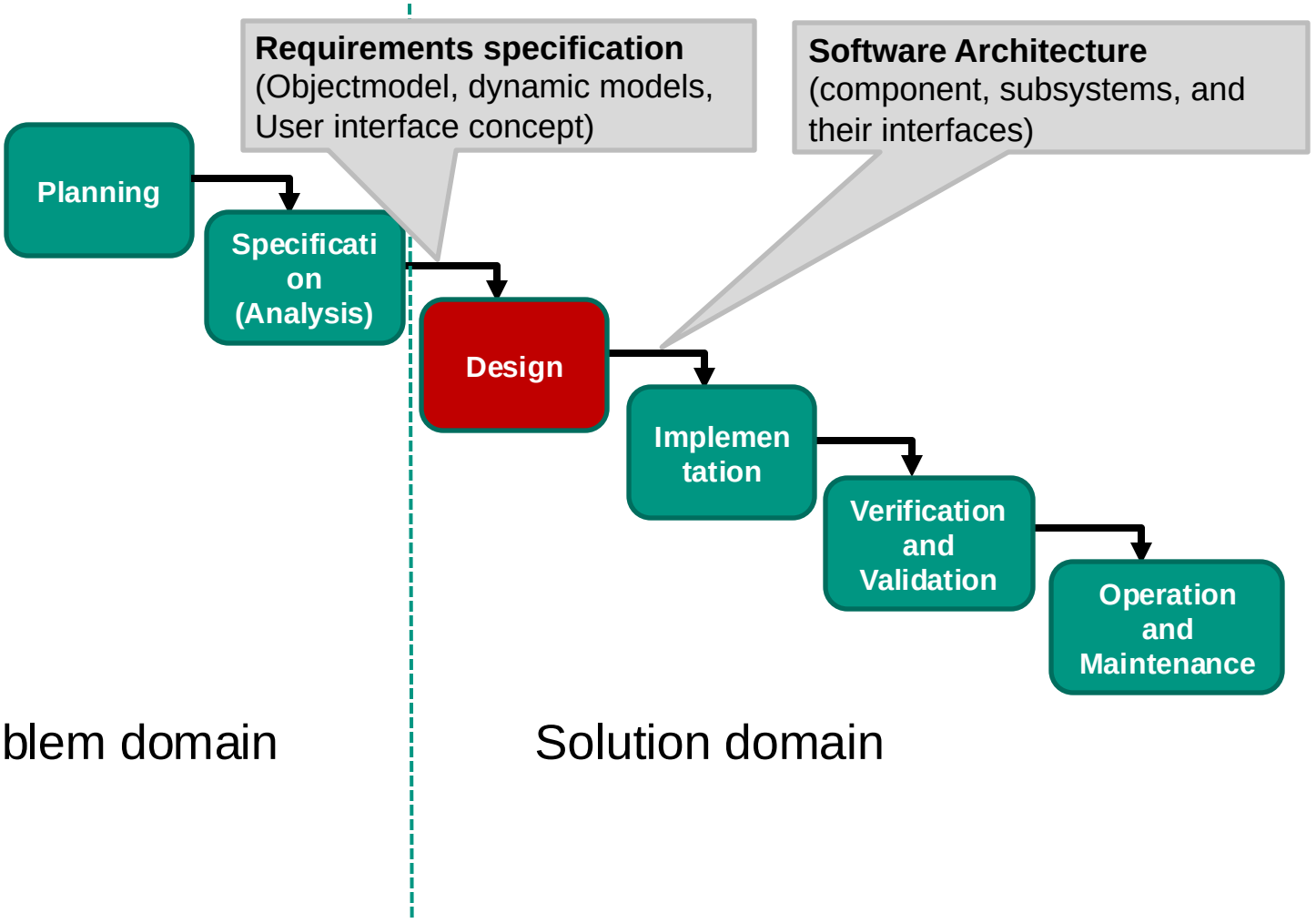
Introduction to Software Engineering

The Design Phase

Prof. Walter F. Tichy

ქუთაისის
საერთაშორისო
უნივერსიტეტი

Where are we?


Requirements specification
(Objectmodel, dynamic models, User interface concept)
Software Architecture
(component, subsystems, and their interfaces)
Planning
Specificati on (Analysis)
Design
Implemen tation
Verification and Validation
Operation and Maintenance
Problem domain
Solution domain

Learning goals

- Define the design phase and its role in the development process
- Produce a modular design for a given specification
- Produce an object-oriented design for a given specification
- describe, compare, recognize, and apply architectural patterns and design patterns.

What is Software Design?

- **Software design phase:**
  Transforming requirements specifications into a software design or software architecture.

