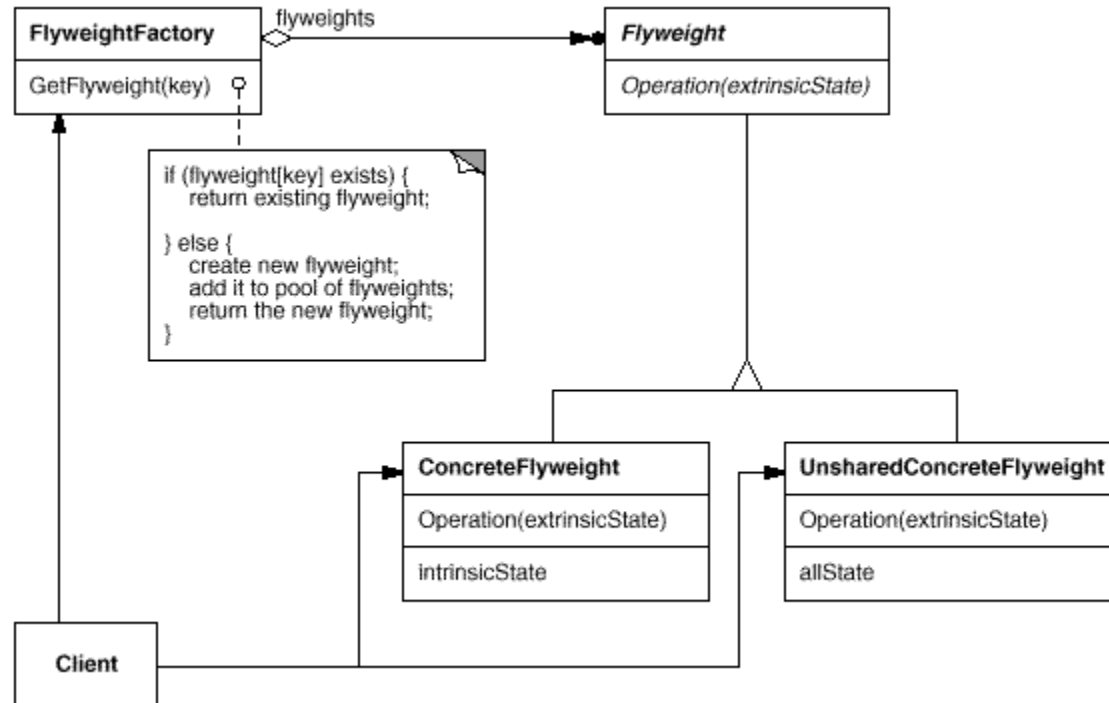


Construction Principles and Design Patterns

Flyweight, Bridge, Builder

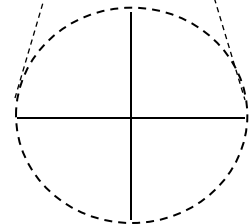
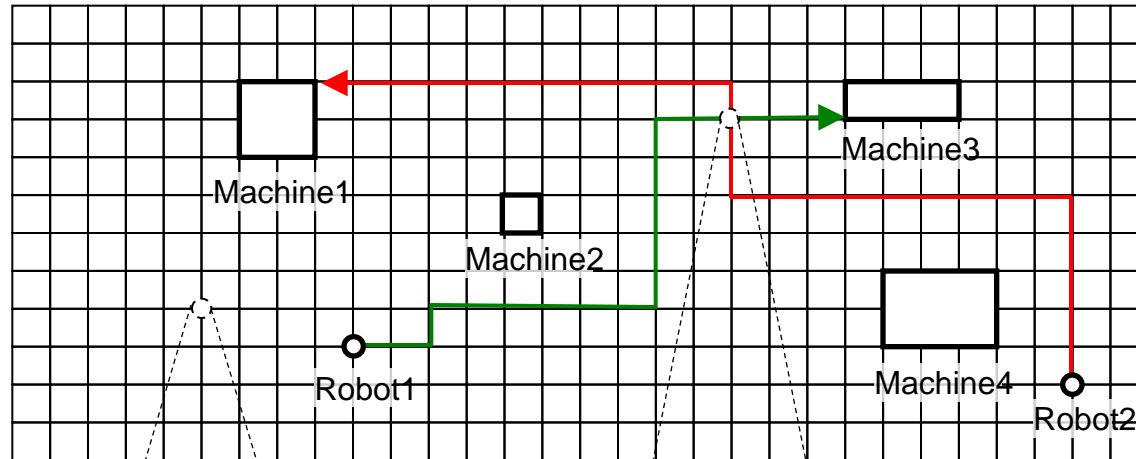
The Flyweight Design Pattern: Structure



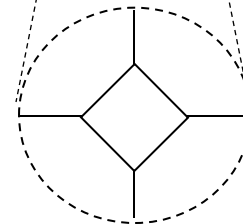
Use:

- To avoid employing a large number of objects with similar state
- When objects with intrinsic state can be re-used by different clients

The Flyweight Design Pattern: Example(I)

