Modularization and Software Architectures

UNIVERSITÄT SALZBURG

Contents

- The term software module (= software component)
- Desired characteristics of modules
- Specification of modules (ADS, ADT)
- Description of software architectures
- Analysis of software architectures
- Multidimensional modularity by aspect-oriented programming (AOP)

UNIVERSITÄT SALZBURG

The Software Module (Component)

UNIVERSITÄT SALZBURG

Definition

A *module* (software component) is defined as *a piece of software with a programming interface*.

One distinguishes between the **interface** of a module and its **implementation**. The possible interaction of several modules is determined by their programming interfaces.

- The programming interface is the **export interface**, which indicates what operations and data a module makes available to other modules.
- The **import interface** indicates what a module uses from other modules.

In the sequel, by interface we mean export interface unless explicitly noted otherwise.

UNIVERSITÄT SALZBURG

Module as a means of abstraction

M. Reiser and N. Wirth (1992) describe the module concept as follows:

It provides mechanisms for:

(1) structuring of a program into independent units;

(2) the declaration of variables that keep their value for the duration the module is active (that is, in memory) – these variables are called global *to the module;*

(3) export of variables *and procedures to be used in other modules.*

The module therefore provides facilities for abstractions.

UNIVERSITÄT SALZBURG

Module as expression means for modeling

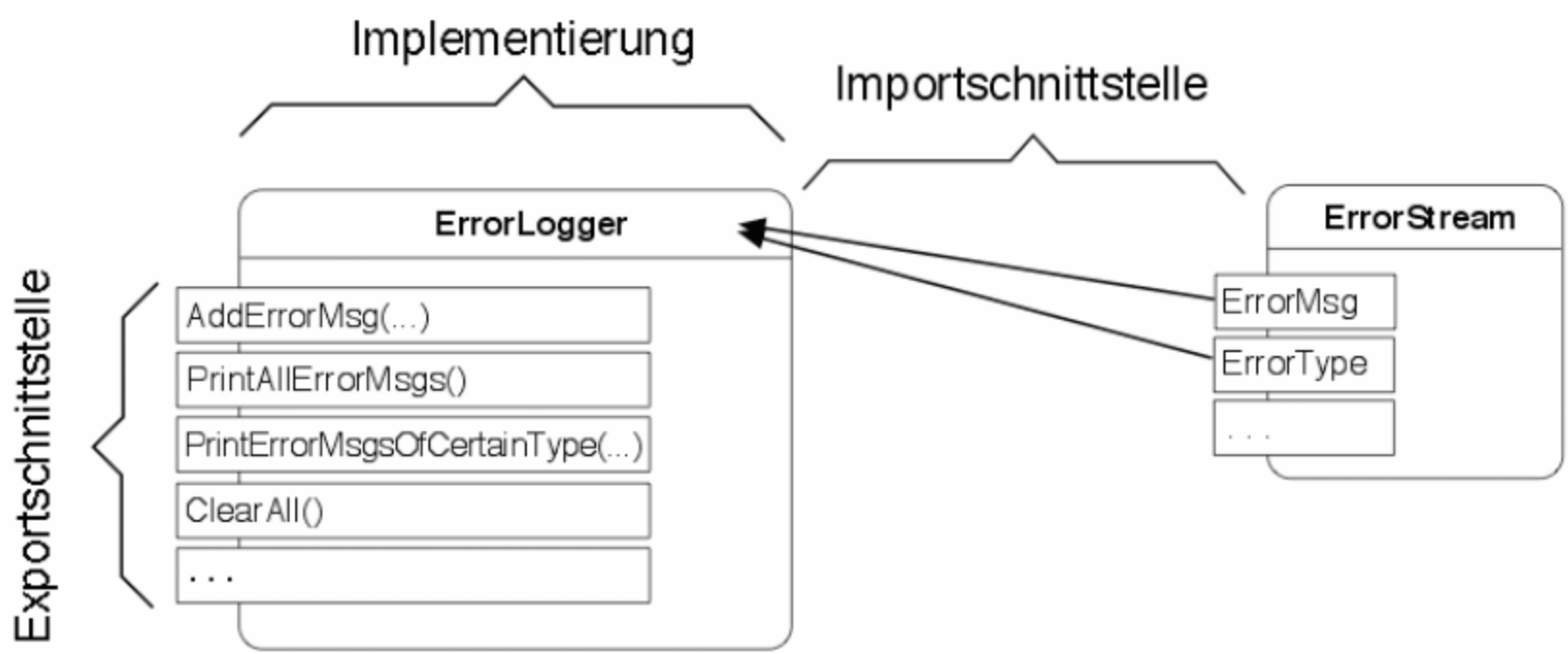
By a skillful factorization of functions, procedures and data, one can form abstractions which correspond to entities from the material world as modeled in a software system.

