Embedded Systems Development - 1. Introduction

Electrical Engineering / Embedded Systems
School of Engineering and Automotive

Johan.Korten@han.nl

V1.1 2021

HAN_UNIVERSITY OF APPLIED SCIENCES

Overview

Theory (Wednesday)

Practical / Lab (Thursday)

To begin with…

Some books…

The C++ Programming Language
Fourth Edition
C++11
Bjarne Stro
Debugging Embedded and Real Time Systems
The art, science, technology, and tools of real-time system debugging
Arnold S. Berger
Real-Time C++
Efficient Object-Oriented and Template Microcontroller Programming
Second Edition
Christopher Kormanyos
Springer
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
UML Distilled
Third Edition
A Brief Guide to the Standard Object Modeling Language
Martin Fowler
Real Time UML
Third Edition
Advances in the UML for Real-Time Systems
Bruce Powel Douglass
Foreword by Professor David Harel
Real-Time Design Patterns
Robust Scalable Architecture for Real-Time Systems
Bruce Powel Douglass
Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin
Clean Architecture
A Craftsman's Guide to Software Structure and Design
Robert C. Martin
The Clean Coder
A Code of Conduct for Professional Programmers
Robert C. Martin

To begin with…

Today:

- Context: Methodical approaches to Systems and Software Engineering
- Good software practices 1.

Schedule (exact info see #OO and roster at insite.han.nl)

		C++	UML
Step 1	**Single responsibility**	**Scope, namespaces, string**	**class diagram**
Step 2	Open-Closed Principle	Constructors, iterators, lamdas	
Step 3	Liskov Substitution Principle	lists, inline functions, default params	
Step 4	Interface Segregation Principle	interfaces and abstract classes	
Step 5	Dependency Inversion Principle	threads, callback	
Step 6	Coupling and cohesion	polymorphism	
Step 7	n/a	n/a	n/a

Note: subject to changes as we go…

Software Engineering Methods

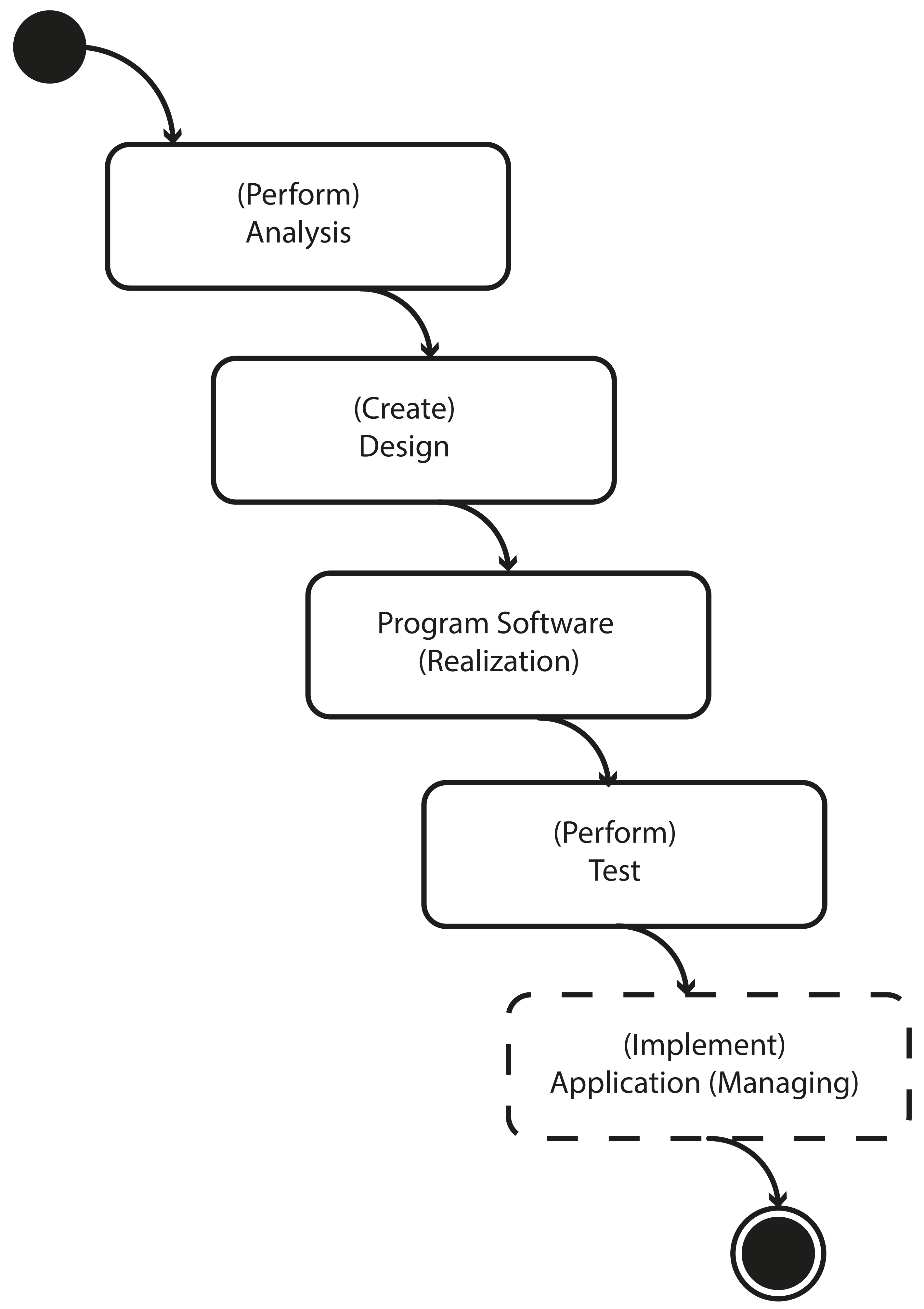
What are the basic ingredients to successfully perform a software project?

Software Engineering Methods

- Analysis
- Functional Design
- Technical Design
- Realization (Software Implementation)
- Testing

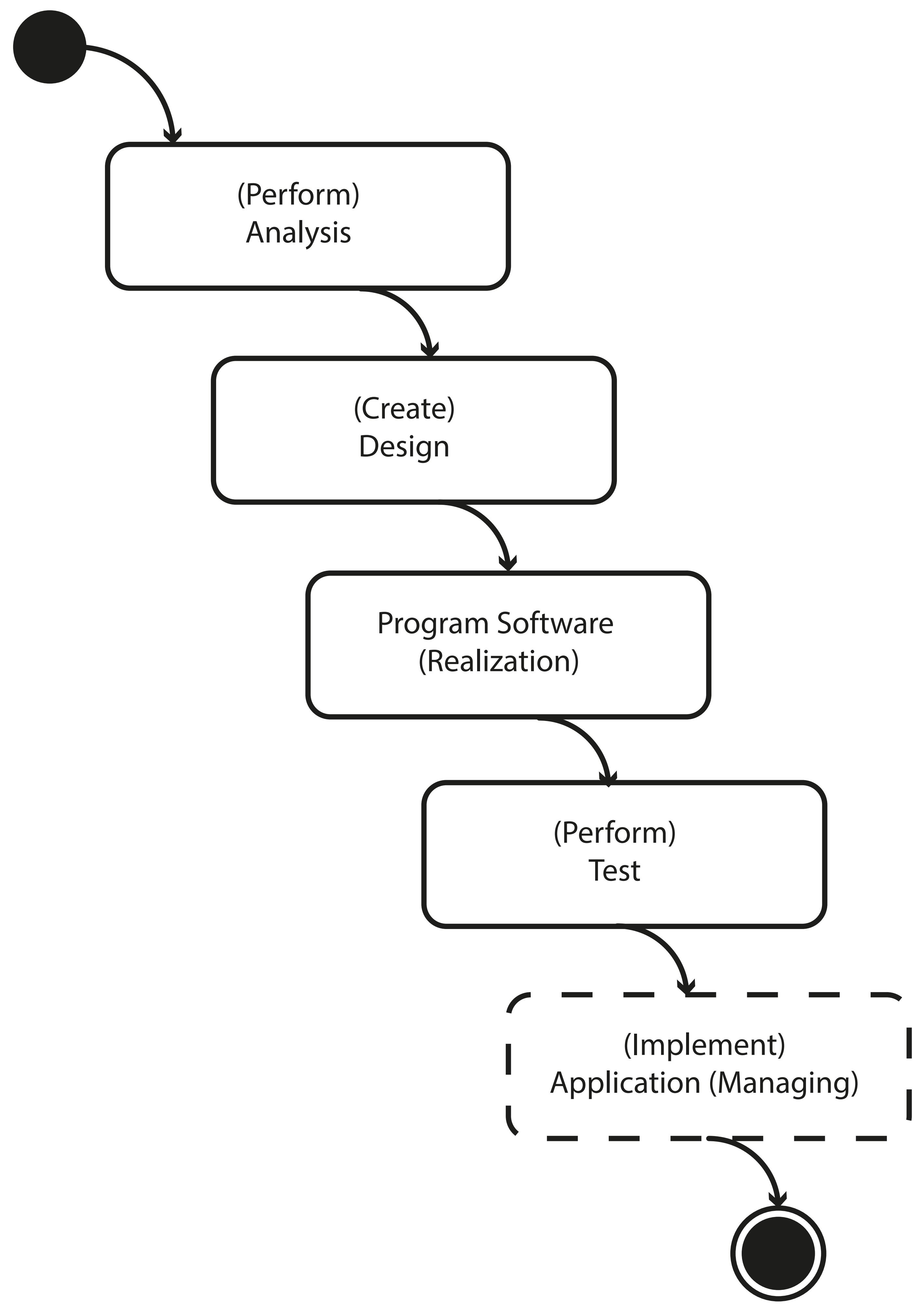
(Implementation (put in operation))

