Fakultas Informatika
School of Computing
Telkom University

CCH1D4
STRUKTUR DATA

Modularity and Data Abstraction

Modularity

- describes a program organized into loosely coupled, highly cohesive modules"
  - Carrano and Prichard, p. 7
- "a technique that keeps the complexity of a large program manageable by systematically controlling the interaction of its components"
  - Carrano and Prichard, p. 106

In Short :

- Modularity is the degree to which a system's components may be separated and recombined
- Here we will know about **Abstract Data Type** (ADT)
  - to understand better about ADT, let’s see the illustration

Student Information System

- Suppose we will make an information System to store students record
- There are 4 menus
  - Add new data
  - Delete data
  - Edit data
  - View stored data

What we usually do?

- Basic Algorithm Writing Style

Type student < … >	Type description of student, name, id, class, etc.
Dictionary …… function add_student(…) ……	Declaring Variables and name of functions that will be used in main program
Algorithm …… ……	Main program, contains menu, interface, etc. The main program will use the functions available or has been specified
Function add_student(…) …… Function delete_student(…) …… Procedure …… …… ……	Function specification, basic operation for student **All in one file (one script)**

Here, we actually have

- 3 parts of program or 3 modules

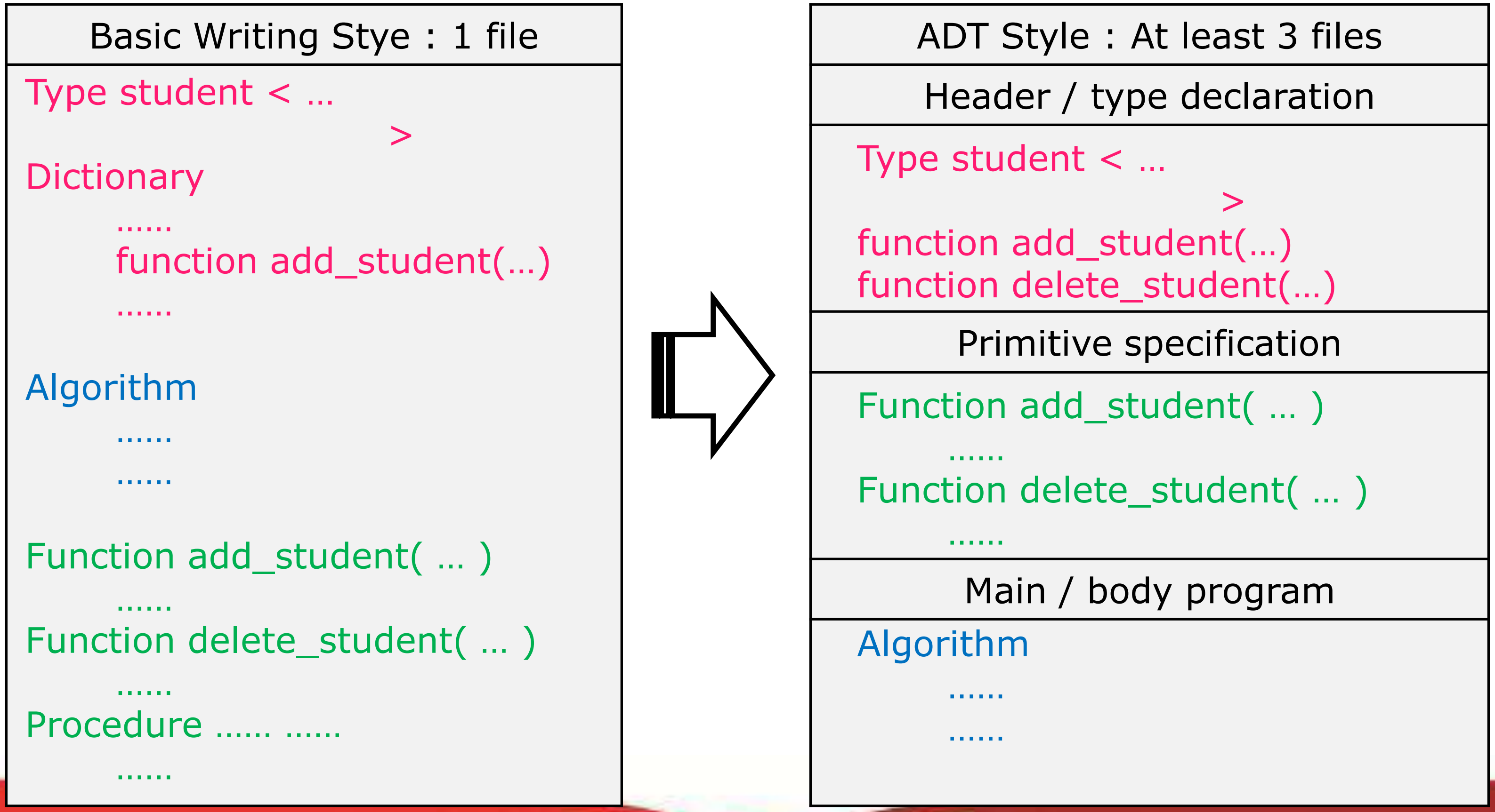
Type student < … >; Dictionary …… function add_student(…) ……	Type description and Primitive declaration [Specification]
Algorithm …… ……	Body of main program [Program implementation]
Function add_student(…) …… Function delete_student(…) …… Procedure …… …… ……	Primitive Implementation (basic operation)