Examples:

- a bank account,
- the GPS navigation unit of a helicopter,
- the file organization of a PC,
- the interaction elements of a graphic user interface.

UNIVERSITÄT SALZBURG

Example: Module for storing error messages



UNIVERSITÄT SALZBURG

Module interface of ErrorLogger (in Modula-2)

```
DEFINITION MODULE ErrorLogger;
  FROM ErrorStream
   IMPORT ErrorMsg, ErrorType; /* Import interface */
  PROCEDURE AddErrorMsg(↓em: ErrorMs);
  PROCEDURE PrintAllErrorMsgs();
  PROCEDURE PrintErrorMsgsOfCertainType(↓et: ErrorType);
  PROCEDURE ClearAll();
  . . .
END ErrorLogger.
```

UNIVERSITÄT SALZBURG

Module implementation of ErrorLogger (in Modula-2)

```
IMPLEMENTATION MODULE ErrorLogger;

  VAR errors: ARRAY [0..cMaxNoOfStoredErrors–1] OF ErrorMsg;

  PROCEDURE AddErrorMsg(↓em: ErrorMs)
     BEGIN
        ...          /* insert em at the next free space in the field errors */
  END AddErrorMsg;
  . . .

END ErrorLogger.
```

ARRAY => statically established upper limit for the number of storable error messages

UNIVERSITÄT SALZBURG

Advantage of the separation of interface from implementation

The implementation of the module can be improved or changed without changing the interface.

For example, in the module ErrorLogger the array structure could be replaced by a concatenated list.

The concept of *information hiding* (see the following section) contributes to a stable interface of a module.

UNIVERSITÄT SALZBURG

Desired Characteristics of Modules

UNIVERSITÄT SALZBURG

Stable and Understandable Module Interfaces by Information Hiding (I)

The design principle *Information Hiding* goes back to David L. Parnas (1972).

Thus, modules are to be designed in such a way that the data structures are hidden from the user.

Access to data and its manipulation is possible only over access procedures, which are aforementioned in the module interface.

UNIVERSITÄT SALZBURG

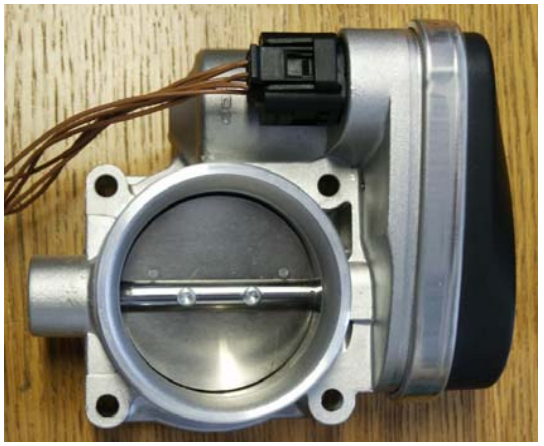
Stable and Understandable Module Interfaces by Information Hiding (II)

- A generalization of the information hiding principle is the requirement that in the design of modules one takes care to hide as many details of the implementation as possible behind the module interface, in order not to confront the user of a module with unnecessary details and complexity.

- This can go beyond just hiding the data structures.