Software Engineering Methods: Waterfall


(Perform) Analysis
(Create) Design
Program Software (Realization)
(Perform) Test
(Implement) Application (Managing)

HAN_UNIVERSITY OF APPLIED SCIENCES

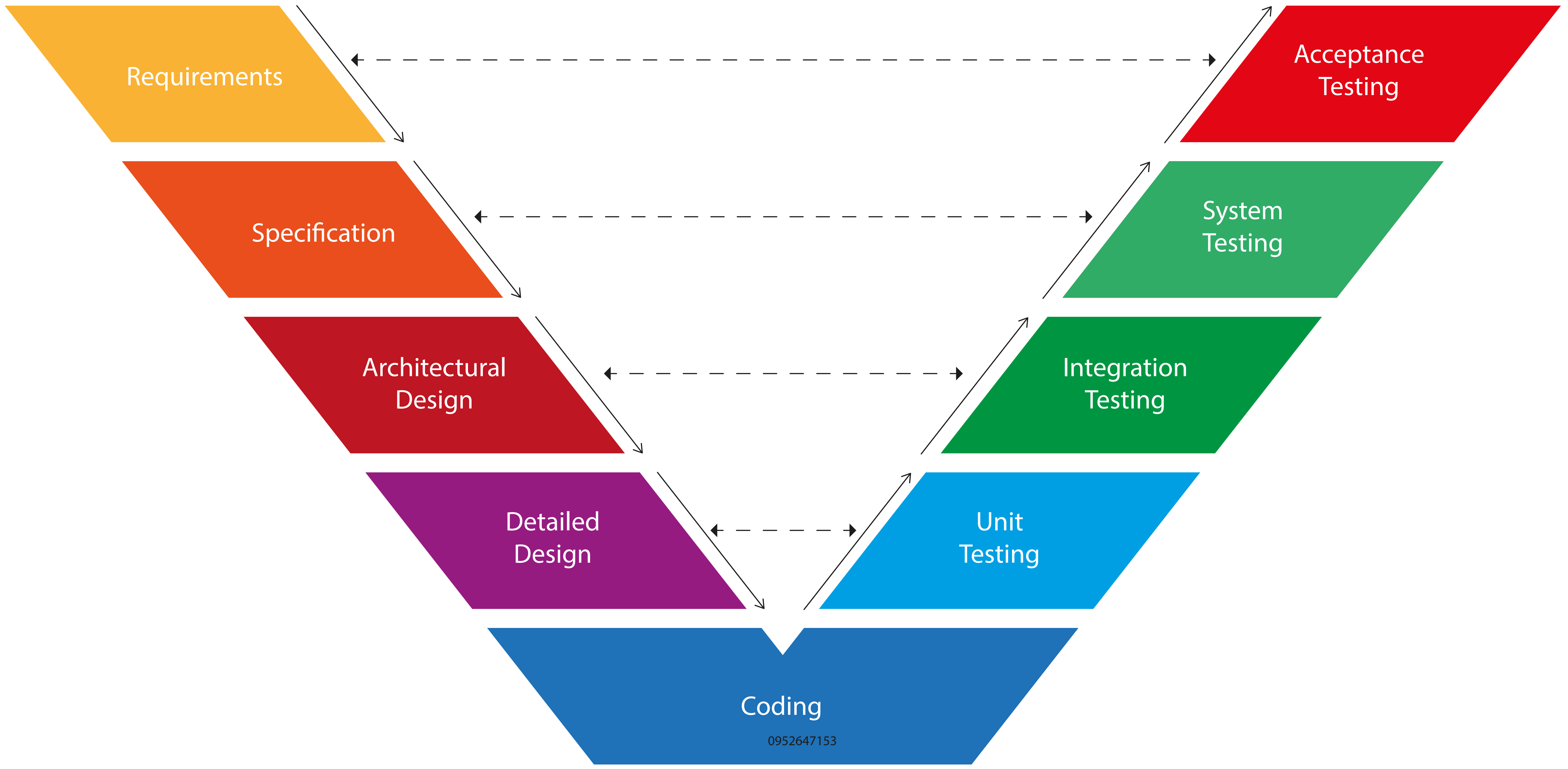
Software Engineering Methods: Waterfall



B.t.w.: what kind of UML diagram did we use?