- **Software design or software architecture consists of:**
  1. Hierarchical subdivision of software into components (modules and subsystems or packages)
     - Satisfies functional and non-functional requirements
  2. Specification of the function and the interfaces of components
  3. Specification of relations among components (uses-relation).
  4. Optional: Detailed Design (data structures and pseudo-code)

- Design activities are also called “programming-in-the-large”.

- Implementation is called “programming-in-the-small”.

The phases planning and specification were in the problem domain. The questions were: What is the problem, and what are we building?

In the design phase, we are entering the solution domain. The questions are: How are we solving the problem, how are we building what we’re supposed to build?

What is Software Design?

There are two major design methods:

1. Modular design
2. Object-oriented design

Object-oriented design extends modular design with inheritance, polymorphism, and data modeling (Entity-relationship) concepts.

Modular Design

A modular design consists of the following:

1. **Module Guide or Component Guide or Software Architecture:**
   - Structure of the system consisting of modules and subsystems (components, packages)
   - Functions of each module and subsystem
   - Uses architectural and design patterns such as software layers, pipes-and-filters, Model-view-controller, and others.

2. **Interfaces**:

Exact description of the interface elements (subprograms or functions or methods, types, variables, macros, entry points, etc.)

For modules with I/O, the interface description includes the exact specification of the I/O formats.

3. Uses relation:

Describes for every module which other modules or subsystems it uses, i.e., needs for correct operation.

The uses relation should be free of cycles, to enable incremental build and test.

4. Detailed Design (optional):

Specification of internal algorithms and data structures.

If a module will be implemented in assembler, then detailed design provides pseudo code for the entire module.

Pseudo code is written in an imaginary high-level programming language, for instance a language with the control structures of Java (for, if, while, function calls), but with statements in natural language. The pseudo code is translated into assembly code in the implementation phase. This is typical for embedded systems.

What is a Module?

The concept of a module is obviously important. We will use the concept of D. Parnas.

Modules should be developed and used independently of each other:

- It should be possible to *implement and test* modules independent of their later use.
- It should be possible to implement a module *without knowing anything* about the internal details of other modules or to influence the behavior of other modules.
- It should be possible to *use* a module *without knowing anything about its internal details.*

What is a Module?

A typical module consists of functions or subprograms that modify one or more data structures; other modules cannot directly access these data structures, except by using subprograms provided by the interface of the module (data encapsulation; this notion is also used in object-oriented design).

Within a module, there should be strong cohesion among data structures and subprograms; the cohesion among modules should be low.

A module should also be small and simple enough to be understood completely.

Module Concept

Module:
A module is a set of software elements that are developed and changed **together** according to the *information hiding principle*.

Software elements are types, classes, constants, variables, data structures, subprograms, functions, lambdas, processes, threads, entry points, macros, exceptions, etc., anything a progr. language can define.

Information Hiding Principle

(Encapsulation or Black Box Principle)

Each module hides an important design decision behind an **interface that will not change** if the design decision changes.

(David Parnas)

Reason: details that are likely to change should not affect the interface. Only aspects that are hidden can be changed without affecting other modules.

Example: Class Line,
without encapsulation or information hiding

- Line: start and end point, fixed length

```
public class LineNoSecret {
  public int startpoint_x;
  public int startpoint_y;
  public int endpoint_x;
  public int endpoint_y;
}
```

- Clients of this class must understand the class down to every detail.
- If we change to a different representation (length and angle, for example), all client code must be adapted.

Example: Class Line (2), with encapsulation, first try

```
public class LineWithSecret {
  private int startpoint_x; private int startpoint_y;
  private int endpoint_x;   private int endpoint_y;

  public int getStartpoint_x() { … }
  public void setStartpoint_x(int startpoint_x) { … }

  public int getStartpoint_y() { … }
  public void setStartpoint_y(int startpoint_y) { … }

   …… // 2 more getters and 2 more setters
}
```

- Direct access to attributes prevented.
- Semantic encapsulation?

Example: Class Line (3), with encapsulation, second try

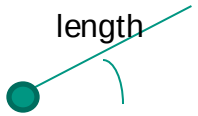
```
public class LineWithSecret {
  private Point startpoint;
  private Point endpoint;

  public Point getStartpoint() { return startpoint; }
  public void setStartpoint(Point startpoint) { … }
  public double getLength() {
    int xdist = endpoint.getX() - startpoint.getX();
    int ydist = endpoint.getY() - startpoint.getY();
    return Math.sqrt(xdist * xdist + ydist * ydist);
  }
}
```

Class Point encapsulates coordinates.

- Semantic encapsulation: internal representation of a line and implementation of length are hidden.