UNIVERSITÄT SALZBURG

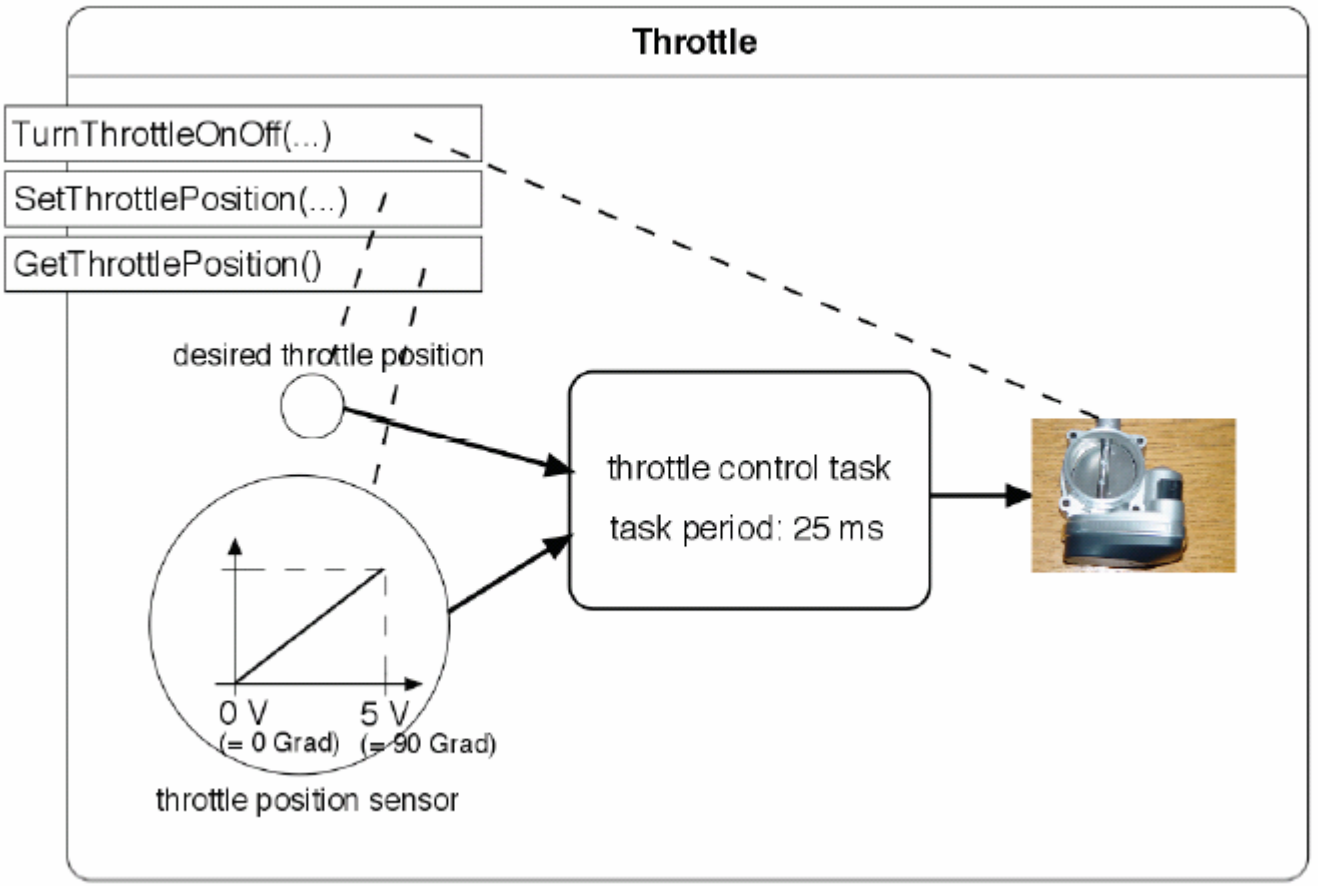
Example: Module Throttle for controlling a butterfly valve (I)



```
interface Throttle {
  bool TurnThrottleOnOff(bool onOff);
  bool SetThrottlePosition(float angle);    // 0..90 grades
  float GetThrottlePosition();
}
```

UNIVERSITÄT SALZBURG

Example: Module Throttle for controlling a butterfly valve (II)



UNIVERSITÄT SALZBURG

Balance between coupling and cohesion (I)

- By *module coupling* we understand the dependence and interaction between modules, which is specified on one hand statically by the import interface and on the other hand dynamically by calls of procedures and functions and/or by the access to data. **The module coupling is to be minimized**.

- By *module coherence* we understand the degree to which data and operations which logically belong together are bound to the same module. **The module coherence is to be maximized.**