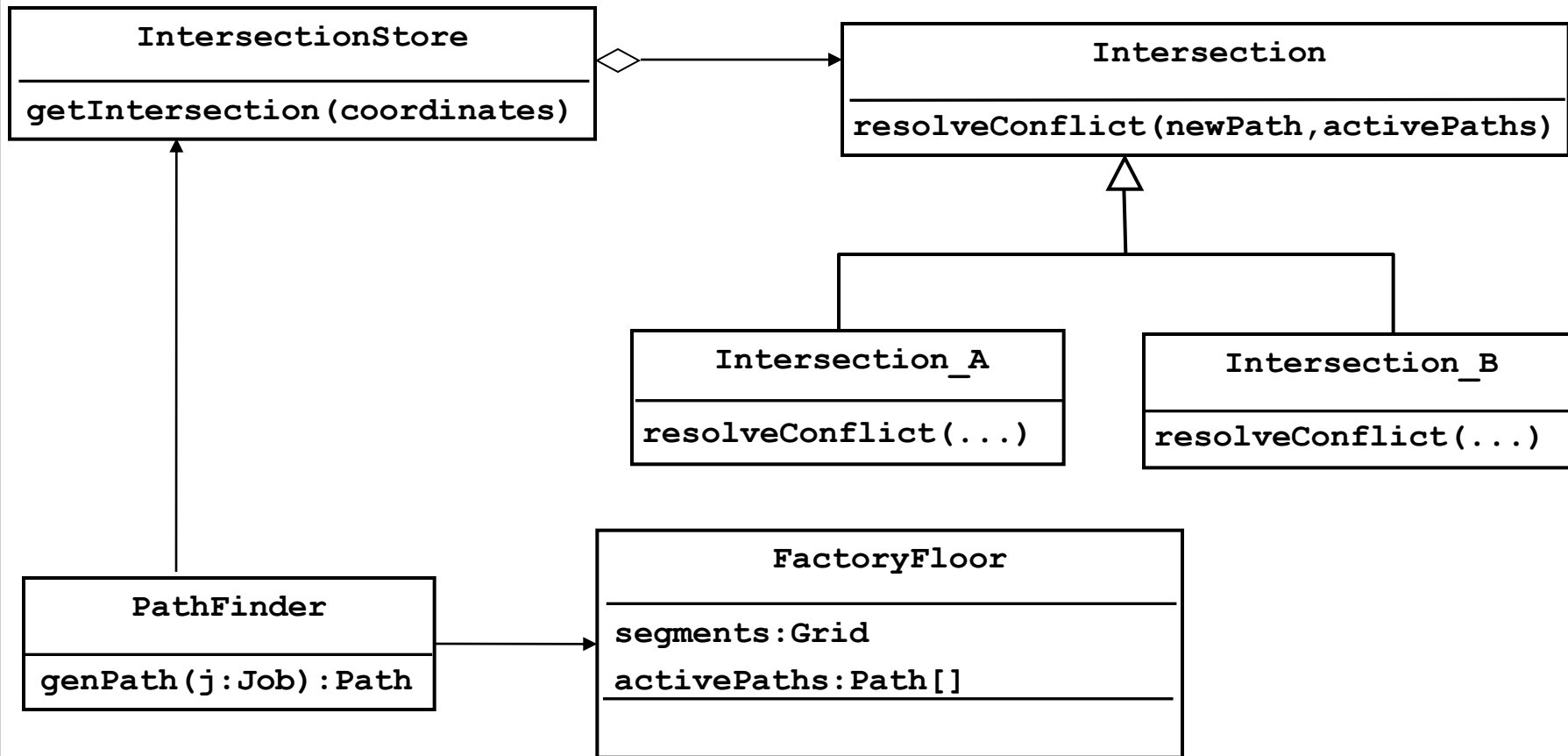


Intersection of type A



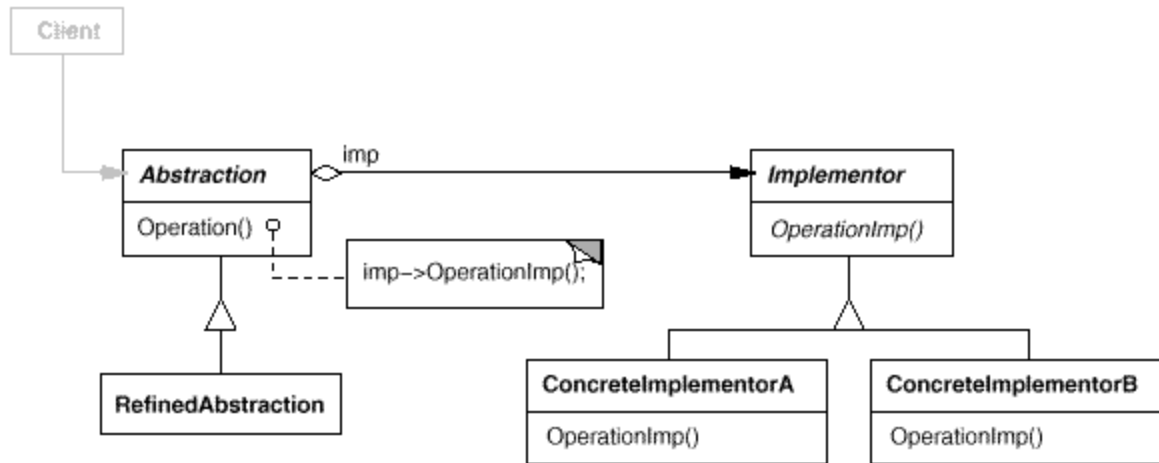
Intersection of type B

The Flyweight Design Pattern: Example(II)



- New intersection types can be easily added!