Example: Class Line (4)
with encapsulation, change of internals




```
public class LineWithSecret {
  private Point startpoint;
  private double length, angle; // polar representation

  public point getEndpoint() {
    // compute endpoint from length and angle
    … }
  public void setEndpoint(point endpoint) {
    // compute length and angle from endpoint
    … }
  public double getLength() {
    return length;
  }
}
```

Internal representation changed, and client code need not be adapted. Change is limited to a single class!

Module Concept

- Anticipated changes should be possible without changing interfaces or client programs.
- Less predictable changes may cause interface changes, but only in a few, rarely used modules.
- Only rarely should it be necessary to change the interfaces of frequently used modules (and client programs).

Case Study: KWIC Index

Design of a program for producing a KWIC Index (permuted Index; KWIC=*Keyword in context*)

Input:

- Sequence of lines („titles“)
- Each line is a sequence of words
- Each word is a sequence of characters

Output:

- Produce circular shifts out of each title (by repeatedly moving the first word of a title to the end).
- The circular shifts are output in sorted order.

KWIC Index

Example:

Suppose we have the following titles:

Harry Potter and the Order of the Phoenix
The Flight of the Phoenix
The Order of the Sword
Religious Orders and Monasteries

After disregarding the words “the”, “of”, “and”, the KWIC index is produced as follows:

KWIC Index, circular shifts

Harry Potter and the Order of the Phoenix
Potter and the Order of the Phoenix, Harry
Order of the Phoenix, Harry Potter and the
Phoenix, Harry Potter and the Order of the
Flight of the Phoenix, The
Phoenix, The Flight of the
Order of the Sword, The
Sword, The Order of the
Religious Orders and Monasteries
Orders and Monasteries, Religious
Monasteries, Religious Orders and

KWIC Index, circular shifts ordered alphabetically

Flight of the Phoenix, The
Harry Potter and the Order of the Phoenix
Monasteries, Religious Orders and
Order of the Phoenix, Harry Potter and the
Order of the Sword, The
Orders and Monasteries, Religious
Phoenix, Harry Potter and the Order of the
Phoenix, The Flight of the
Potter and the Order of the Phoenix, Harry
Religious Orders and Monasteries
Sword, The Order of the

KWIC Index, nicely formatted

The	Flight of the Phoenix
	Harry Potter and the Order of the Phoenix
Religious Orders and	Monasteries
Harry Potter and the	Order of the Phoenix
The	Order of the Sword
Religious	Orders and Monasteries
Harry Potter and the Order of the	Phoenix
The Flight of the	Phoenix
Harry	Potter and the Order of the Phoenix
	Religious Orders and Monasteries
The Order of the	Sword

A KWIC index is suitable for keyword search in a set of titles.
For each keyword, one can quickly find the titles that contain it.

KWIC Index

Another example:

Input: **„abs: integer absolute value“**

Circular shifts**:**

abs: integer absolute value

absolute value, abs: integer

integer absolute value, abs:

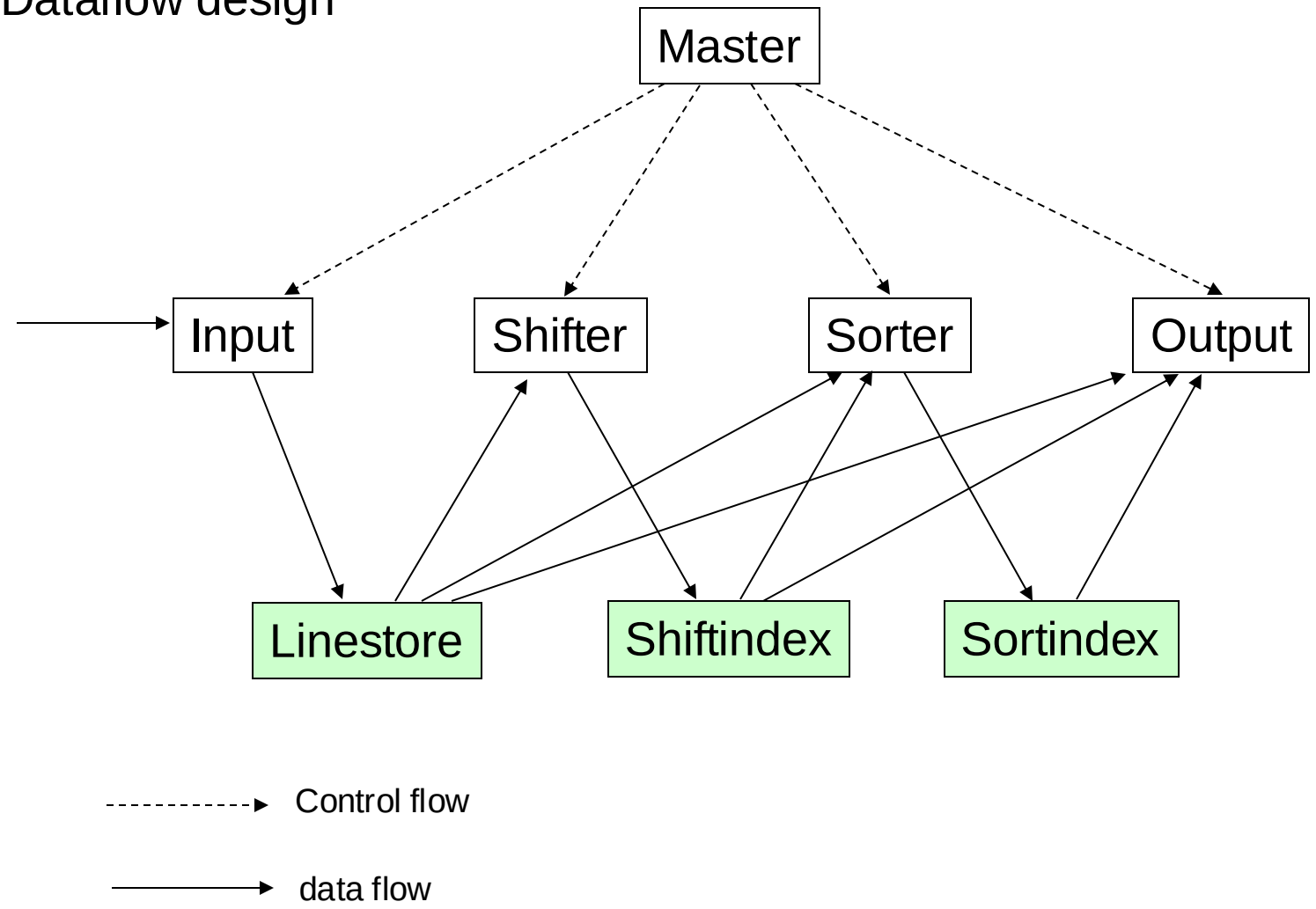
value, abs: integer absolute

formatted und sorted:

	abs: integer absolute value
abs: integer	**absolute value**
abs:	**integer absolute value**
abs: integer absolute	**value**

KWIC Index

Dataflow design

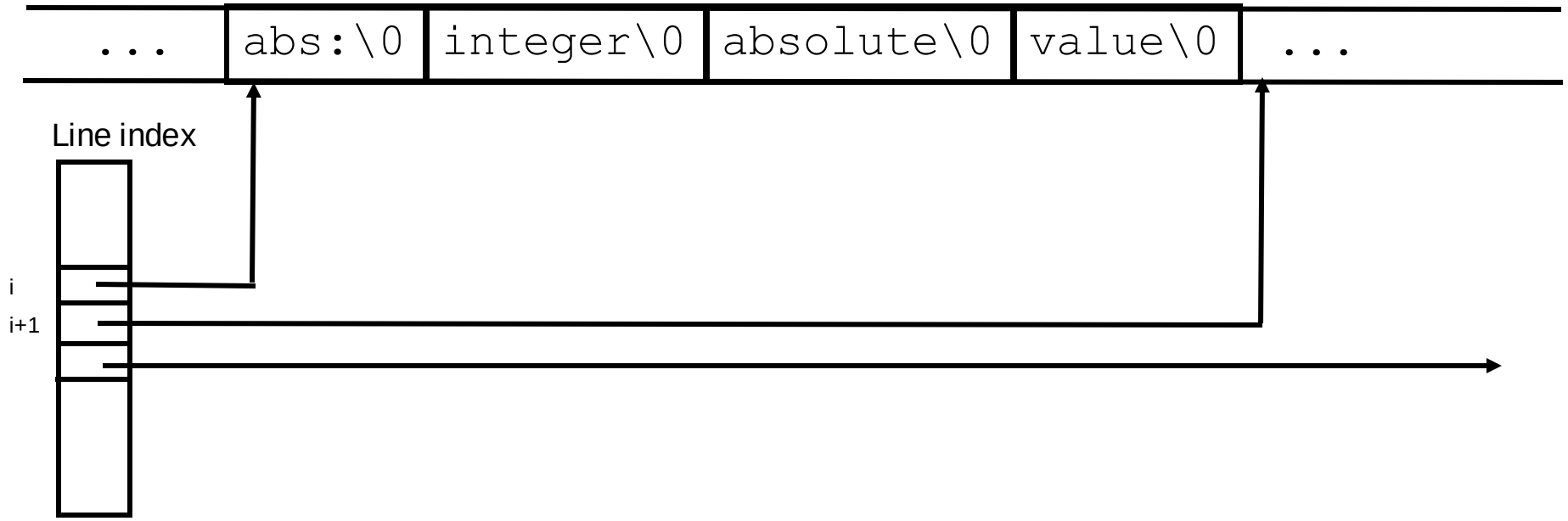


KWIC Index

Input:

- Reads lines from input and stores them in main memory
- Stores four characters per memory word; input words are terminated by special character.
- A line index marks the beginning of each line.