UNIVERSITÄT SALZBURG

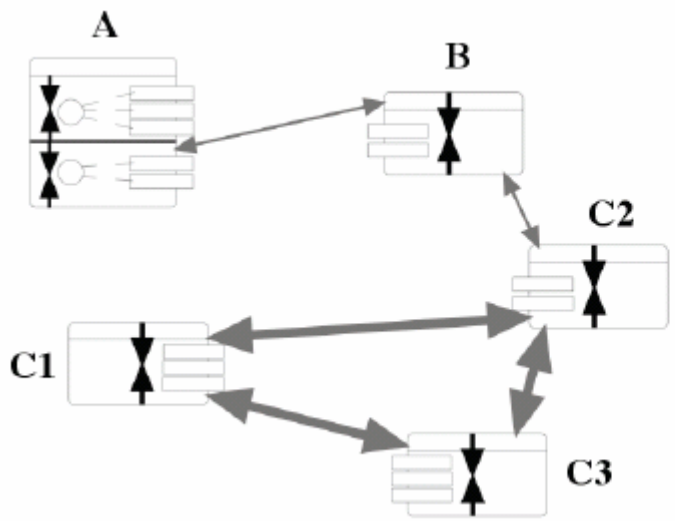
Balance between coupling and cohesion (II)

- One could bring the coupling to a minimum, by defining only one module. If only one module is present, this is coupled with no other module, hence the coupling is minimal. However such a module would have also a minimum coherence (except in trivial cases), since all aspects of the system are mixed in the module.

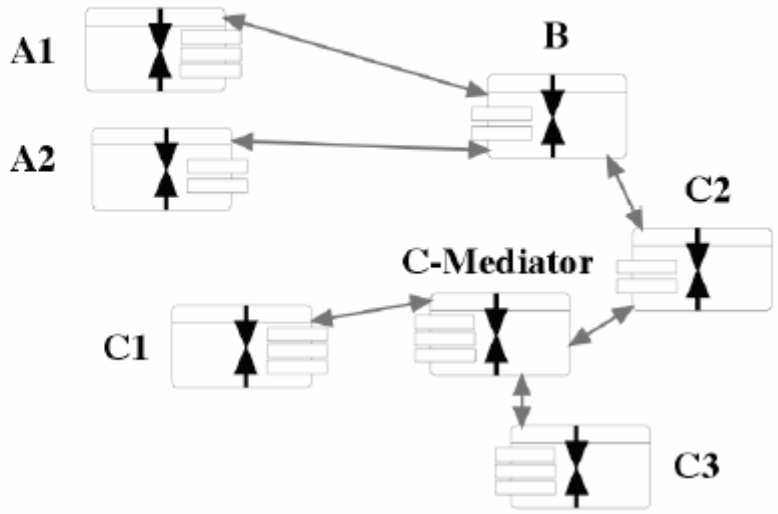
- The other extreme would be that each function and procedure is enclosed in a separate module. That would lead to a very strong coupling of the modules. One cannot talk about coherence in this setup.

UNIVERSITÄT SALZBURG

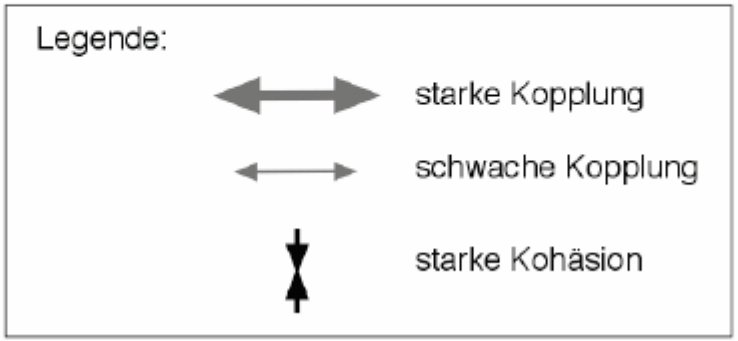
Example for improved modularity



(a)



(b)



UNIVERSITÄT SALZBURG

Rules for coherence maximization (I)

- The **data structures** (instance variables in classes) and the **operations** (=functions/procedures/methods) **should be in close relationship to each other**. Distinct groups of operations, working on different data, are an indicator for the fact that logically unrelated aspects were included in a module. Splitting up the module contributes to the improvement of the coherence.
- **The module interface should not contain redundant operations**. This is applicable if several slightly different operations are specified for the same functionality. In addition, each operation should be conceived such that it uses a small number of parameters .

UNIVERSITÄT SALZBURG

Rules for coherence maximization (II)

- A **consistent and expressive naming schema** is to be used. This applies in particular to the names of modules as well as to the names of the operations defined in the interfaces. Examples of well selected and consistent naming schemas are modern object-oriented class libraries like the .NET and Java libraries.

- **Global data objects should be avoided**

UNIVERSITÄT SALZBURG

Heuristic for achieving an adequate coupling

- The individual module should be well understandable in itself. In other words, for understanding a module it should not be necessary to look at other modules.