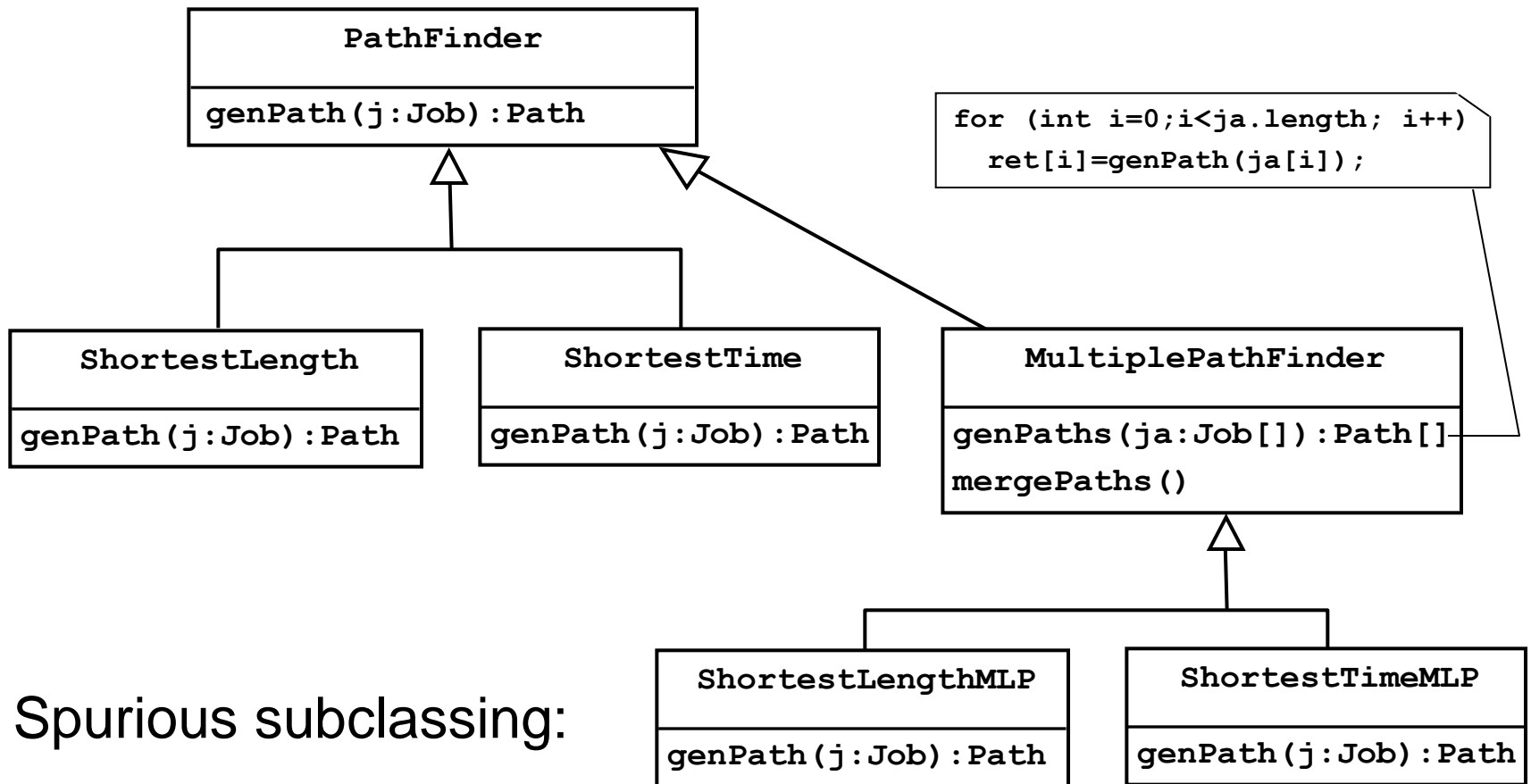
The Bridge Design Pattern: Structure



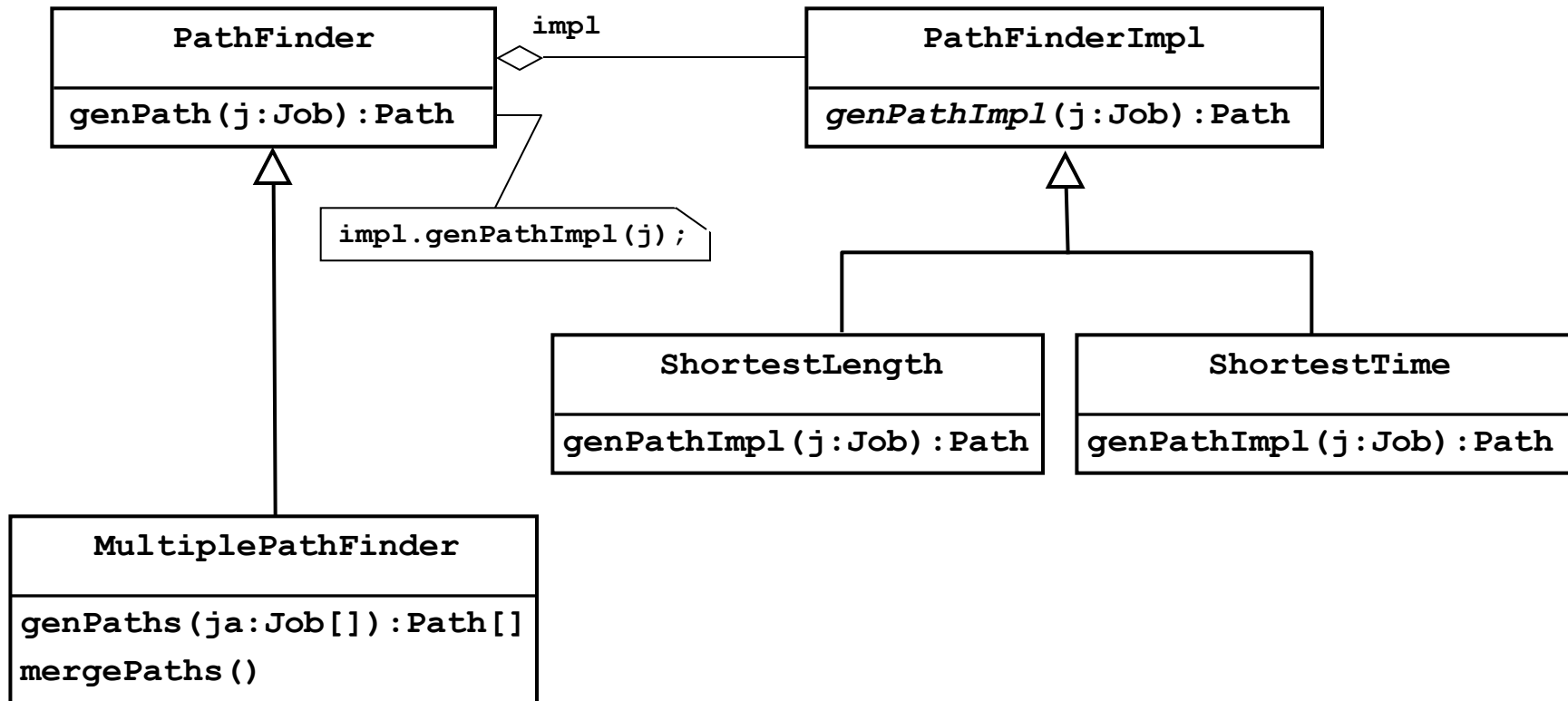
Used when:

- An abstraction should be decoupled from its implementation
- The abstraction and its implementation should be (independently) extensible by subclassing

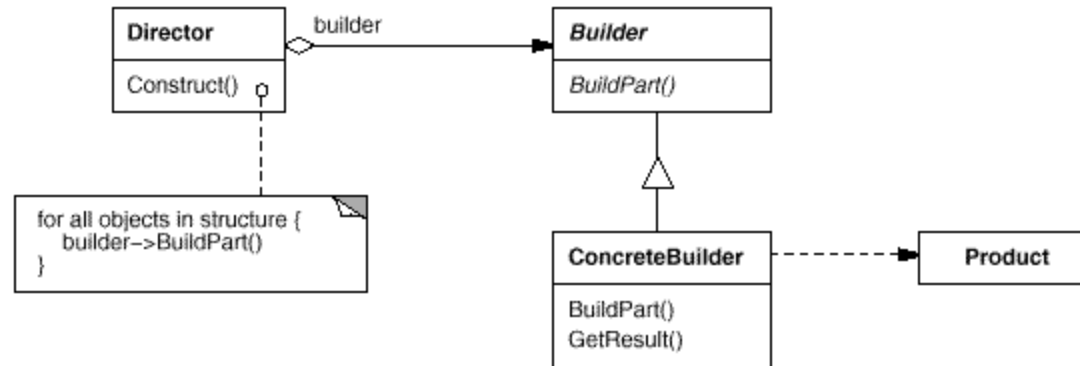
The Bridge Design Pattern: Example (I)



The Bridge Design Pattern: Example (II)