Abstract Data Type

- The Goal of ADT is to **separates** between specification and implementation
- ADT itself is the Specification part that contains **TYPE** declaration and **PRIMITIVE** specification

Change the basic style

- ADT style Algorithm



Basic Writing Stye : 1 file

```
Type student < …
                >
Dictionary
        ……
        function add_student(…)
        ……

Algorithm
        ……
        ……

Function add_student( … )
        ……
Function delete_student( … )
        ……
Procedure …… ……
        ……
```

ADT Style : At least 3 files

Header / type declaration

```
Type student < …
                >
function add_student(…)
function delete_student(…)
```

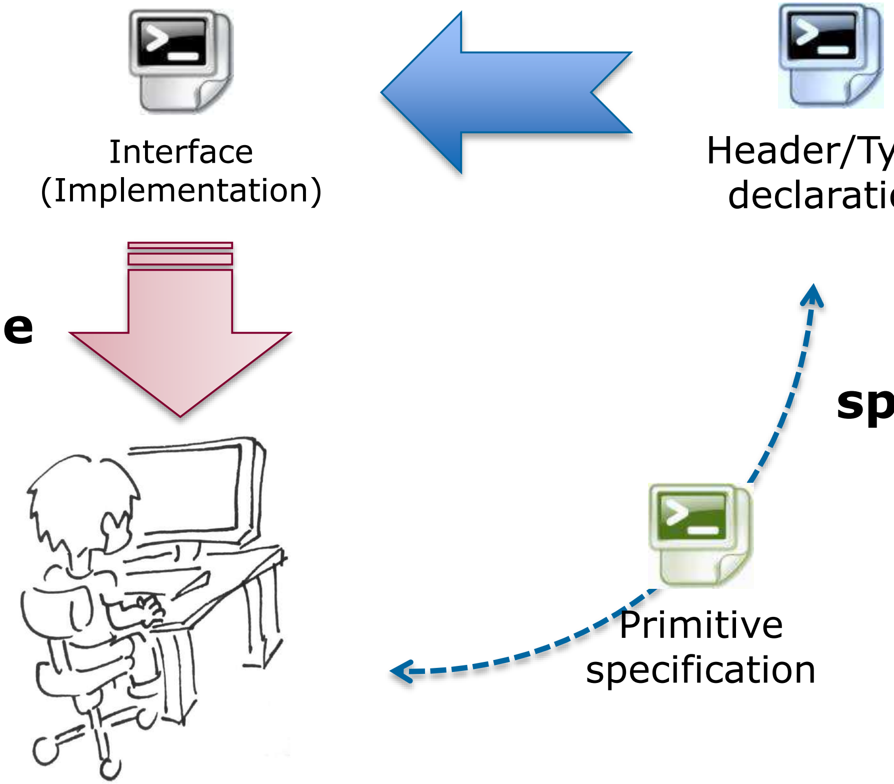
Primitive specification

```
Function add_student( … )
        ……
Function delete_student( … )
        ……
```

Main / body program

```
Algorithm
        ……
        ……
```

Connectivity Between Modules



implement

Interface
(Implementation)