- A similar statement applies to testing modules. The larger is the ensemble of modules which is needed to test a module, the stronger is the coupling of the respective module with the other modules.

UNIVERSITÄT SALZBURG

Evaluation of modularization quality (I)

- The software architecture analysis method (SAAM) aims to examine the coupling and coherence for low complexity systems.

- There is no generally valid metric that can evaluate the two characteristics objectively.

UNIVERSITÄT SALZBURG

Evaluation of modularization quality (II)

- The correct balance of coupling and coherence in the structuring of software is therefore an art, which requires high qualification and much experience.
- As in Architecture, examples can help to create awareness and feeling for good modularity.
- Contrary to Architecture, unfortunately only few good examples of software architectures are available in the open literature.

UNIVERSITÄT SALZBURG

Module Specification

UNIVERSITÄT SALZBURG

Module as Abstract Data Structure (ADS)

- A module is defined without specifying a type
- Example: Module ErrorLogger as defined above

```
DEFINITION MODULE ErrorLogger;
  FROM ErrorStream
     IMPORT ErrorMsg, ErrorType;              /* Import interface */
  PROCEDURE AddErrorMsg(↓em: ErrorMs);
  PROCEDURE PrintAllErrorMsgs();
  PROCEDURE PrintErrorMsgsOfCertainType(↓et: ErrorType);
  PROCEDURE ClearAll();
  . . .
END ErrorLogger.
```

UNIVERSITÄT SALZBURG

Module as Abstract Data Type (ADT) I

- A module is defined as a type and one can form as many instances of it as desired.
- In Modula-2: Module ErrorLogger with *opaque type* ErrorLogger.

```
DEFINITION MODULE ErrorLoggers;
        FROM ErrorStream IMPORT ErrorMsg, ErrorType;
        TYPE ErrorLogger;
        PROCEDURE NewErrorLogger(↑VAR el: ErrorLogger);
        PROCEDURE AddErrorMsg(↓↑VAR el: ErrorLogger,
                              ↓em: ErrorMsg);
        PROCEDURE PrintAllErrorMsgs(↓el: ErrorLogger);
        PROCEDURE PrintErrorMsgsOfCertainType(↓el: ErrorLogger,
                                              ↓et: ErrorType);
        PROCEDURE ClearAll(↓↑VAR el: ErrorLogger);
        . . .
    END ErrorLoggers.
```

Module as Abstract Data Type (ADT) II

Producing instances with `NewErrorLogger`:

```
calcErrors, inputErrors: ErrorLogger;
NewErrorLogger(↑calcErrors);
NewErrorLogger(↑inputErrors);
AddErrorMsg(↓↑calcErrors, ↓any message);
. . .
PrintAllErrorMsgs(↓calcErrors);
PrintAllErrorMsgs(↓inputErrors);
```

UNIVERSITÄT SALZBURG

ADT in OO Languages (e.g. in C#) I

```
using System;
    using System.Collections;
    namespace ErrorLibrary {
        public class ErrorLogger {
            private IList errorList;

            public ErrorLogger() {
              errorList= new ArrayList();
            }

            public void AddErrorMsg(ErrorMsg em) {
              errorList.Add(em);
            }
            . . .
        }
    }
```

Interface and implementation in the same file!