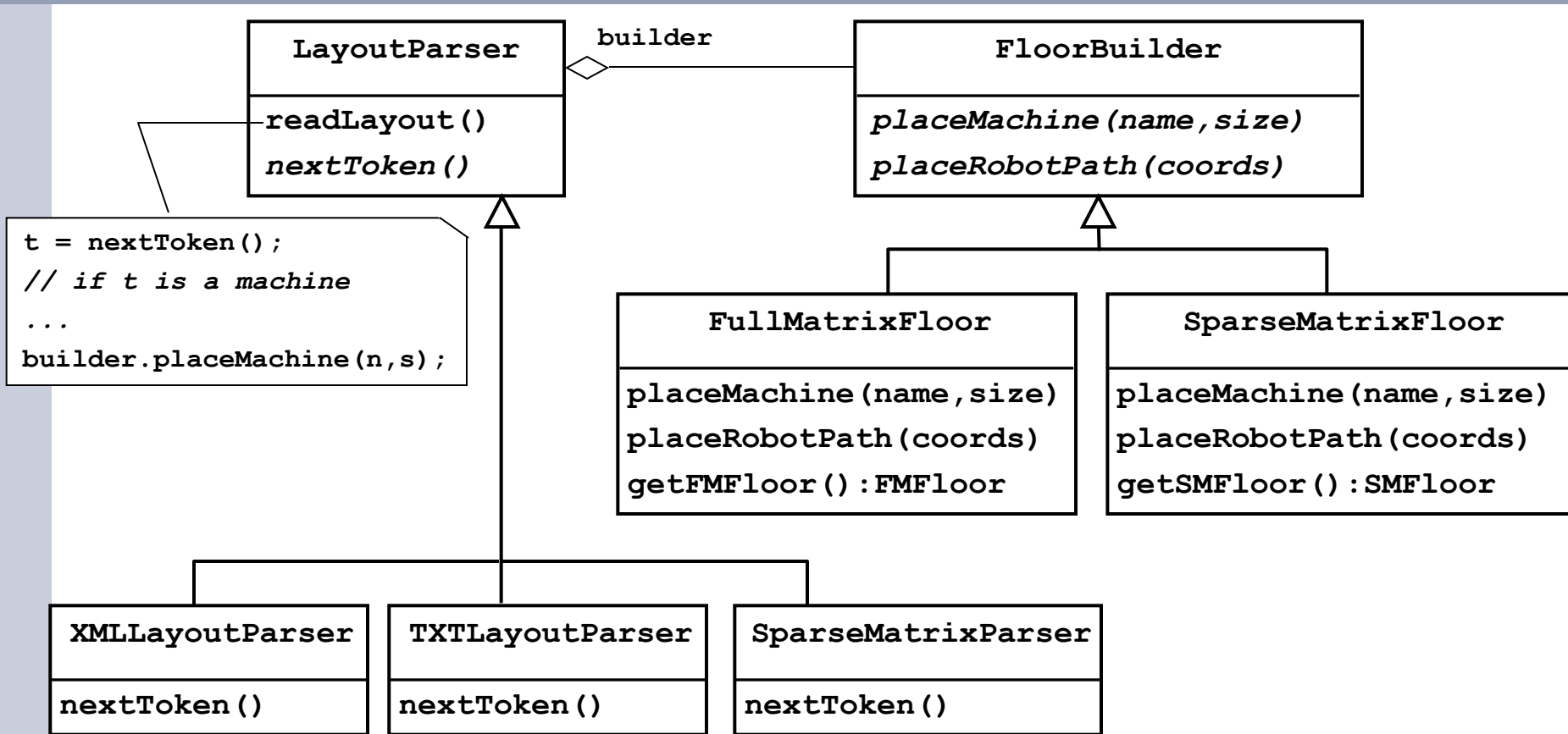
The Builder Design Pattern: Structure



Used when

- The construction of a product is independent of the parts
- Different representations for the product can be employed

The Builder Design Pattern: Example (I)



Hook Method and Hook Object
with the same template method

The Builder Design Pattern: Example (II)

- Usage:

```
SMFloor sparse_floor;  
FloorBuilder sm_builder = new SparseMatrixFloor();  
LayoutParser layIn = new XMLLayoutParser(sm_builder, xml_file);  
layIn.readLayout();  
sparse_floor=sm_builder.getSMFloor();
```

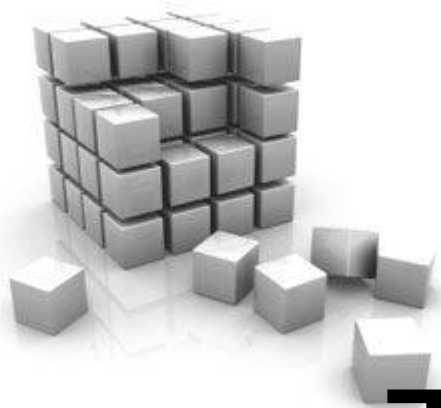
- Change of built object at runtime:

```
FMFloor full_floor;  
FloorBuilder fm_builder = new FullMatrixFloor();  
LayoutParser layIn = new SparseMatrixParser(fm_builder,  
                                             sparse_floor);  
layIn.readLayout();  
full_floor = fm_builder.getFMFloor();
```

Modularization and Software Architectures

Contents

- The notion of software module
- Desired characteristics of modules
- Specification of modules (ADS, ADT)
- Description of software architectures
- Analysis of software architectures



The Software Module

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.

Module as a means of abstraction

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

The module provides mechanisms for:

- (1) structuring of a program into independent units;*
- (2) 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.

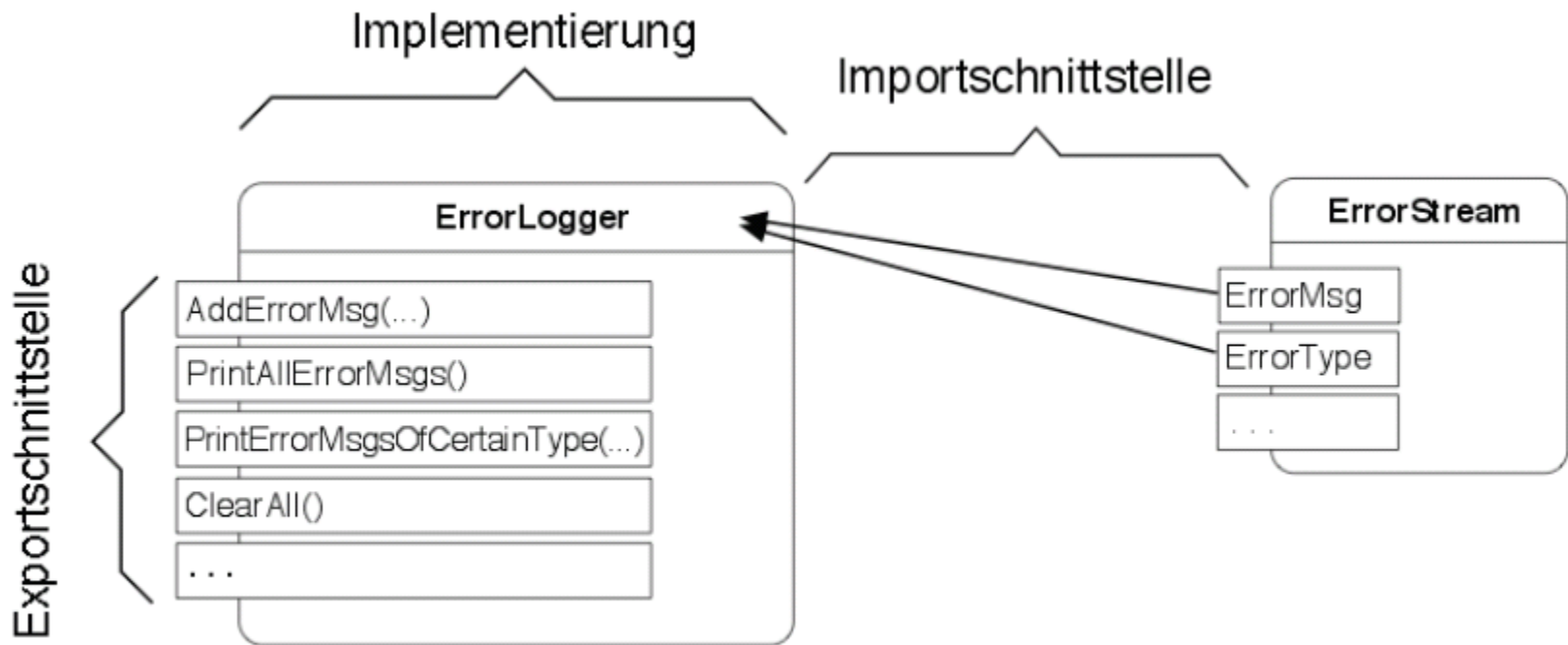
Module as expression means for modeling

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 system of a PC,
- the interaction elements of a graphic user interface.