HAN_UNIVERSITY OF APPLIED SCIENCES

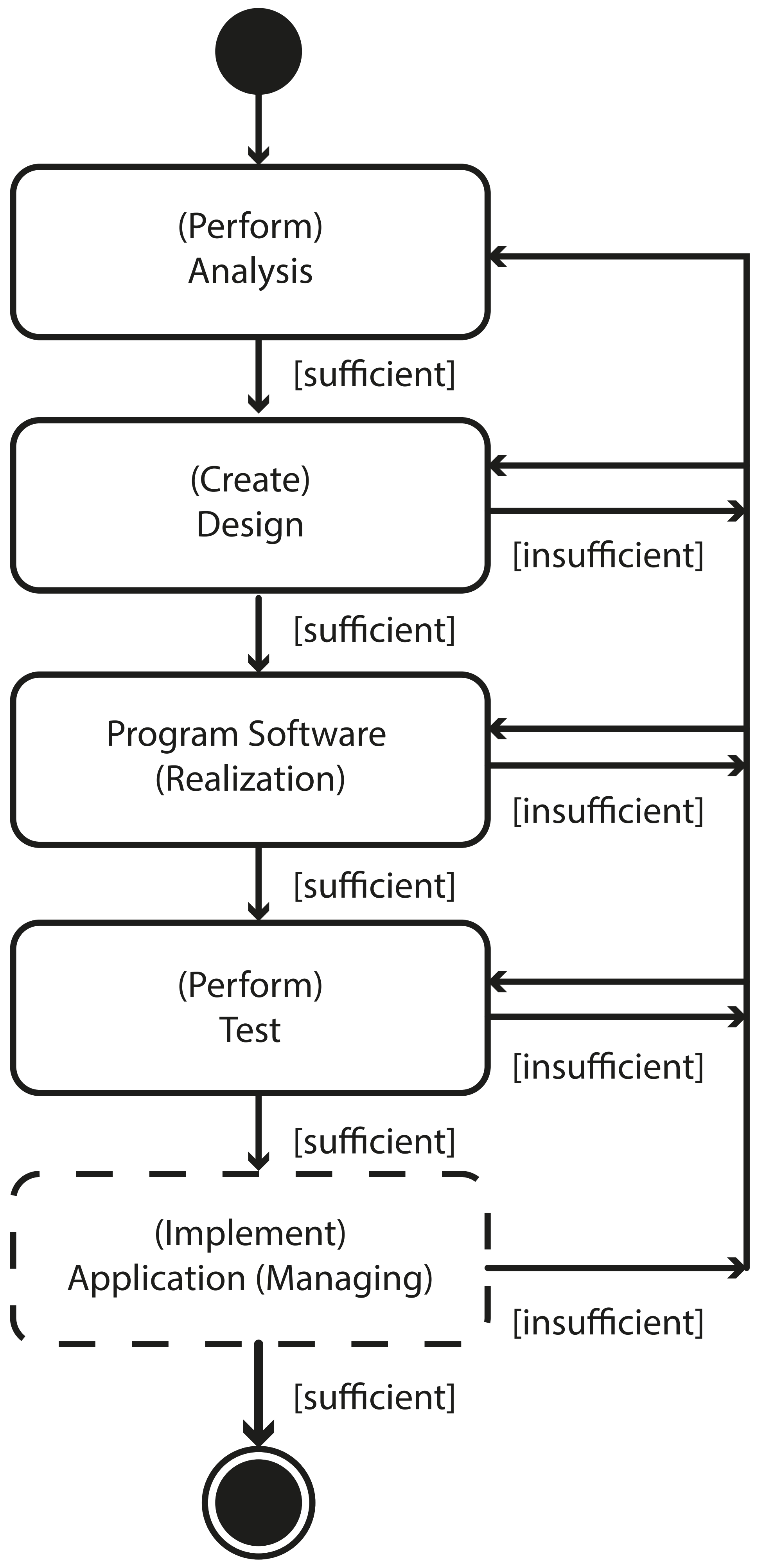
Software Engineering Methods: V-Model



Precisely gives you what you should do…

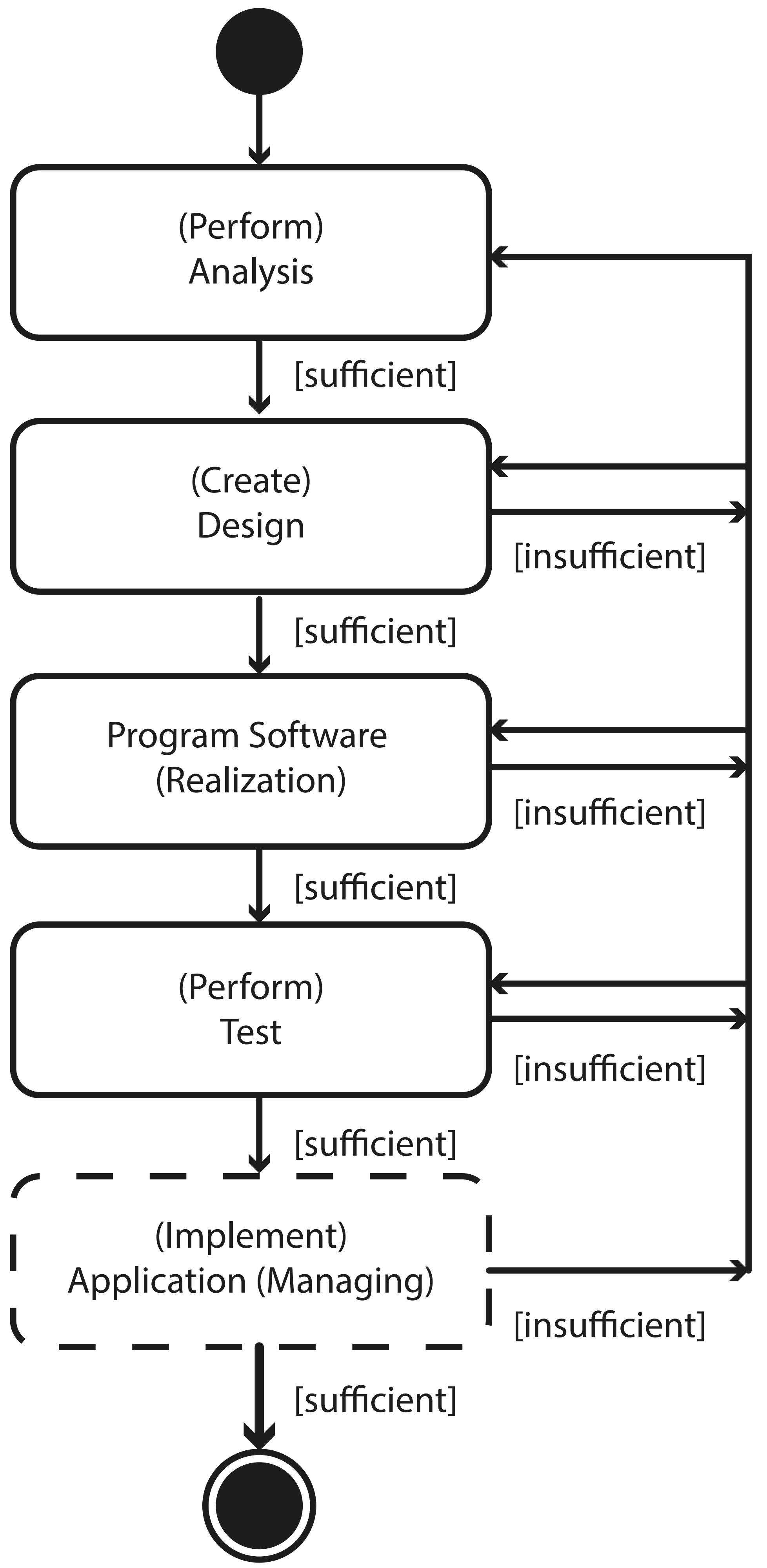
HAN_UNIVERSITY OF APPLIED SCIENCES

Software Engineering Methods: Iteratief



HAN_UNIVERSITY
OF APPLIED SCIENCES

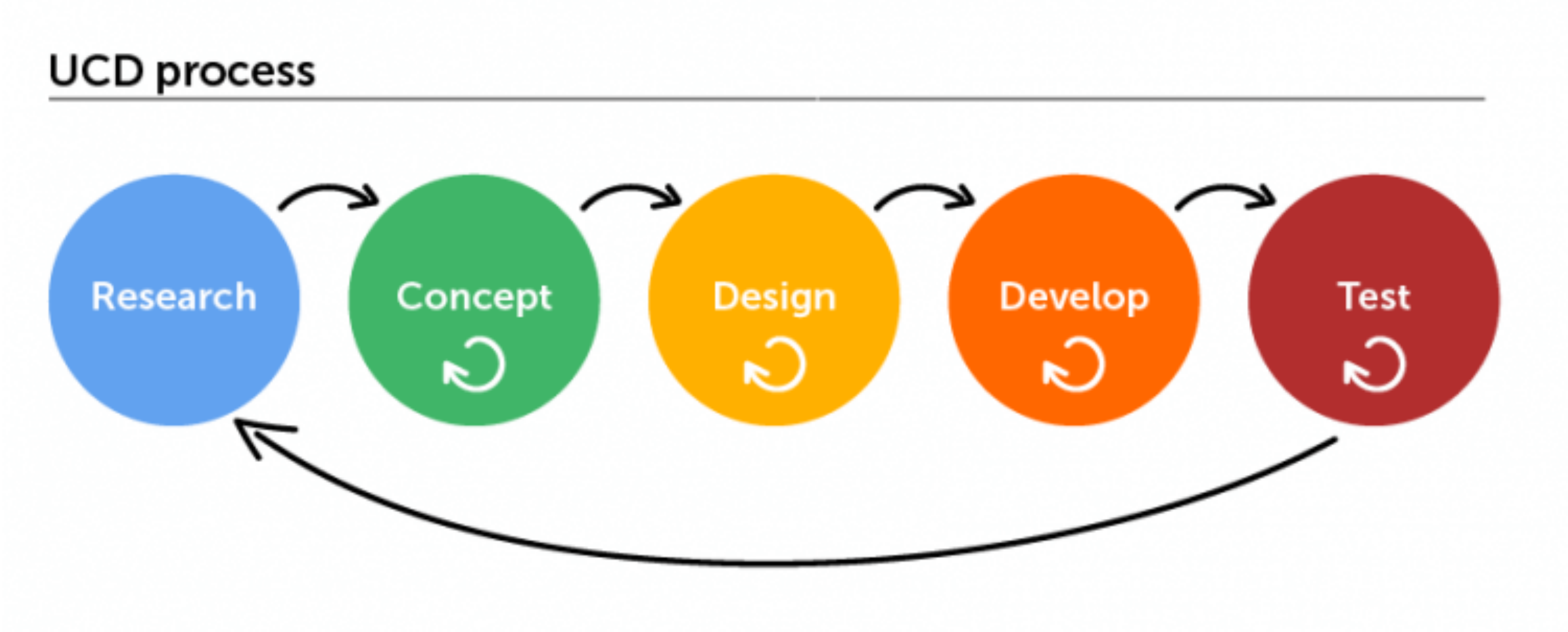
Software Engineering Methods: Waterfall



B.t.w.: what kind of UML diagram did we use?

HAN_UNIVERSITY OF APPLIED SCIENCES

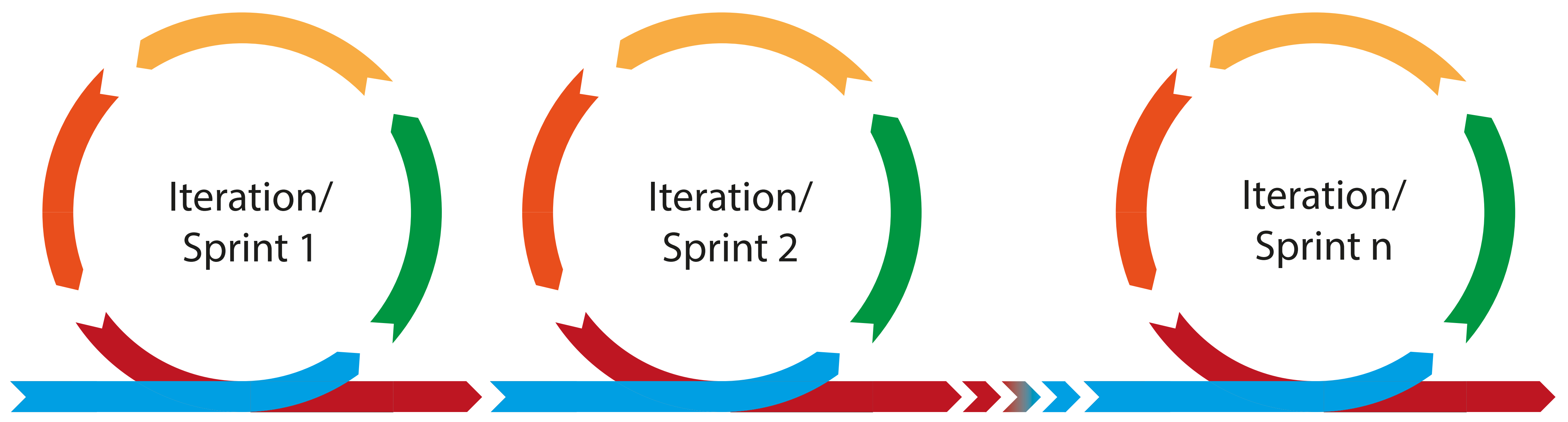
Software Engineering Methods: User-Centered Design



HAN_UNIVERSITY OF APPLIED SCIENCES

Software Engineering Methods: Sprints

Iterations / Sprints



- Analysis
- Design
- Realization
- Evaluation
- Implementation

Software Engineering Methods

Different ways to follow these models:

- one single time (waterfall)
- several times (iterative)
- very often (sprints) (e.g., every two weeks)

Very clear goals, set beginning and end, no trouble expected? Waterfall

A lot of uncertainty, agile (scrum, extreme programming, test-driven development, etc.)

Somewhat in between? E.g. RUP (basic iterative)

Software Engineering Methods: How to choose approach

https://www.capgemini.com/nl-nl/wp-content/uploads/sites/7/2017/07/Whitepaper_Keuze_Ontwikkelmethode_0.pdf (alas in Dutch)

- What you do is important.
- Why you do what is even more important for HBO-level competency: justify your choices…

Software Engineering Methods: Steps

- Analysis
- Functional Design
- Technical Design
- Realization (Software Implementation)
- Testing
- Implementation (put in operation)

Software Engineering Methods: Steps

- **Analysis:**
  - Make sure you get your assignment / goals clear
  - Architecture pictures might be useful
- Functional Design
- Technical Design
- Realization (Software Implementation)
- Testing
- Implementation (put in operation)

Note: for each step, certain cards of the Engineering / Automotive Methods pack might be useful:

https://ese.han.nl/mediawiki/index.php/Methods

Software Engineering Methods: Steps

- Analysis:

- Functional Design

- Functional and non-functional requirements (e.g., FURPS, prioritize using MoSCoW)
- Note: V-Model and original Waterfall even start with Requirements!
- Helps to create an overview (UML use case diagrams and use case descriptions)
- Should be understandable for non-Engineers (e.g., you manager)

- Technical Design
- Realization (Software Implementation)
- Testing
- Implementation (put in operation)

Note: for each step, certain cards of the Engineering / Automotive Methods pack might be useful:

https://ese.han.nl/mediawiki/index.php/Methods

Software Engineering Methods: Steps

- Analysis:
- Functional Design
- **Technical Design**
    - Specifies how you will implement your software
    - Numerous diagrams (e.g., class diagrams, flows, collaboration, states, etc.): goal think first, then implement, also useful for other programmers (if you hand over your project).
- Realization (Software Implementation)
- Testing
- Implementation (put in operation)

Note: for each step, certain cards of the Engineering / Automotive Methods pack might be useful:

https://ese.han.nl/mediawiki/index.php/Methods

HAN_UNIVERSITY OF APPLIED SCIENCES

Software Engineering Methods: Steps

- Analysis:
- Functional Design
- Technical Design
- **Realization** (Software Implementation)
    - That is what this subject is all about
- **Testing**
    - **That is what it is also about…**
- Implementation (put in operation)
    - That is what we often ignore (e.g., software aging etc, it is a challenge!)