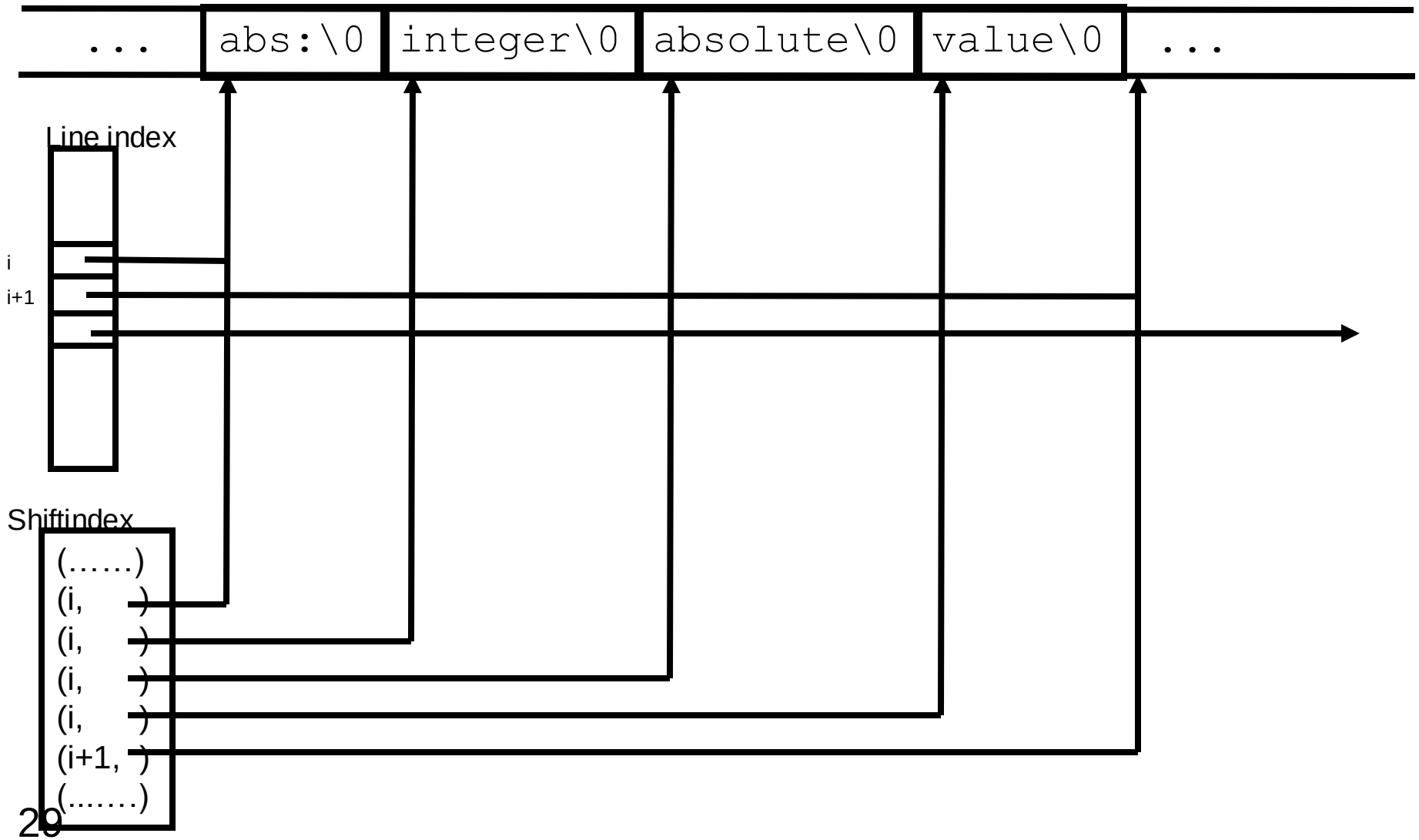
KWIC Index



Shifter:

- Is called after input has been completely read.
- Produces an index that contains all circular shifts.
- Each circular shift consists of (<lineNr>,<address>)
  - <lineNr> is the index of the input line
  - <address> is the address of the first character of the shift.

KWIC Index


...
abs:\0
integer\0
absolute\0
value\0
...
Line index
i
i+1
Shiftindex
(.....)
(i,)
(i,)
(i,)
(i,)
(i+1,)
(.....)

Sorter:

- Works on the data structures produced by the preceding two components.
- Produces a vector, in which the shift numbers are listed in sorted order (a permutation of shift index).