Example: Module for storing error messages



Module interface of ErrorLogger (in Modula-2)

```
DEFINITION MODULE ErrorLogger;  
  FROM ErrorStream  
  IMPORT ErrorMessage, ErrorType; /* Import interface */  
  PROCEDURE AddErrorMessage(↓em: ErrorMessage);  
  PROCEDURE PrintAllErrorMessages();  
  PROCEDURE PrintErrorMessagesOfCertainType(↓et: ErrorType);  
  PROCEDURE ClearAll();  
  . . .  
END ErrorLogger.
```

Module implementation of ErrorLogger (in Modula-2)

```
IMPLEMENTATION MODULE ErrorLogger;
```

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

```
PROCEDURE AddErrorMsg(↓em: ErrorMsg)
```

```
  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

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.

Desired Characteristics of Modules

Stable and Understandable Module Interfaces by Information Hiding (I)

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

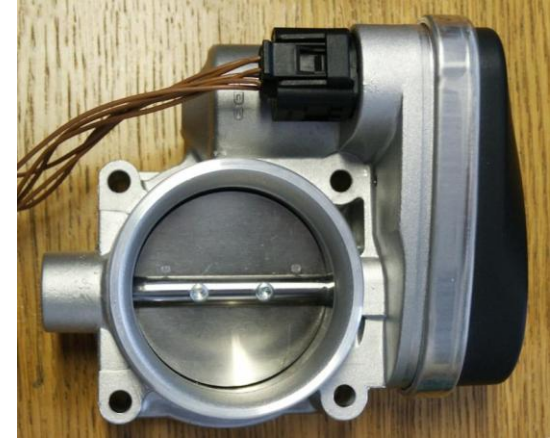
Thus, modules are to be designed in such a way that the data structures are hidden from the user.

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

Stable and Understandable Module Interfaces by Information Hiding (II)

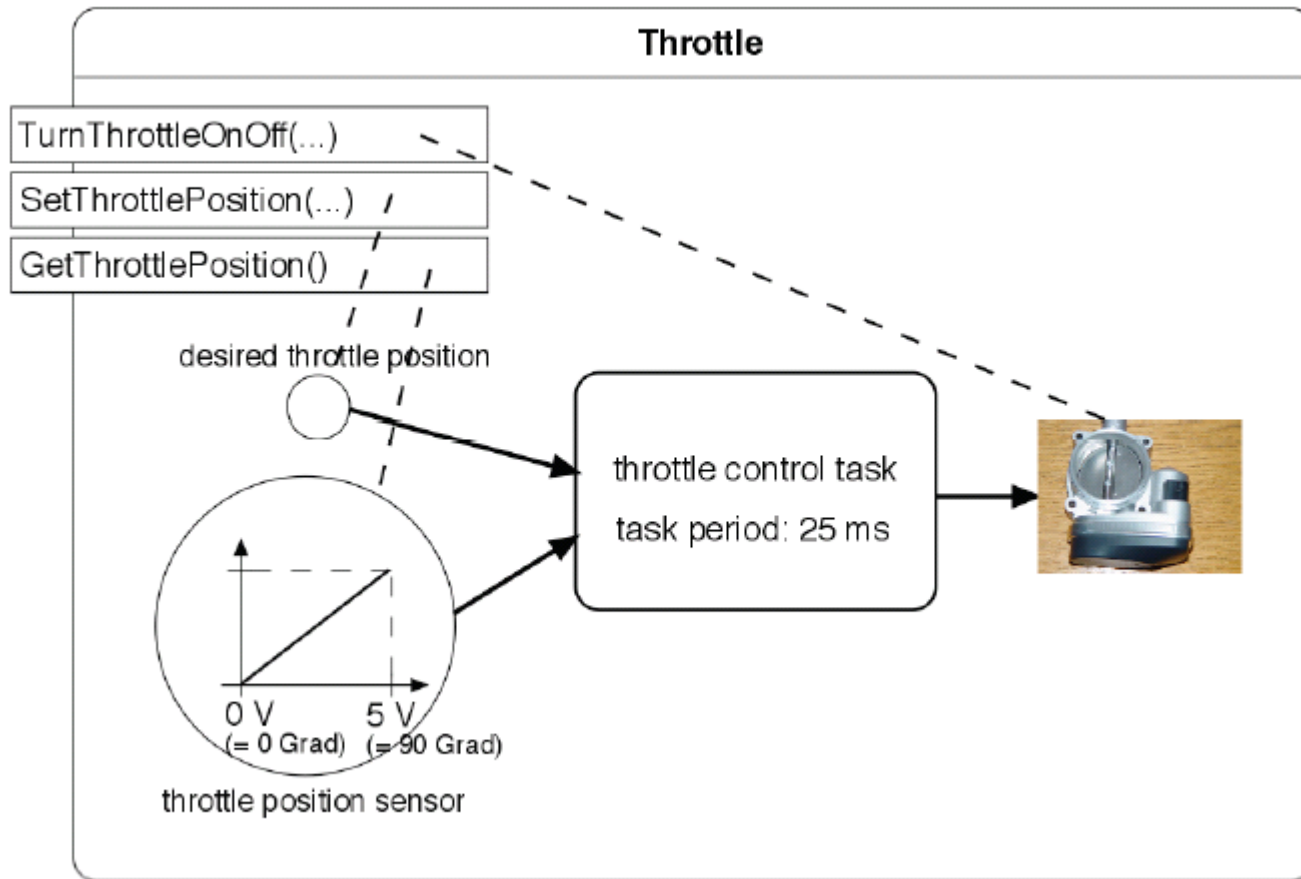
- A generalization of the information hiding principle is the requirement that in the design of modules one takes care to hide as many details of the implementation as possible behind the module interface, in order not to confront the user of a module with unnecessary details and complexity.
- This can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

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



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


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



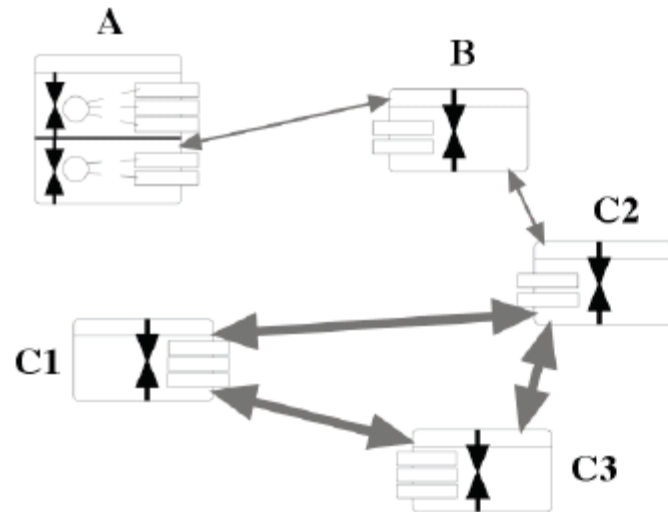
Balance between coupling and cohesion (I)