Note: for each step, certain cards of the Engineering / Automotive Methods pack might be useful: https://ese.han.nl/mediawiki/index.php/Methods

HAN_UNIVERSITY OF APPLIED SCIENCES

Software Engineering Methods: Some other considerations

When you work agile, you don’t make the FD/TD to start with, but it will grow as you go.

When you perform the rapid prototyping ‘anti-pattern’, make sure that you define it like that, maybe your FD/TD steps are quick and dirty in that case, but give it at least some consideration…

Software Engineering Methods: Some other considerations

There are many additional tools, pick/choose them wisely.

- Scrumboard / Kanban board (To Do, In Progress, Done) (Trello)
- Agile methods got hijacked by non-Engineers, see Uncle Bob (one of the writers of the “Agile manifesto”)
- It is wise to define quality: Definition of Done, etc., especially if you work in sprints
- Testing is key, Uncle Bob advocates:
    - Test Driven Development
    - Pair programming

There are software tools to help you:

Git, Jenkins, Trello, etc. etc.

Advanced tools for Continuous Integration/Deployment (CI/CD)

Towards Craftsmanship

- UML is not an annoying extra: it is a way to keep your thinking structured and document your software in a visual way also for non-programmers.

- Even though Design Patterns are sometimes abstract really abstract (and initially maybe somewhat confusing) they help you to keep your software structured.

- Uncle Bob has excellent views on Software Craftsmanship so please listen to his advices.

Next period we focus on Clean Code / Clean Architecture with Uncle Bob.

Programming Paradigms

- structured programming
- object-oriented programming
- functional programming (next week -> lambdas)

There are others including:

- protocol oriented programming (e.g. Swift)

Structured programming: Dijkstra (CWI)

- structured use of if/then/else and while
- functional decomposition (for use this can be done at a software and a hardware level!)
- create code that is testable and provable by splitting it into falsifiable units

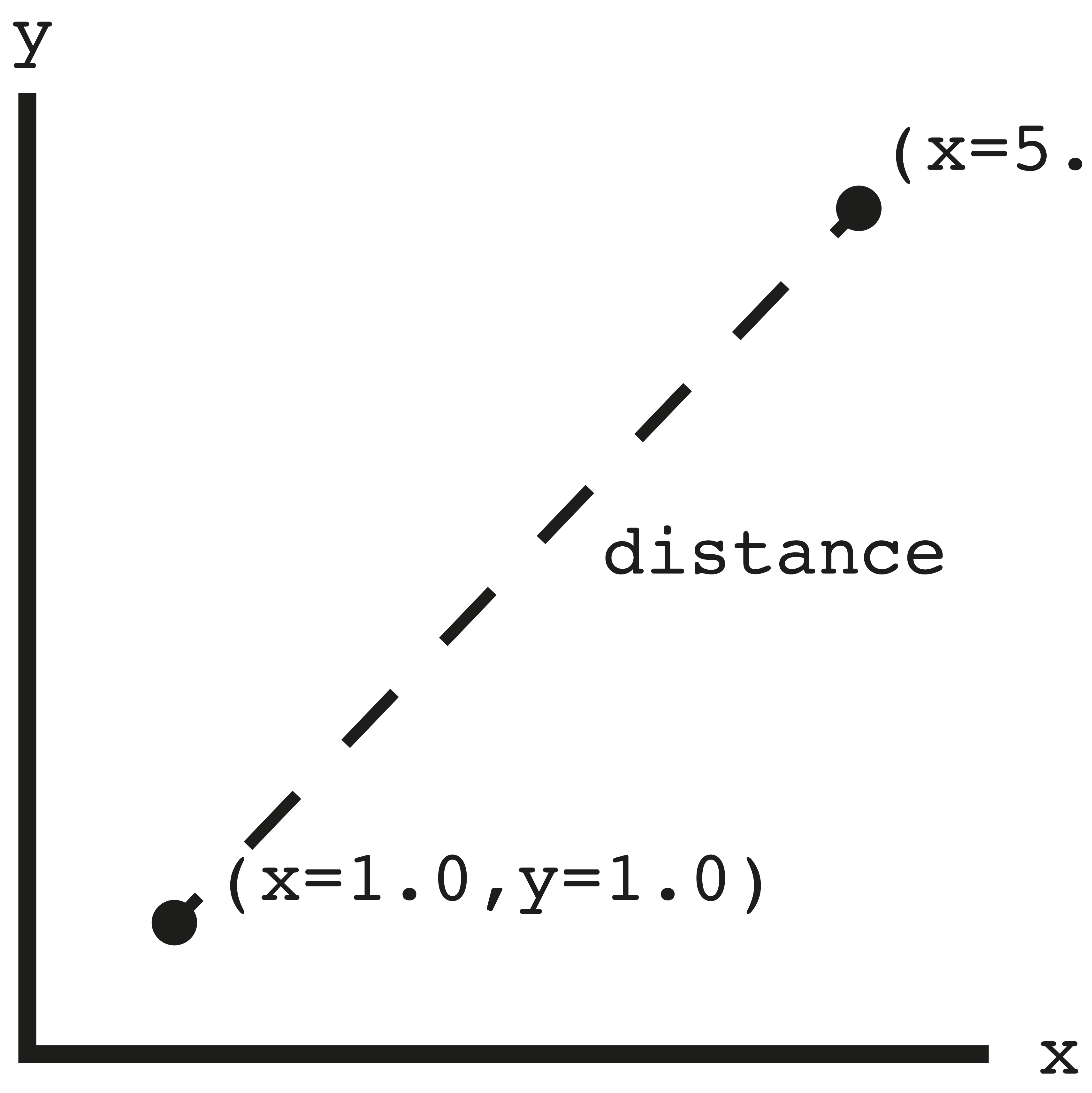
Remark from Dijkstra: “Testing shows the presence, not the absence of bugs”

Object-Oriented programming

- encapsulation
- inheritance
- polymorphism

HAN_UNIVERSITY
OF APPLIED SCIENCES

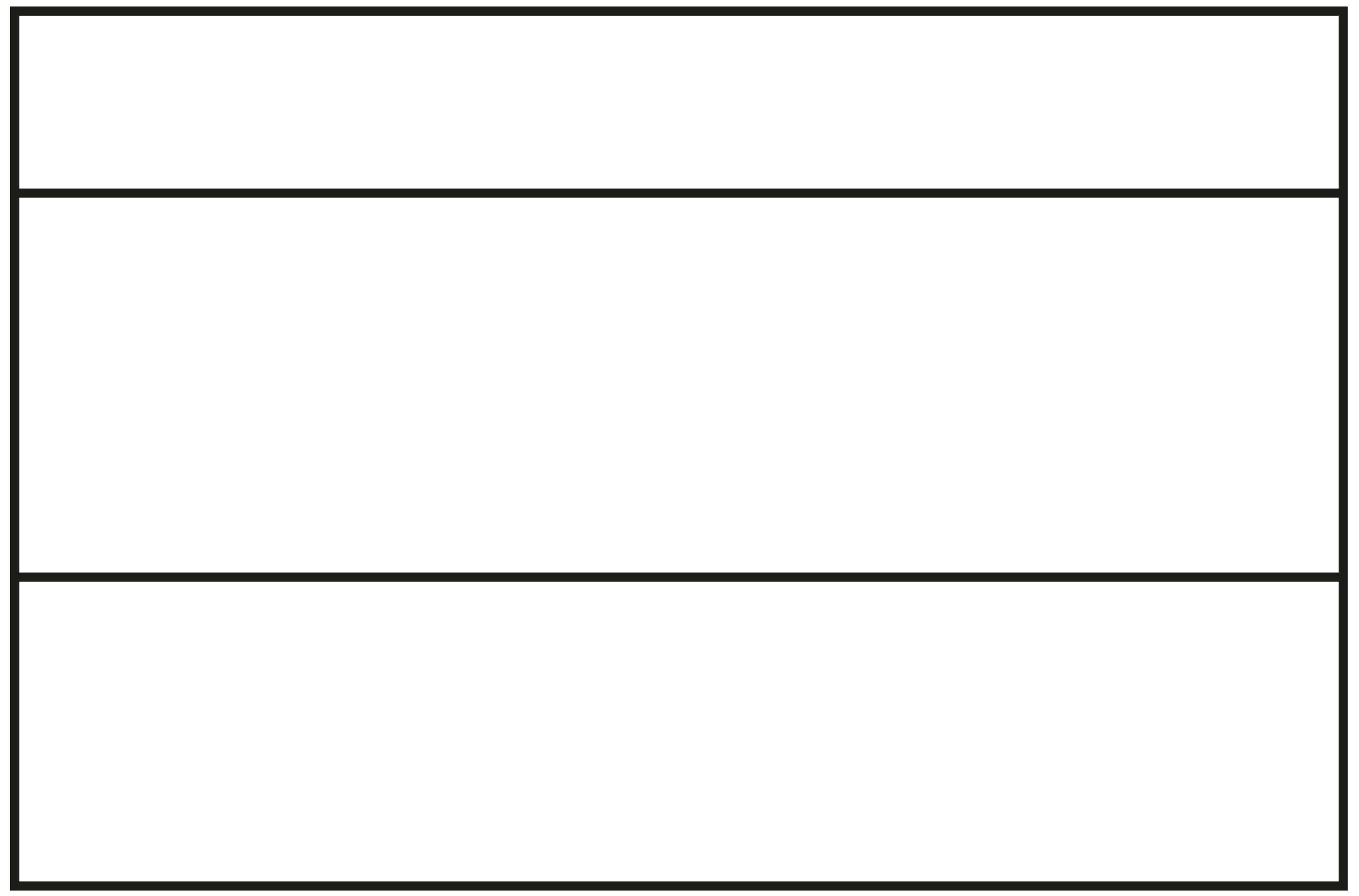
Object-Oriented programming: encapsulation example

