Desired Characteristics of Modules
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 specified in the module interface.
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)
Module coupling and cohesion

- 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

- 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 even talk about cohesion in this setup.
Example for improved modularity

Legend:
- 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 use 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

```plaintext
DEFINITION MODULE ErrorLogger;
  FROM ErrorStream
    IMPORT ErrorMsg, ErrorType; /* Import interface */
  PROCEDURE AddErrorMsg(em: ErrorMsg);
  PROCEDURE PrintAllErrorMsgs();
  PROCEDURE PrintErrorMsgsOfCertainType(et: ErrorType);
  PROCEDURE ClearAll();
  . . .
END ErrorLogger.
```
A module is defined as a type and one can form as many instances of it as desired.

In Modula-2: Module ErrorLoggers with \textit{opaque type} ErrorLogger.

DEFINITION MODULE ErrorLoggers;

\begin{verbatim}
FROM ErrorStream IMPORT ErrorMsg, ErrorType;

TYPE ErrorLogger;

PROCEDURE NewErrorLogger(VAR el: ErrorLogger);
PROCEDURE AddErrorMsg(VAR el: ErrorLogger, VAR em: ErrorMsg);
PROCEDURE PrintAllErrorMsgs(VAR el: ErrorLogger);
PROCEDURE PrintErrorMsgsOfCertainType(VAR el: ErrorLogger, VAR et: ErrorType);
PROCEDURE ClearAll(VAR el: ErrorLogger);

\ldots

END ErrorLoggers.
\end{verbatim}
Module as Abstract Data Type (ADT) II

Producing instances with `NewErrorLogger`:

```plaintext
calcErrors, inputErrors: ErrorLogger;
NewErrorLogger(↑calcErrors);
NewErrorLogger(↑inputErrors);
AddErrorMsg(↓↑calcErrors, ↓any message);
...
PrintAllErrorMsgs(↓calcErrors);
PrintAllErrorMsgs(↓inputErrors);
```
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);
        }
        ...
    }
}
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
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 namespaces are means to group classes into units.
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++ (e.g., IDL module maps to C++ namespace)
  - 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

  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

<table>
<thead>
<tr>
<th>Architectural style</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-centered</td>
<td>Repository Architecture</td>
</tr>
<tr>
<td></td>
<td>Blackboard Architecture</td>
</tr>
<tr>
<td>Data-flow</td>
<td>Batch/Sequential Architecture</td>
</tr>
<tr>
<td></td>
<td>Pipes&amp;Filters Architecture</td>
</tr>
<tr>
<td>Call &amp; Return</td>
<td>Top-Down Architecture</td>
</tr>
<tr>
<td></td>
<td>Network Architecture (Object oriented)</td>
</tr>
<tr>
<td></td>
<td>Layered Architecture</td>
</tr>
<tr>
<td>Virtual Machine</td>
<td>Interpreter Architecture</td>
</tr>
<tr>
<td></td>
<td>Rule-based Architecture</td>
</tr>
<tr>
<td>Independent Components</td>
<td>Event-driven Architecture</td>
</tr>
</tbody>
</table>
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.
In Pipes&Filters, data is incrementally transformed: it is divided into smaller units which 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 tools like Simulink (from MathWorks) and Ptolemy II (UC Berkeley).
The Call&Return style is used for describing typical control flow for imperative programming: Procedures, functions and methods of a module are called from other modules and upon finishing their execution the control jumps back to (immediately after) the calling place.

Top-down form of the Call&Return architectural style: In conventional, not object-oriented implementations, the Call&Return leads to a top-down-oriented architecture, i.e. a (main or root) procedure/function/method calls further procedures/functions/methods, etc.
In network and/or object-oriented implementations of the Call&Return architectural style:

The constructs available in object-oriented languages allow the formation of network-oriented architectures alongside with the hierarchical structuring of top-down-oriented architectures. Method calls take place in a network of objects.
A further form of the Call&Return architectural style is the so-called **layered architecture**, which is used in order to introduce abstractions.

A layer corresponds to a module which offers a certain functionality and which can contain a number of other modules.

Each layer exposes an interface to be used by the above layer and uses an interface implemented by the layer below.
Example: Layer architecture of AUTOSAR (Automotive Open Systems Architecture)

Application layer

Runtime Environment

Services Layer

ECU Abstraction Layer

Microcontroller Abstraction Layer

Complex Device Drivers

Microcontroller
Advantages of the layered architecture

- Abstractization
- Ease of change: a layer interacts only with adjacent layers
- Information hiding
- Standardized layer interfaces for libraries and frameworks
- Enables service-oriented applications
Issues to be addressed in layered architectures

- Determining the right abstractization level
- Avoiding multi-layer crossing
- Performance
A virtual machine serves for providing functionality which is needed for the execution of an application, without making available details specific to the hardware and/or system software on which the application may run.

**Portability** is improved.
Virtual Machine (II)

- **Interpreter architecture**: The concept of the virtual machine has strengthened since the introduction of Java, becoming more commonly used. The fact that software portability can be improved by using virtual machines was already shown at the beginning of the 70s by the definition of the P-Code (Pascal code). Thus Pascal compilers, which produced P-Code instead of machine code, became portable. A virtual machine (which interpreted the P-Code), had to be made available for each hardware platform on which the compiled code had to be executed.

- **Rule-based architecture**: Expert systems, which work by interpreting rules (mostly of logical inference).
Independent Components (I)

- This architectural style postulates loose coupling between independent components. We call components independent, if the components do not directly call functions/procedures/methods of other components.
- A usual way to loosely connect independent components is by so-called **event-oriented linkages**: A component registers itself with another component, from which it wants to be informed about changes. When a change (event) occurs, a component informs all its registered components. The coupling is loose, because the component only informs other components about a change, without specifying what they have to do. They may or may not react to the change.
Independent Components Example (I)
Independent Components Example (II)