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

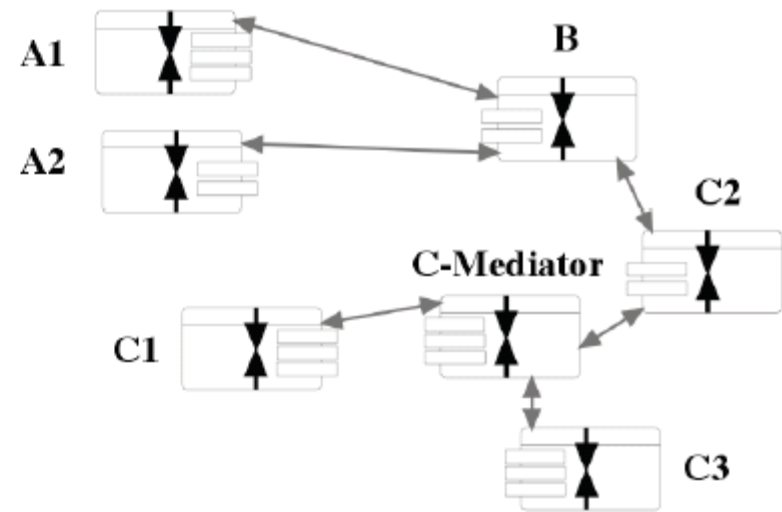
Balance between coupling and cohesion (II)

- 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 cohesion (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 cohesion in this setup.

Example for improved modularity



(a)



(b)

Legende:



starke Kopplung



schwache Kopplung



starke Kohäsion

Rules for cohesion 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 cohesion.
- **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 .

Rules for cohesion 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**

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.

Evaluation of modularization quality (I)

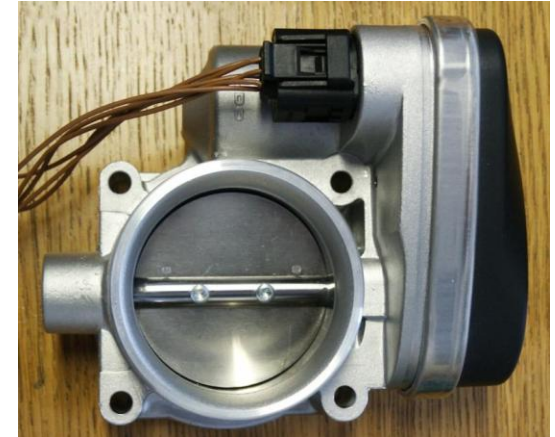
- The software architecture analysis method (SAAM) aims to examine the coupling and cohesion for low complexity systems.
- There is no generally valid metric that can evaluate the two characteristics objectively.

Evaluation of modularization quality (II)

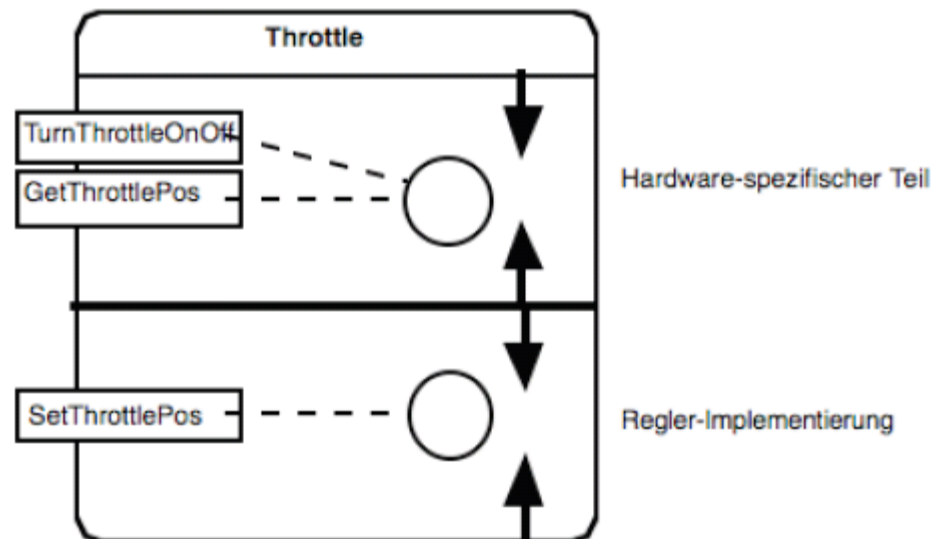
- The correct balance of coupling and cohesion in the structuring of software is therefore an art, which requires high qualification and much experience.
- As in Architecture, examples can help to create awareness and feeling for good modularity.
- Contrary to Architecture, unfortunately only few good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

Improving Cohesion in the Butterfly Valve Example

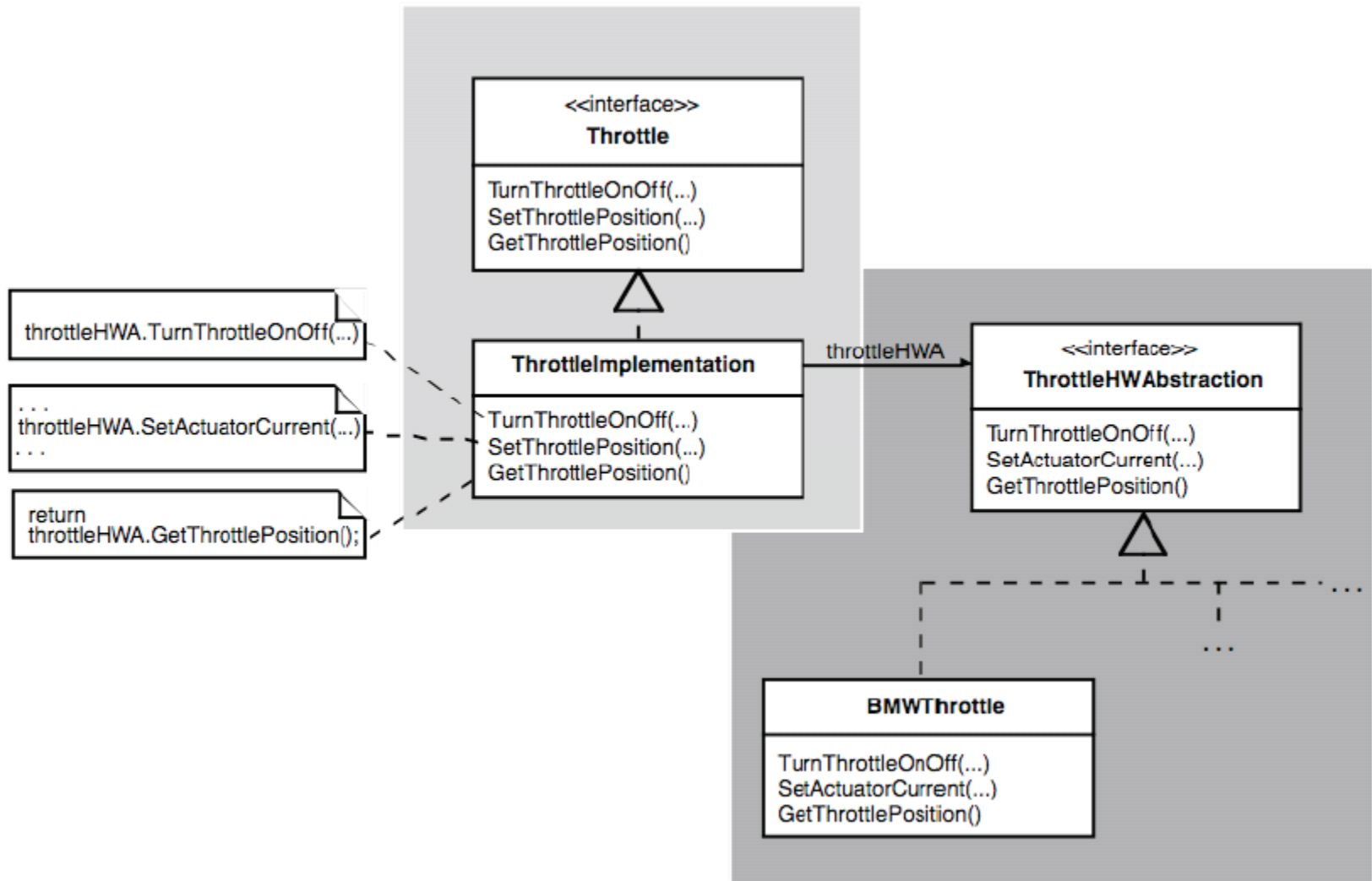
```
interface Throttle {  
    bool TurnThrottleOnOff(bool onOff);  
    bool SetThrottlePosition(float angle); // 0..90 Grad  
    float GetThrottlePosition();  
}
```



Low cohesion:



Split up the module Throttle while maintaining the interface



Module Specification

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.
```