In the “abs:” example, there are four shifts:

1, 2, 3, 4

Sorted:

3, 1, 2, 4

Output:

- Operates on the data structures produced by input, shifter and sorter.
- Outputs the circular shifts in sorted order.

Master:

- Calls subprograms.
- Manages error messages.

(The exact specification of the interfaces is not provided.)

KWIC Index

This design would work.

Data structure is parsimonious (shifter produces no duplicates)

Which problems can you identify?

(Think of potential changes.)

KWIC Index

Important considerations, part 1:

The storage of the titles might change

- Main memory
- Disk
- Remote (on a server)
- mixed
- different character sets,
- different storage of character strings

KWIC Index

Important considerations, part 2:

The storage structure might change

- Packed/unpacked storage of character strings
- Storage of words on a heap, removal of duplicates
- Marking or deleting unimportant words such as *a, an, the, that, of,* etc.
- Different languages (differences in unimportant words).

KWIC Index

Important considerations, part 3:

The decision to produce an index for the circular shifts rather than producing the shifts directly might change.

KWIC Index

Important considerations, part 4:

The decision about sorting might change

- Search for entries as needed
- Sort partially (according to first letter, for instance)

The I/O formats may change

(ASCII, PS, PDF, HTML, XML, ...)

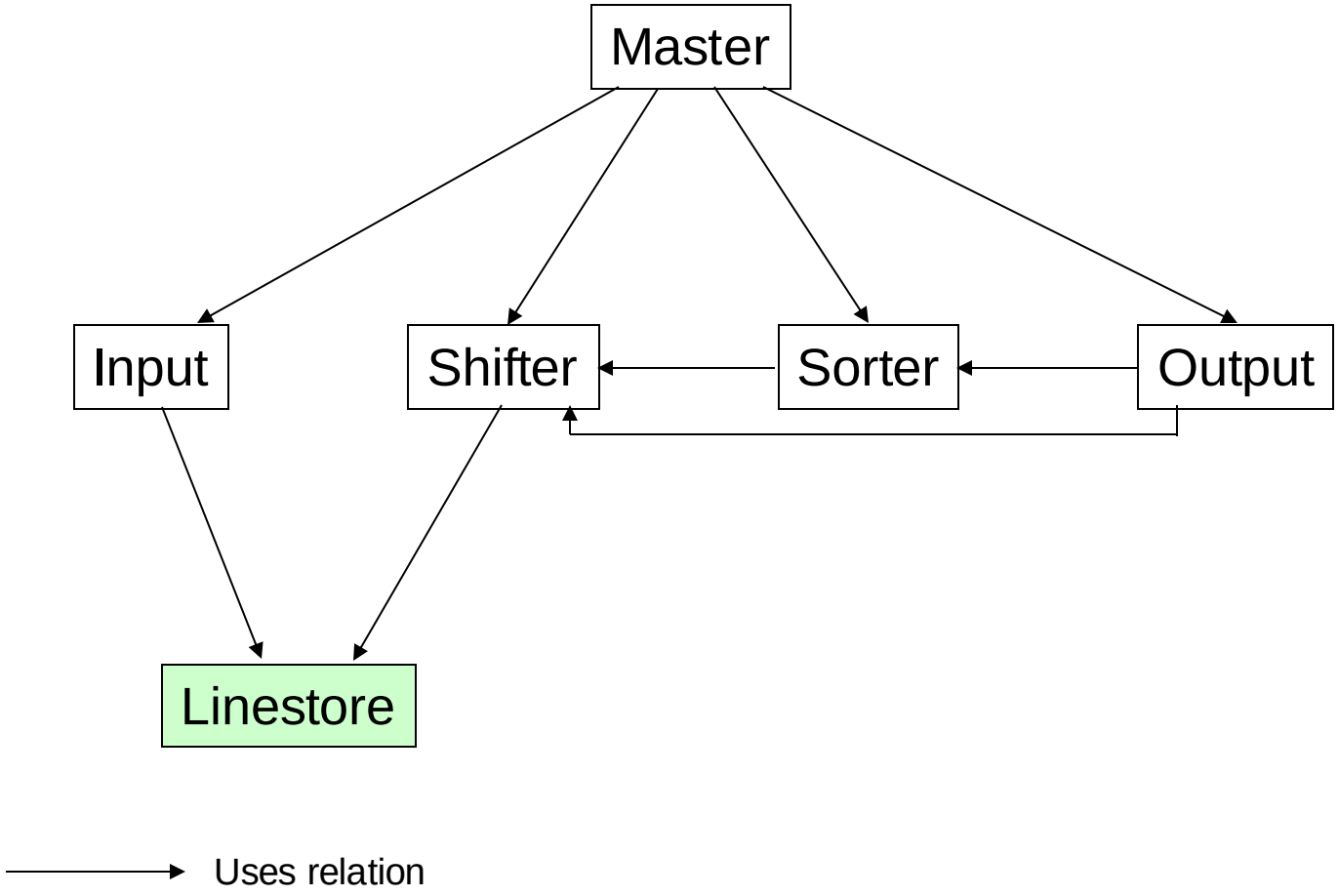
Input from data base rather than file

KWIC Index

- Except for the last two changes, all modules would have to be modified.
- There is too much information shared about data structures.
- How can this sharing be reduced?
- Answer: with programmatic interfaces rather than data structure interfaces.