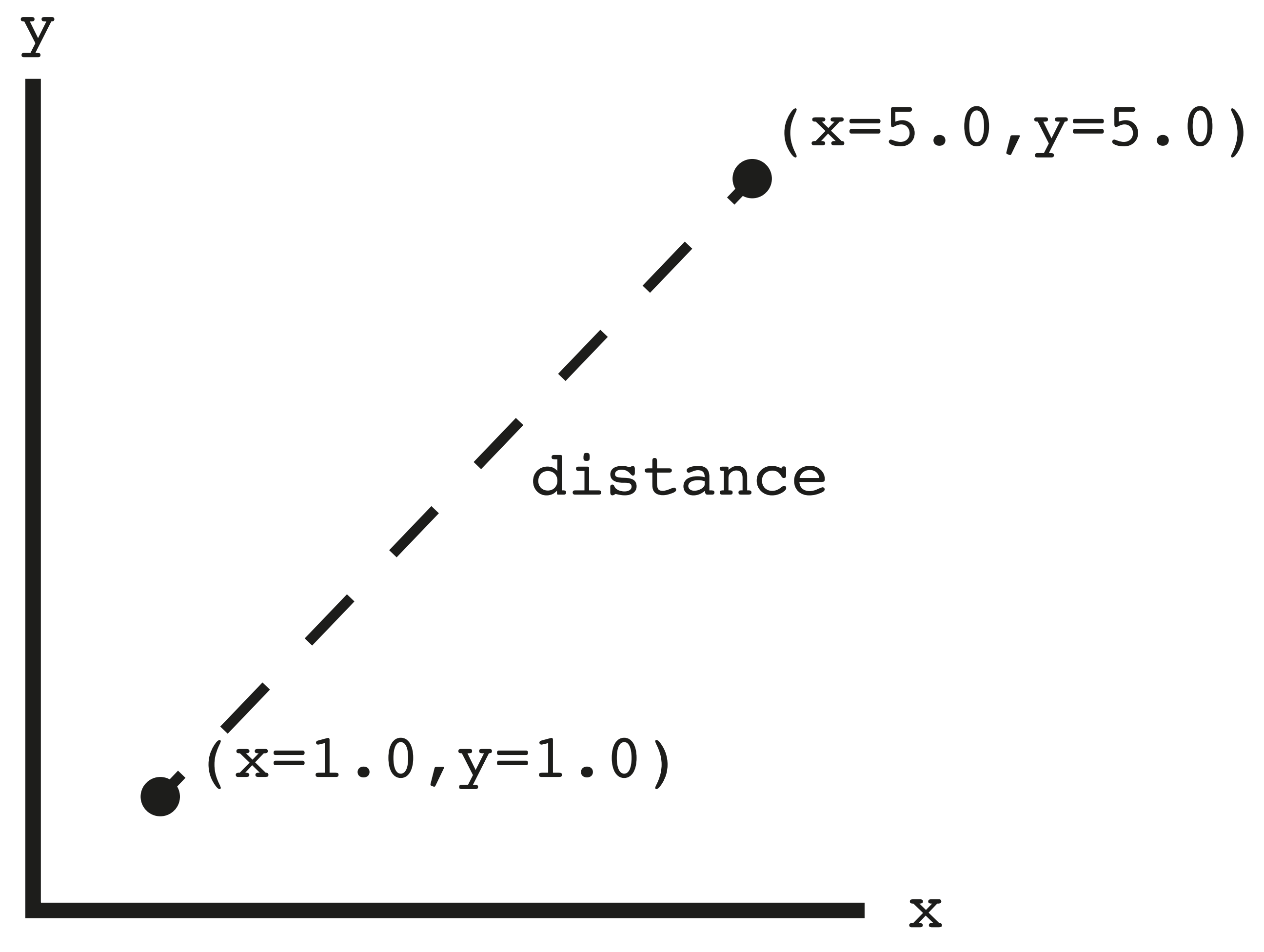


HAN_UNIVERSITY
OF APPLIED SCIENCES

Object-Oriented programming: encapsulation

- what diagram to use?
- how do we fill it in (what goes where)?