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);
```

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();
        }

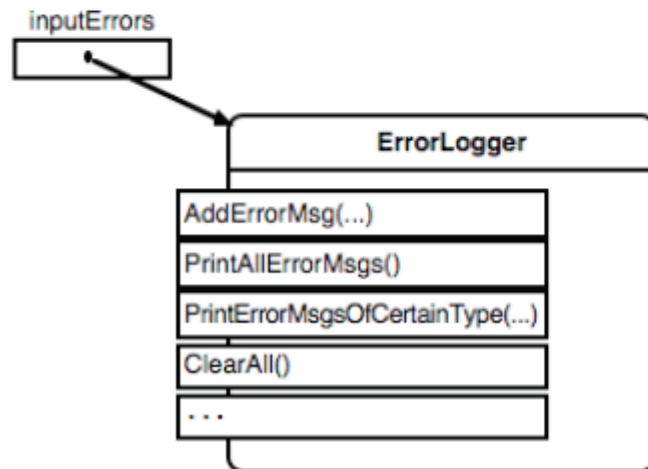
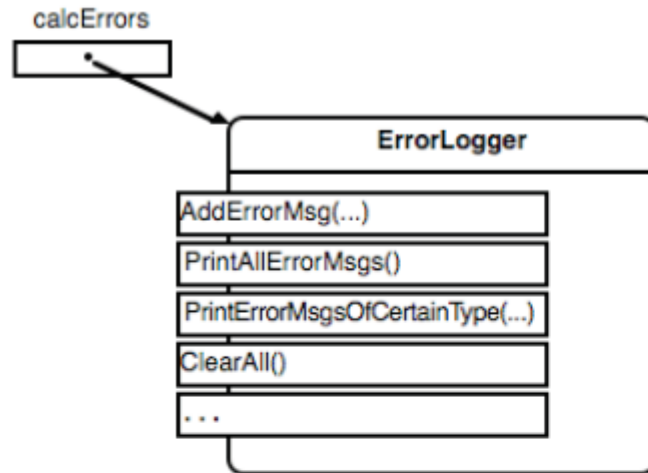
        public void AddErrorMsg(ErrorMsg em) {
            errorList.Add(em);
        }
        ...
    }
}
```