KWIC Index

Information hiding design



Linestore: storage of lines

- Idea: rather than a data structure, offer access functions. This is simple, because titles, words in titles, and characters in words have an index. The internal data structure can be chosen by implementer.
- `getLineCount()` returns the number of lines stored in Linestore.
- `getWordCount(r)` returns the number of words in line r.
- `getCharCount(r,w)` returns the number of characters in word number w in line r.
- `getChar(r,w,c)` returns character number c in word number w in line r.
- `setChar(r,w,c,d)` sets character number c in word number w in line r to character d.

Output of all lines (tests suitability of the interface):

```
for (r = 0, r < getLineCount(), r++) {
    for (w = 0, w < getWordCount(r), w++) {
       for (c = 0, c < getCharCount(r,w), c++)
           printc(getChar(r,w,c));
       printc(' ');
    }
    printc('\n');
}
```

KWIC Index

- The representation of lines is now arbitrary, because it is hidden behind an interface that is insensitive to changes.
- The interface essentially provides three nested iterators to access the data (iteration over lines, words in lines, and characters in words), and to enter the lines.
- Exercise: define a data structure and algorithms for this interface.
- Exercise: write a program that fills Linestore with data from the input, using the interface (hint: think about when to increment and reset the counters for lines, words, and characters)
- How would you avoid storing duplicate words?

KWIC Index

Shifter:

- `ShInit()` initializes the shifter.
- `ShGetShiftCount()` returns the number of circular shifts.
- `ShGetWordCount(s)` returns the number of words in circular shift s.
- `ShGetCharCount(s,w)` returns the number of characters in word w of shift s.
- `ShGetChar(s,w,c)` returns character c in word w of shift s.

KWIC Index

Options for Shifter:

- produce an index
- produce the shifts directly (copying)
- produce shifts on demand.
- produce only the shifts for a given keyword.

KWIC Index

Sorter:

- `Init()` initializes the sorter.
- `getShift(i)` returns the number of the circular shift that is in the i´th position in the sorted list of circular shifts (permutation)

Options for Sorter:

`Init()` can sort completely or partially or not at all.

The sorted order can be produced on demand in `getShift`, piece by piece.

Or sort only the shifts for a given keyword.

KWIC Index

Input:

- `readLines()` reads lines from the input and stores them character by character into Linestore.

Output:

- `printLines()` prints all circular shifts in the order given by `getShift(i)`

- The changes discussed earlier can be implemented by changing only a single module.
- How would you handle stopwords such as the, a, an, of, this, that, etc.? They should not be included when producing shifts, but be printed nevertheless.

Documenting Software Architecture

The design is recorded in a software architecture document. It contains:

1. Module guide or software architecture

   For each module or subsystem it describes:

   - The design decision that is the secret of the module/subsystem
   - The function of the module/subsystem
   - The composition of subsystems out of modules and other subsystems.

Applying the information hiding principle:

1. Make a list of design decisions that are difficult (i.e., it is likely that they will have to be revised)

   or that are likely to change.

2. Assign each decision to a module and design module interfaces such that they will not change if the decisions change.

   This hides every decision behind an invariant interface.

Candidates for hiding (part 1):

- Implementation of data structures and the operations on them (*abstract data types)*
- *Example: design a module for storing and manipulating graphs. How many ways are there to represent graphs? How many variants of graphs are there? Design the interface independent of the data structures used.*
- Low-level hardware details (for example device drivers, character codes and their order, I/O control)

Candidates for hiding (part 2):

- Operating system details (for example I/O interfaces, file formats, network protocols, command languages)
- Basic libraries such as data bases, graphics packages, operating system specific libraries.
- (Of course, the major software producers want you to do the exact opposite, to lock you and your customers into their platforms)

Candidates for hiding (part 3):

- I/O formats (ASCII, binary, HTML, XML, ….)
- User interfaces (command line, graphical, free-hand writing, web page, speech…) — rest of system unaffected by changing or dynamically switching the interface.
- Text of dialogues, error messages, labels on buttons, help texts (prepare for localization for different countries/regions)
- Size of data structures
- Order of processing

Module Guide

A carefully developed module guide has the following advantages:

- Avoids duplicates
- Avoids gaps
- Supplies a subdivision of the system
- Simplifies finding affected modules during maintenance.