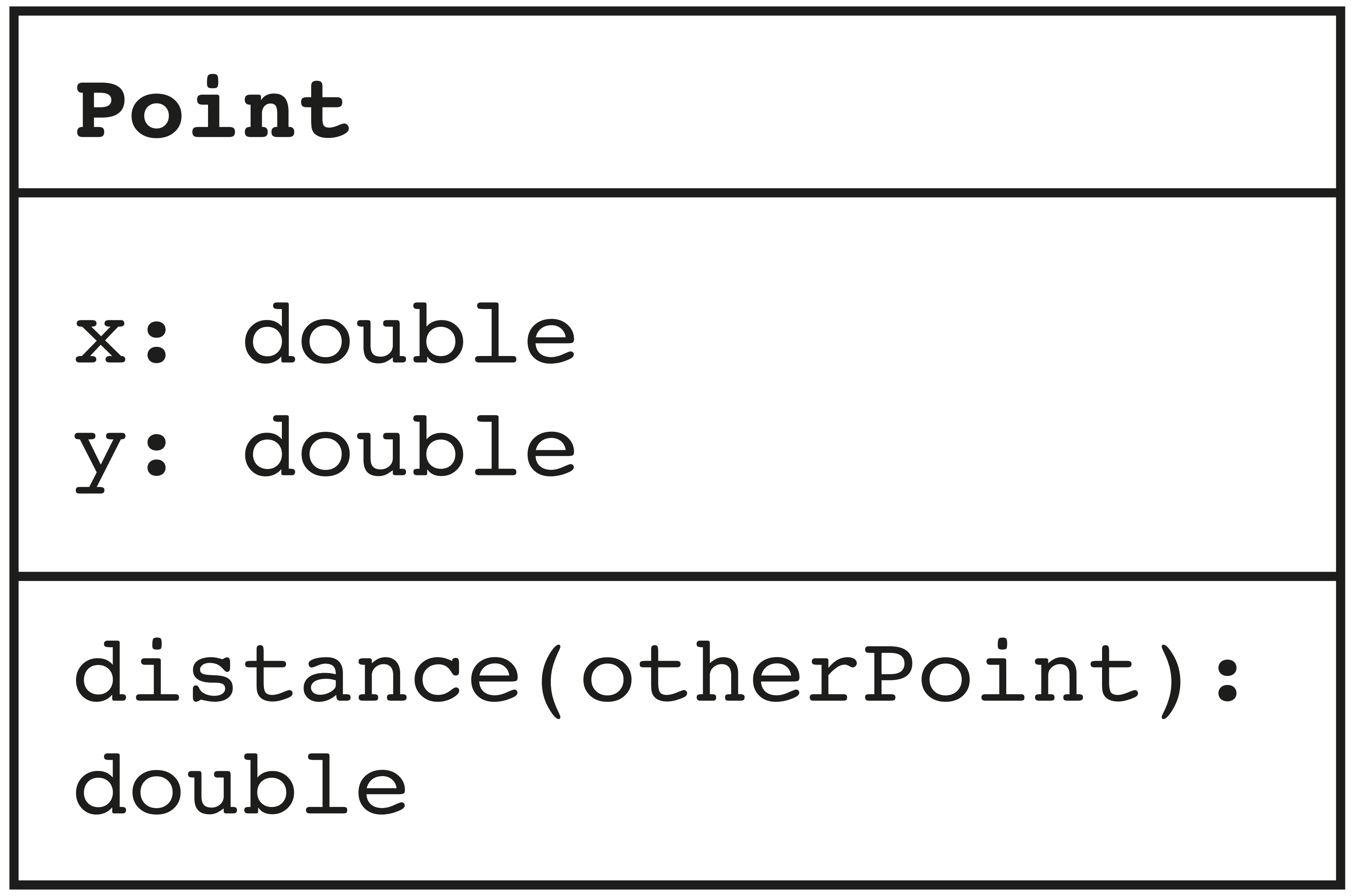




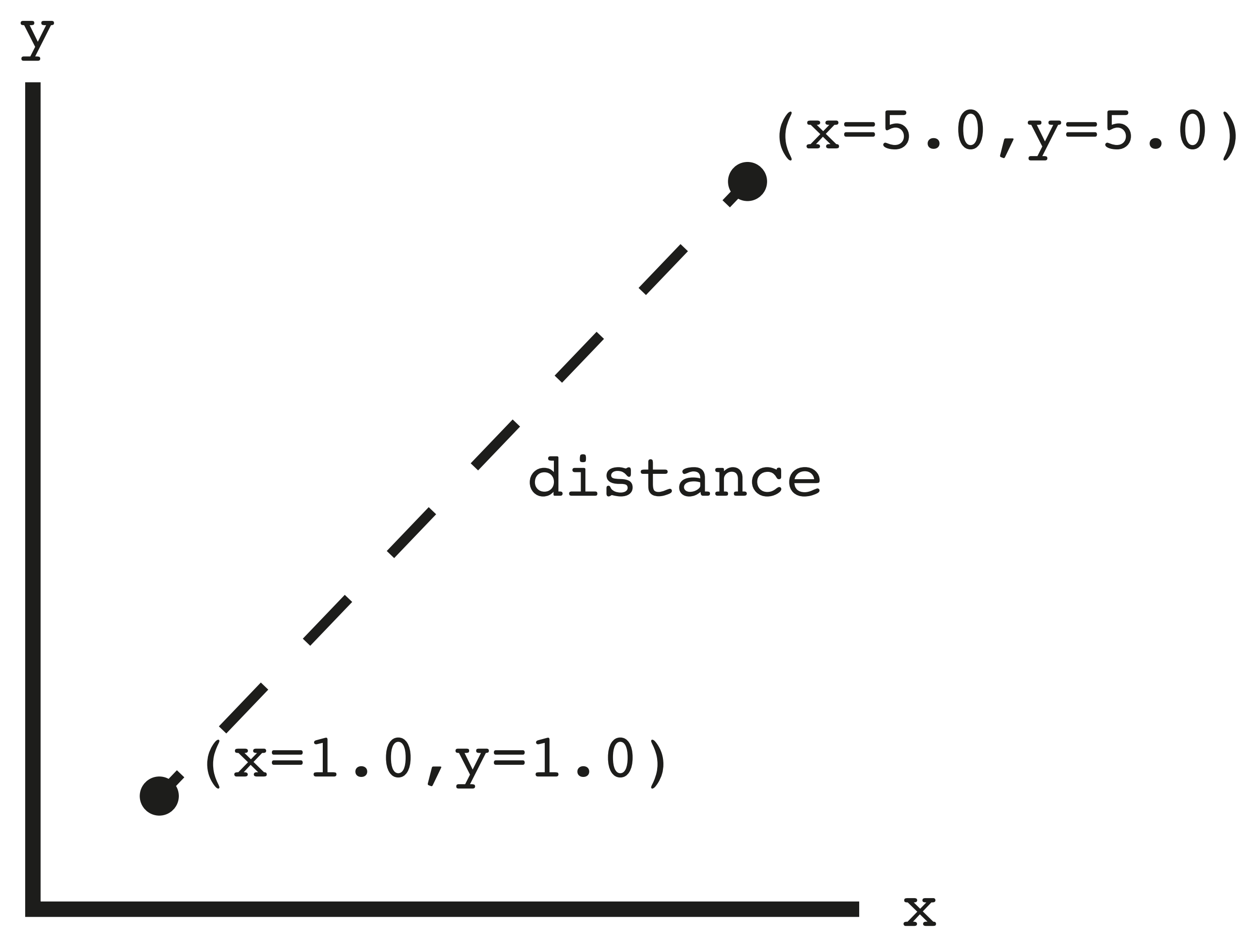
HAN_UNIVERSITY OF APPLIED SCIENCES

Object-Oriented programming: encapsulation

Class diagram:

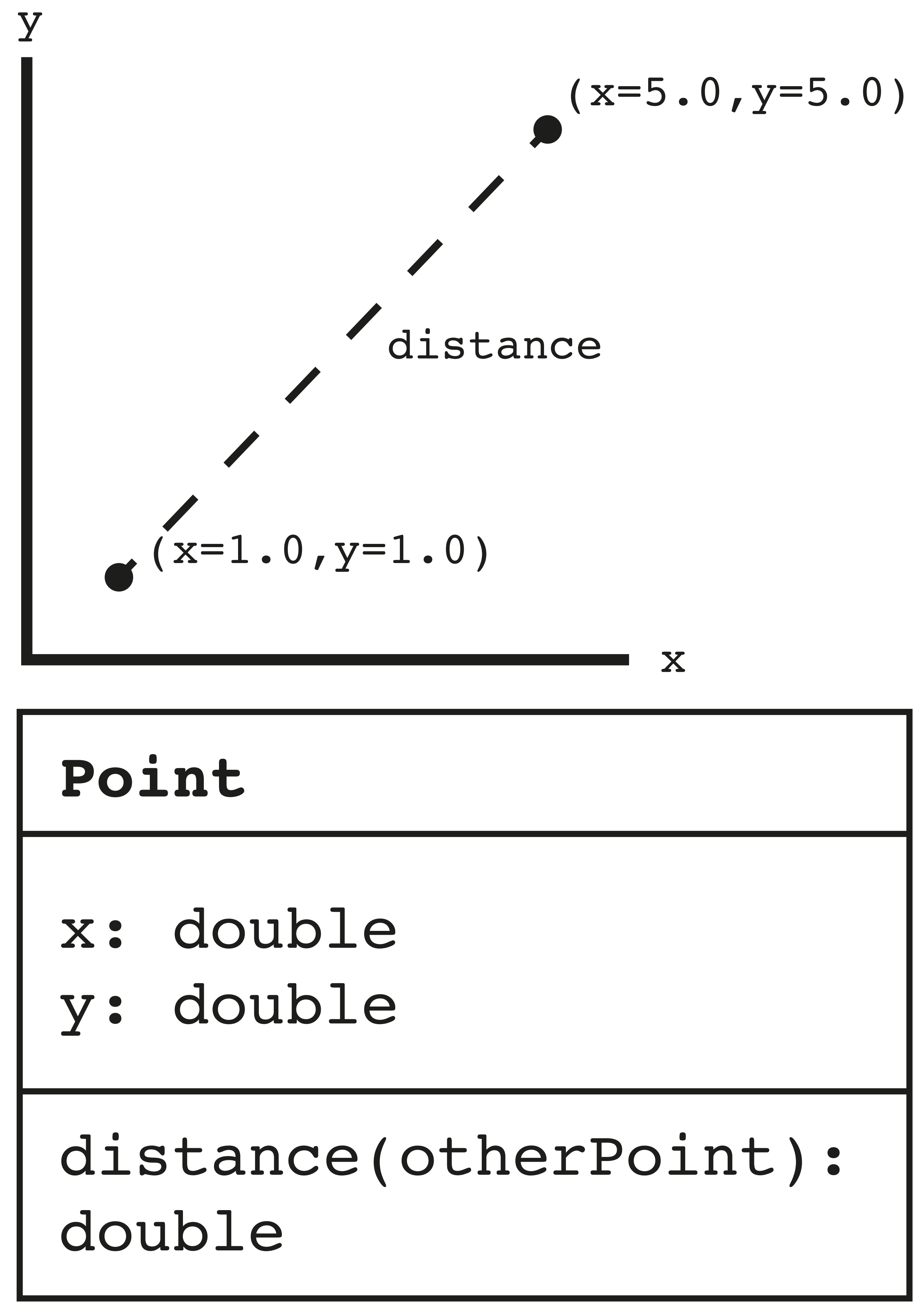


```
Point
-----------------------
x: double
y: double
-----------------------
distance(otherPoint):
double
```



Object-Oriented programming: encapsulation

How do we program it in C++?



Object-Oriented programming: encapsulation

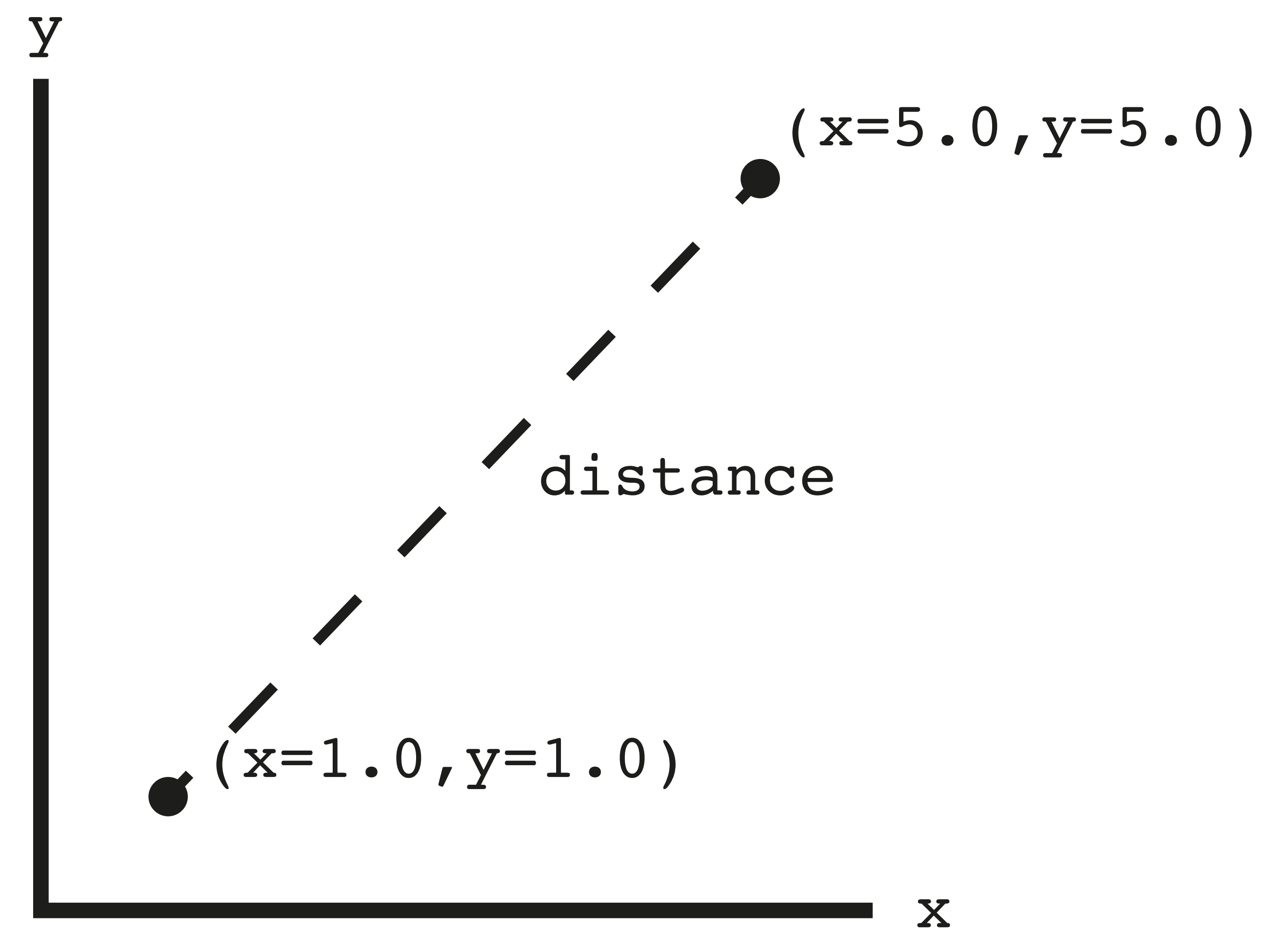
.h-file with the blueprint:

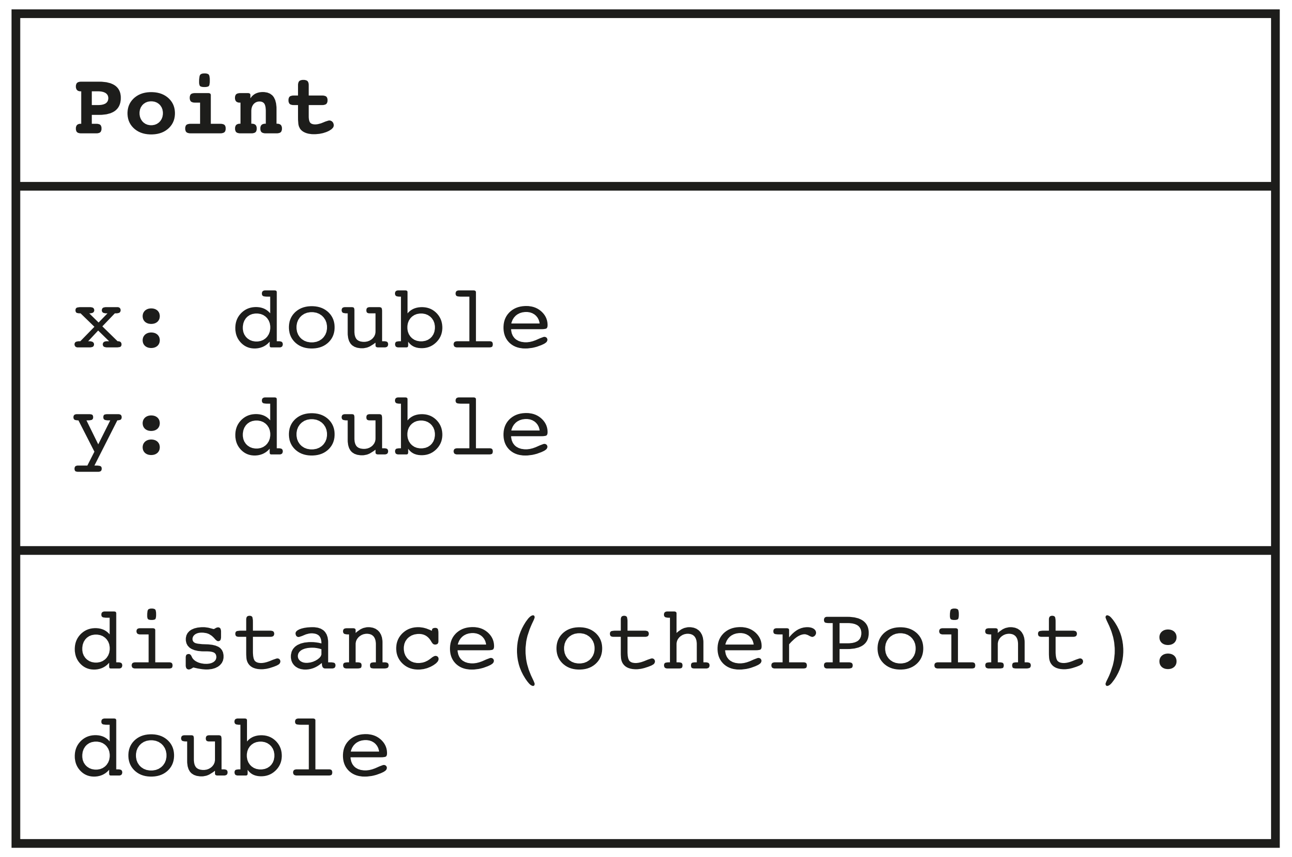
```
public:
    Point(double x, double y);
    double distance(const Point& p) const;

private:
    double x;
    double y;
};
```

Remember the names of those ‘elements’?

How does distance ‘call'?





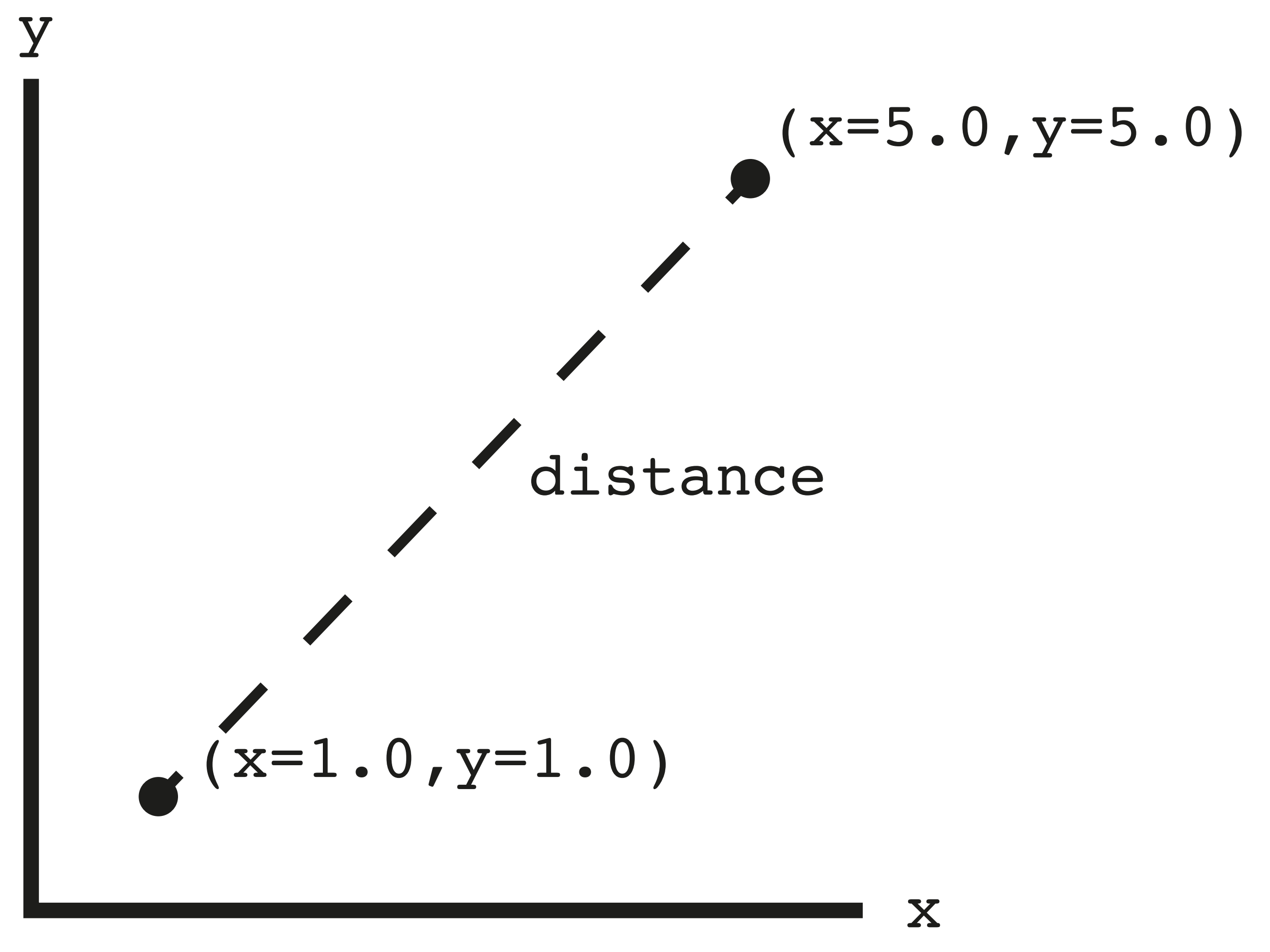
Object-Oriented programming: encapsulation