*Interface and implementation in
the same file!*

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  
}
```

ADT in OO Languages III



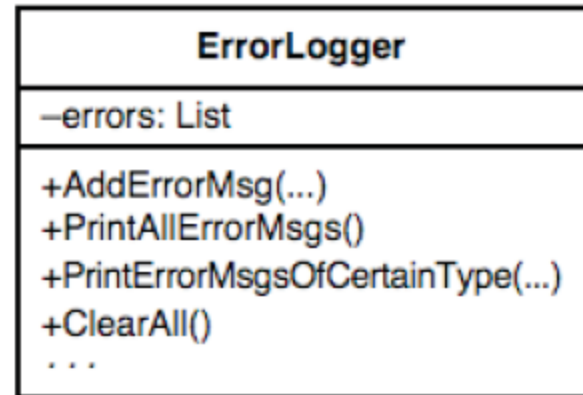
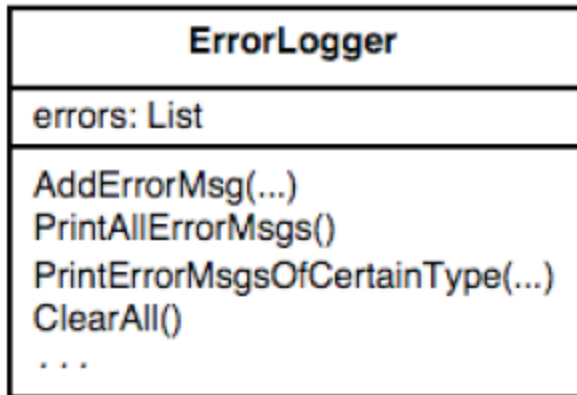
in UML:

calcErrors: ErrorLogger

inputErrors: ErrorLogger

ADT in OO Languages IV

as UML class diagram:



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.

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)

Description of Software Architectures

Definition of Software Architecture

The assembly of all the components (modules) of a software system together with their interactions.

Architectural styles

- In the 90s, The Software Engineering Institute (SEI) of the Carnegie Mellon University in Pittsburgh, Pennsylvania, considerably contributed to the establishment of **architectural styles** for the description of software architectures.
- SEI originally suggested a dedicated notation for architecture description; since 2003, SEI has used also the UML for that.

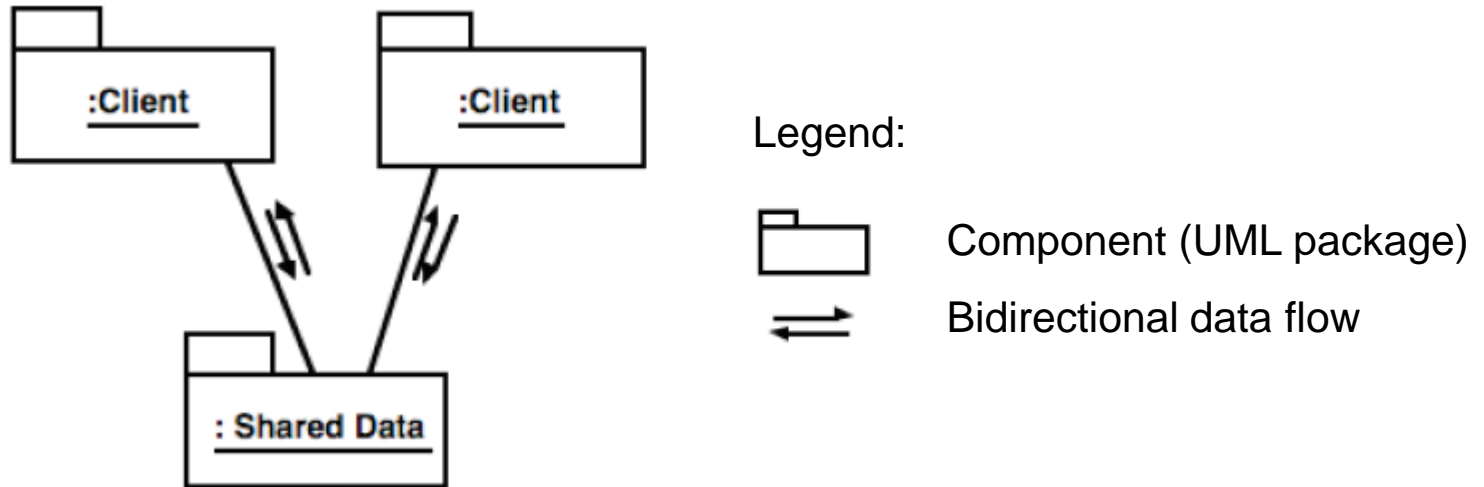
Examples of well described software architectures

- *Project Oberon—The Design of an Operating System and Compiler* by Wirth and Reiser (Addison-Wesley 1992).
The informal description is supplemented by schematic representations, screenshots, and source text.
- *Design Patterns* of Gamma et al. (Addison-Wesley 1995)

Overview of SEI architectural styles

Architectural style	Characteristics
Data-centered	Repository Architecture Blackboard Architecture
Data-flow	Batch/Sequential Architecture Pipes&Filters Architecture
Call & Return	Top-Down Architecture Network Architecture (Object oriented) Layered Architecture
Virtual Machine	Interpreter Architecture Rule-based Architecture
Independent Components	Event-driven Architecture

Data-centered (I)



- In a Repository architecture the data is passive.
- A Blackboard architecture has quasi-active data, which informs the clients interested in changes. The Blackboard architecture style is similar to the Observer design pattern (Gamma et al., 1995).

Data-centered (II)

Advantage:

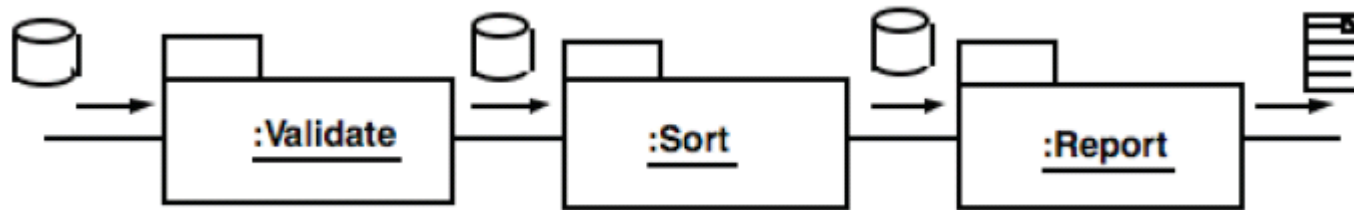
- Clients are independent from each other. Thus, a client can be changed, without affecting the others. Also further clients can be added.
- This advantage pales if the architecture is changed in such a way that clients are coupled closely (thus deviating from the recommended architecture style), for example in order to improve the performance of the system.

Data-centered (III)

Issues that must be addressed:

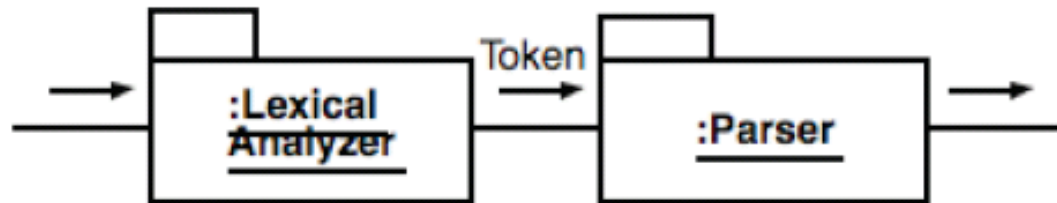
- Data consistency - synchronization of read/write operations
- Data security, access control
- Single point of failure

Data-flow style: Batch/Sequential



- The style describes a succession of transformations of input data.
- Data flow-oriented architecture parts are particularly characterized by reusability and modifiability.
- In the Batch/Sequential form, each transformation procedure must be terminated before the next one begins.

Data-flow style: Pipes&Filters



- In the Pipes&Filters form, data is incrementally transformed. That is, the data is divided into smaller units and these units are processed by filters.
- Pipes are stateless and transport the data from filter to filter in such a way that each filter autonomously determines when it needs the next element (input) of the data stream from the preceding filter.
- The difference between Pipes&Filters and Batch/Sequential is not evident in a UML representation.

Data-flow style: Advantages and Disadvantages

- The main advantage of data-flow is the low complexity of interactions between components. The processing modules are black boxes.
- The data-flow-oriented architecture style is unsuitable for modeling interactive applications.
- A further disadvantage is the frequently insufficient performance and efficiency. If filters need the entire input stream as context, appropriate buffers must be used. That affects the memory efficiency negatively.
- The data-flow style is well suited for modeling physical systems. It is used for example in the tool Simulink (from MathWorks).