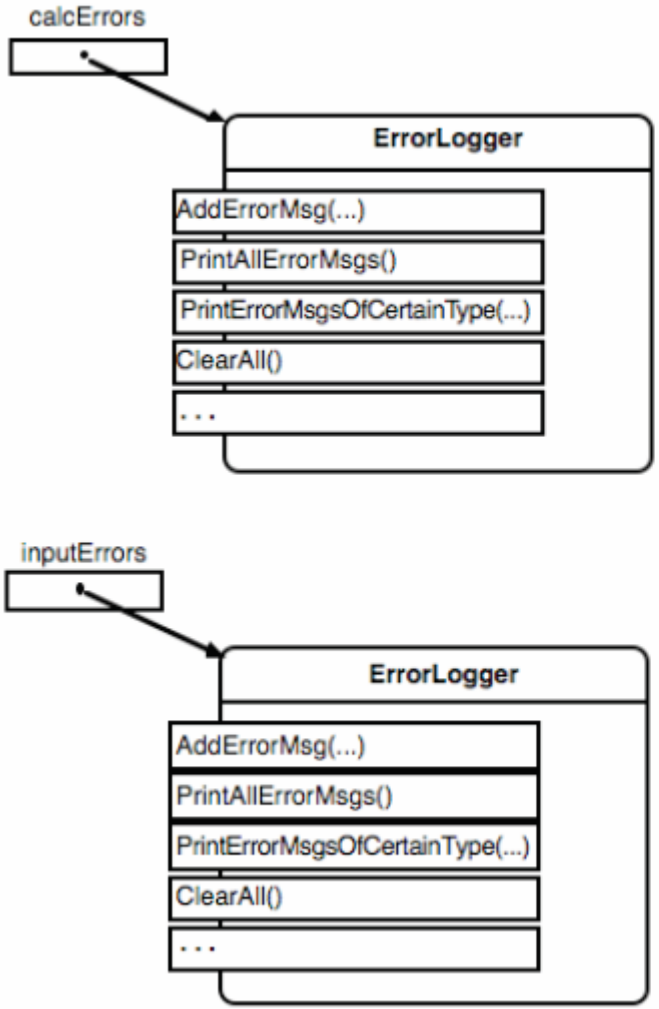
UNIVERSITÄT SALZBURG

ADT in OO Languages (e.g. in C#) II

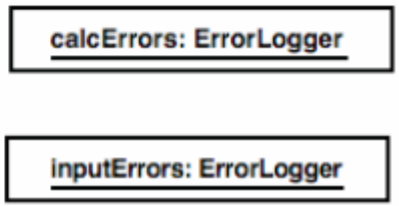
```
ErrorLogger calcErrors, inputErrors;
calcErrors = new ErrorLogger();
inputErrors = new ErrorLogger();
. . .
try {
        ...
} catch (FloatingPointException fe) {
        calcErrors.AddErrorMsg(new ErrorMsg("floating point exception“,
                                ...);    // ... means more information
}
```

UNIVERSITÄT SALZBURG

ADT in OO Languages III



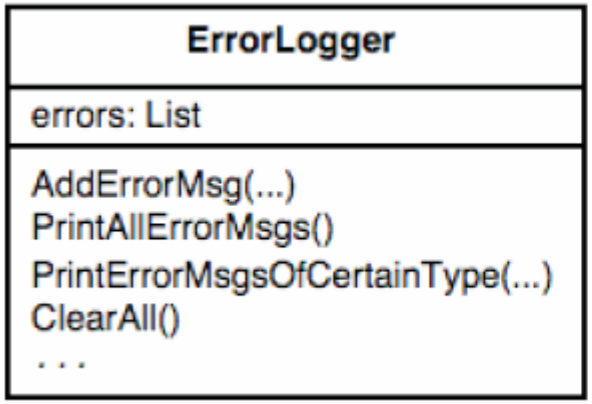
in UML:

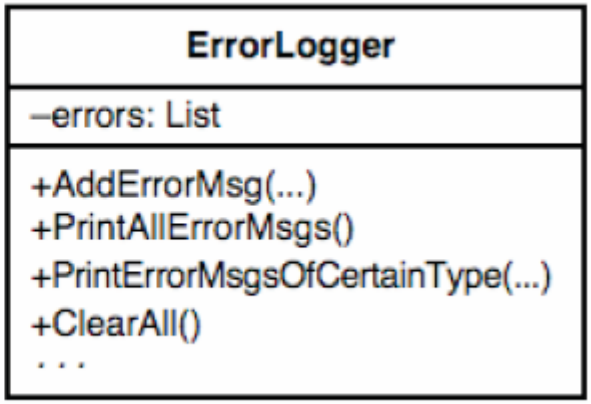


UNIVERSITÄT SALZBURG

ADT in OO Languages IV

as UML class diagram:





UNIVERSITÄT SALZBURG

Definition of ADS and ADT in programming languages

- Oberon (- 2): supports both concepts by language constructs

- Java and C#: Classes for the definition of ADT. Syntactic support of the definition of ADS by static instance variables and methods.

  Packages and name spaces let several classes combine into a unit.

UNIVERSITÄT SALZBURG

Modules in current component standards

- Component standards differ among other things in the syntax, how the interface is defined by components:
  - CORBA (Common Object Request Broker Architecture): CORBA IDL (Interface Description LANGUAGE); maps to C++
  - JavaBeans: Interface is defined in Java
  - Web services: XML-based WSDL (Web Services Description Language)

UNIVERSITÄT SALZBURG