2. Module Interfaces

The result is a „black box“ description of each module.

- Contains exactly the information needed for using and implementing the module, and no more.
- The interface is invariant with respect to anticipated changes.

Module Interfaces

The interface description consists of:

1. List of available software elements,
2. Parameters for subprograms/methods/functions,
3. Effect of subprograms/methods/functions,
4. I/O formats (if an I/O module),
5. Runtime, precision, memory requirements, and other quality attributes where required,
6. Error situations (error return codes or exceptions) and how to resolve them,

Modular Design

How are modules expressed in the language C?

- C provides file inclusion; this means that files ending in „.h“ can be included in the compilation process of a file.
- The interface of a module is specified as public function headers and variables in a „.h“-file, for instance in *LineStore.h*, *Shifter.h,* and *Sorter.h*
- The implementation of a module is a „.c“-file, with an inclusion directive at the beginning. *Shifter.c* contains:

```
#include ´´Linestore.h´´
#include ´´Shifter.h´´
/* rest of module */
……
```

- Thus, the compiler has the information needed to access Linestore.c and can check that the interface of Shifter is implemented type-correctly. In languages such as Modula and Ada, the mechanism for expressing modules is more elaborate.

Program Families or Software Product Lines

A **program family** or a **software product line** is a set of programs that share and reuse a significant amount of requirements, design information, or software components.

New members of a program family can be created by reusing significant amounts of information, libraries and other components from the existing product line. It is thus significantly cheaper to create a new family member than to build it from scratch.

Program Families

- Goal:
- Exploit commonalities
- Reuse requirements, designs, specifications, components.
- Reduce development and maintenance costs.
- In practice, program families require a significant initial investment in analysis, design, and reusable components that is not recovered before the third family member.

Program Families

How do program family members differ?

Externally:

- They run on different hardware and operating systems
- They provide supersets or subsets of functionality, as not all users need all the functions or not all hardware and operating systems can support all functions.
  - e.g., home version, professional version, enterprise version
- They differ in user interfaces, language of messages, national differences such as holidays, tax codes, business rules, and other aspects.

Program Families

Internally:

- They may use different data structures or different algorithms because the amount of data to be processed, the available computing resources, and the relative frequency of certain events may differ.

Modular Design

Prof. Dr. David Lorge Parnas

*10.2.1941 in New York

- Inventor of the information hiding principle (1971)
- Formulated concepts of modular design (1972)
  - Specification of modules
  - Separation of sepcification and implementation
  - Modular software architecture
- Contributions to software engineering and quality of software (since 1975)

Bibliography Modular Design

„Designing Software for Ease of Extension and Contraction“, David L. Parnas, IEEE Trans. On Software Engineering, SE-5(2), March 1979.

„On the Criteria to be Used in Decomposing Systems into Modules“, D. L. Parnas, Communications of the ACM, 15(12), Dec. 1972.

„The Modular Structure of Complex Systems“, Davis L. Parnas, IEEE Trans. on Software Engineering, SE-11(3), March 1985.

„A Rational Design Process: How and Why to Fake it“, D. L. Parnas, P. C. Clements, IEEE Trans. on Software Engineering SE-12(2), Feb. 1986.

Object-Oriented Design uses the concepts of classes, methods, objects, inheritance, associations, etc. from object-oriented languages or UML to express software designs.

The principles of modular design (information hiding and change-invariant interfaces) continue to apply.

Object-oriented Design

- The analogue to module is the class or the package.
- In a package, several classes are combined that share common design decisions.
- A single class can represent an entire module; but in general, several classes are required for a module.
- The software architecture is documented in UML class and package diagrams, accompanied by specifications of the interfaces of classes.
- It is possible to describe interfaces without implementation, by using abstract classes or interface classes.

Object-oriented Design

There are additional capabilities in object-oriented design that go beyond the module concept:

- Multiple instantiation of classes (a module exists only once),
- Inheritance and polymorphism,
- Program variants by implementing the same interface several times in the same program,
- Data modeling with associations and cardinalities.

Object-oriented Design

Both modular and object-oriented design patterns exist.
Design patterns provide solutions to recurring design problems. Design patterns will also be discussed.

Literature: Bruegge, Dutoit, Chap 6. Read it!

Grady Booch

*27.2.1955 in Texas
Chief Scientist
Rational Software Corporation

Pioneer in modular and object-oriented design

- 1983: book *Software Engineering with Ada*
- 1987: book *Software Components with Ada*
- 1991/94: book *Object Oriented Design with Applications*

Pioneer in reusable components

- 1987: component library for Ada
- 1991: class library for C++.