, $F$)
  - Number of states $|Q|$
    - 9
  - Initial state $q_0$
    - 0
  - Final states $F$
    - 8
  - Alphabet symbols $\Sigma$
    - a e h r t v w
    - Other ASCII symbols reset the automata into state $q_0$
  - Transition function $\delta : Q \times \Sigma \rightarrow Q$
    - Row i defines all explicit transitions for state i

- Automaton matching „whatever“

9
0
8
aehrtvw
0000001
0020000
3000000
0000400
0500000
0000060
0700000
0008000
0001000

# Class Diagram for Aufgabe 6