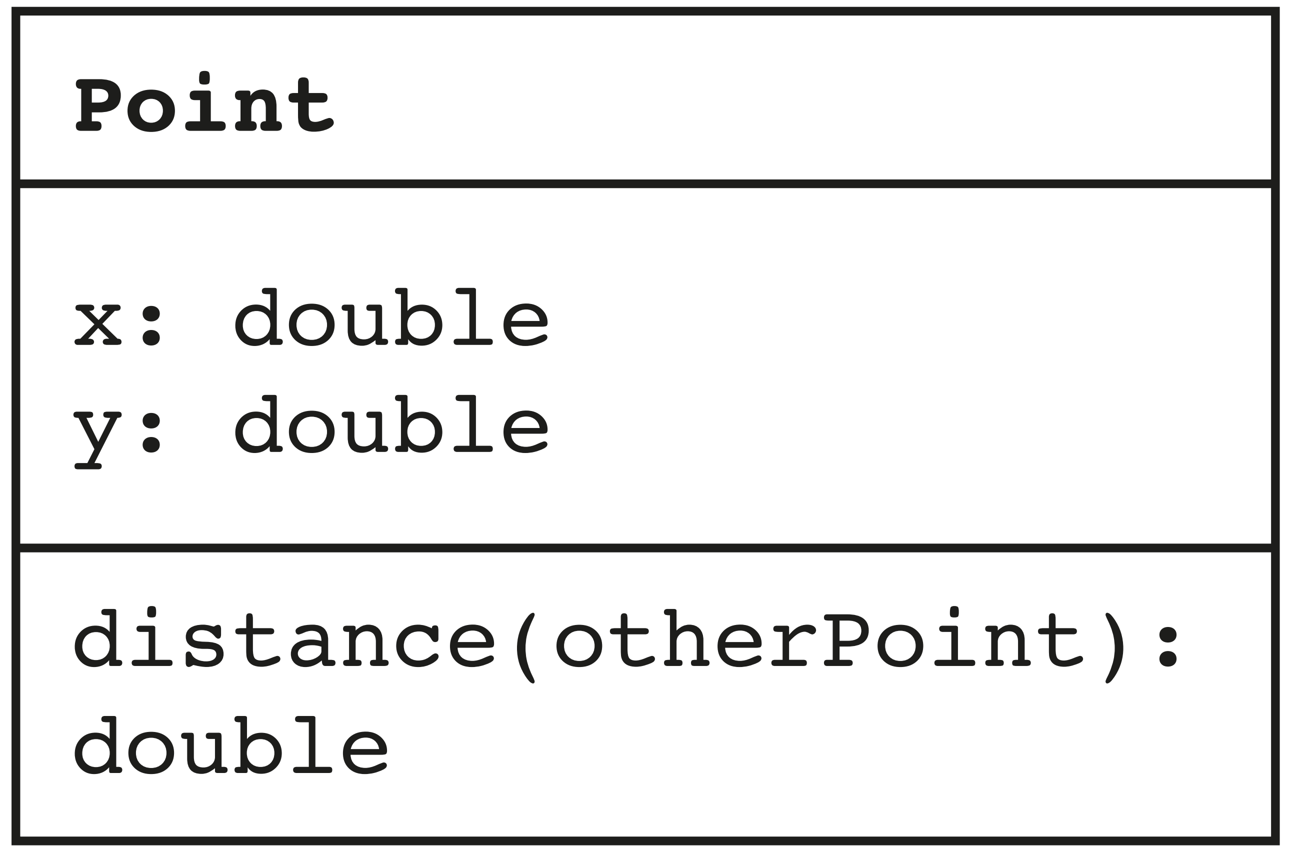
.cpp-file with the implementation:

```cpp
#include "point.h"
#include <math.h>

Point::Point(double x, double y): x(x), y(y) {}

double Point::distance(const Point& p) const {
    double dx = x - p.x;
    double dy = y - p.y;
    return sqrt(dx*dx + dy*dy);
}
```





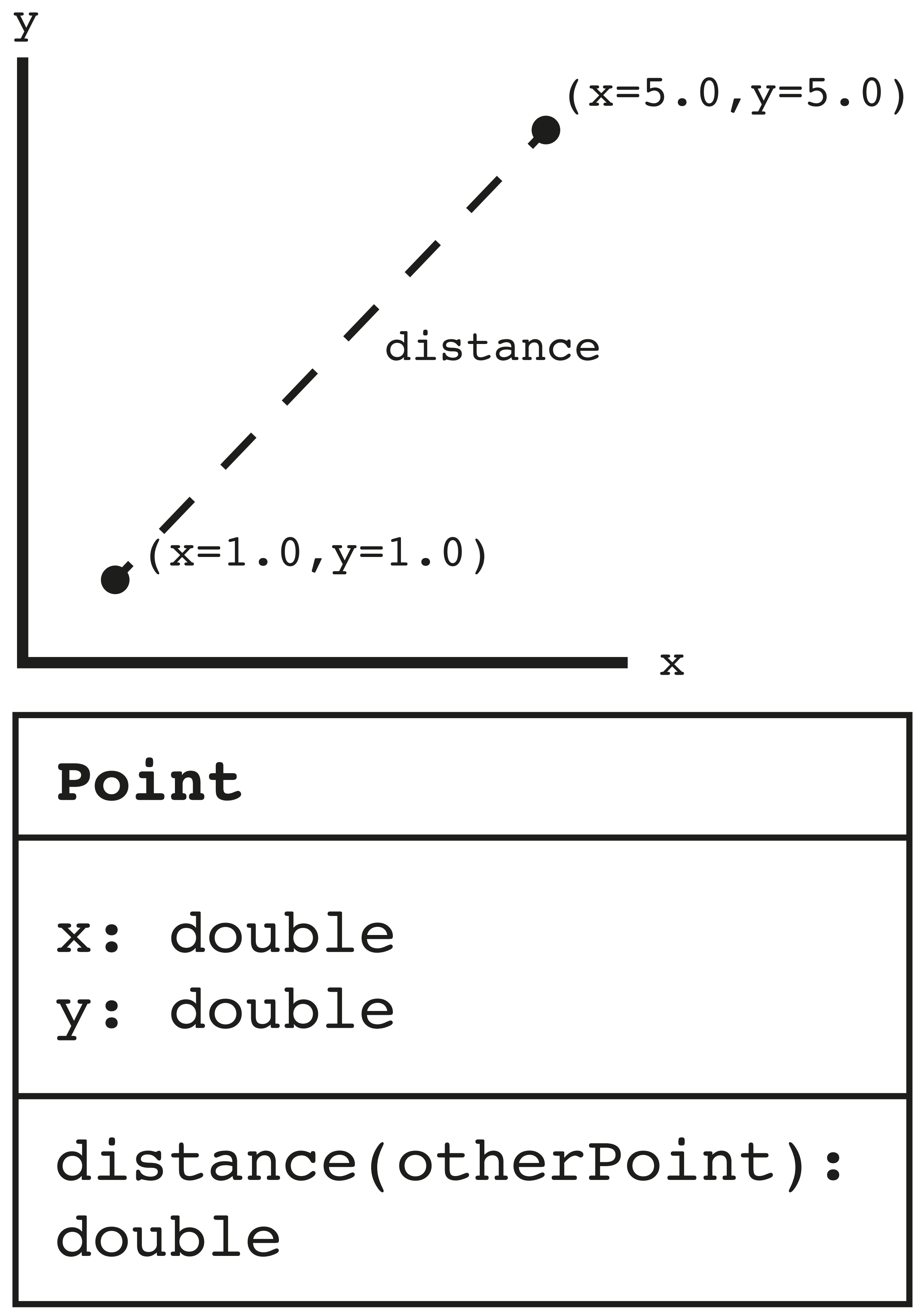
Object-Oriented programming: encapsulation

main (for ‘testing’)

```
#include <stdio.h>
#include "point.h"

int main( int argc, const char* argv[] )
{
    Point testPointA = Point(1.0, 1.0);
    Point testPointB = Point(5.0, 5.0);

    double distance = testPointA.distance(testPointB);
    printf( "\n Distance: %f\n\n", distance);
}
```



Object-Oriented programming: access

public: members are accessible from outside the class

private: members cannot be accessed (or viewed) from outside the class

protected: members cannot be accessed from outside the class, however, they can be accessed in inherited classes

Object-Oriented programming: inheritance / polymorphism

Inheritance lets us inherit attributes and methods from another class.

- a coloredPoint is also a Point but has additional features (attributes)

Polymorphism lets us use *inherited methods* to perform different tasks

We will come back to that in later classes…

Good SW Design Principles

SOLID:

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Component Principles:

- Cohesion
- Coupling

In short: loose coupling, strong cohesion.

Good SW Design Principles: Single Responsibility Principle

Single Responsibility Principle:

- Make sure your module / class / function is only responsible for one single part of functionality that your software provides

E.g. a *search and rescue* robot should be designed in such a way that the search algorithm can be changed without in ***any way*** effecting the rescue algorithm.

If you can, also design your hardware in such a way: this also help to increase *robustness and reliability*!

C++ features

Scope and Namespacing
Strings

C++ features: Scope and namespace

```
namespace a {
int i = 1;
double d = 1.1;
} // namespace a

namespace b {
int i = 2;
double d = 2.2;
} // namespace b

// ::  scope resolution operator

double d = 3.3; // Global variable

int main()
{
   double d = 4.4; // Local variable in main()

   std::cout << "a::i = " << a::i << "  b::i = " << b::i << std::endl;
   std::cout << "d = " << d << "  ::d =" << ::d << "  a::d = " << a::d
             << "  b::d = " << b::d << std::endl;
   std::cout << std::endl;

   return 0;
}
```

HAN_UNIVERSITY
OF APPLIED SCIENCES

C++ features: Scope and namespace

Make sure:

- You don’t fall into the ‘shadowing’ pit;
- your constants, variables and attributes are sound (have a understandable name); and
- your code is readable even without comments.

Note: the examples show you how certain constructs work, but are by no means meant to show good programming practices unless explicitly stated!

On C++ vectors and arrays

	Vector	Array
Template class vs. built-in	Template class (C++)	Built in (C/C++)
Memory model	Dynamic (list interface) using heap	Static (stack) Dynamic (heap) using primitive data type interface
Scalability	Flexible / resizable	Fixed
Deallocation	Automatic	Manual
Know size?	In O(1) time	No (if dynamically allocated)
Passable as argument?	Yes, directly	With size as second arg
Automatic reallocation when full?	Yes	No
Return type?	Yes	No, unless dynamically allocated
Direct copying?	Yes	No

https://www.geeksforgeeks.org/advantages-of-vector-over-array-in-c/

“

Any questions?

HAN_UNIVERSITY
OF APPLIED SCIENCES