Header/Type
declaration

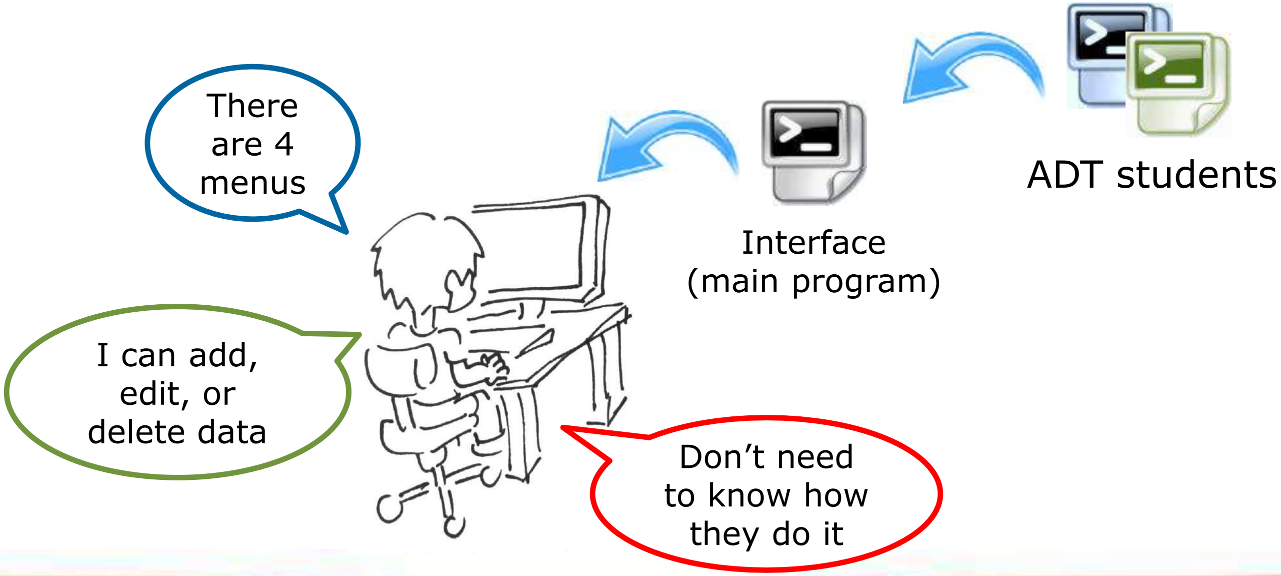
compile

specify

Primitive
specification

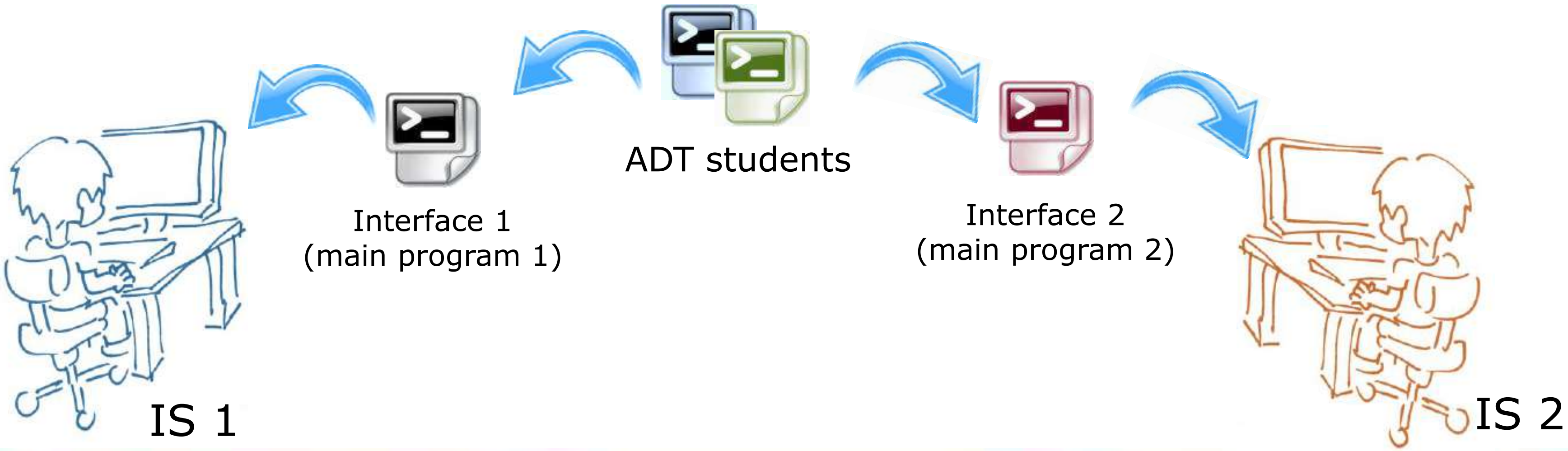
Why we use ADT?

- Security
  - user only needs to know the specifications of features without the need to be given detailed implementation of these features.


There are 4 menus
I can add, edit, or delete data
Don’t need to know how they do it
Interface (main program)
ADT students

Why we use ADT?

- Reusability
  - Suppose we’re going to build another information system that happens also use students record (add, edit, delete)
  - We don’t need to code the ADT again, we can use what we already have



Question?

DRY Principles

- Don't repeat yourself
- “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”
  - When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements

Task/Exercise : Clock ADT

- Create a Clock ADT (Clock.h) to store time (hour, minute, and second)

```
TYPE Hour       : integer {0..23}
          TYPE Minute    : integer {0..59}
          TYPE Second    : integer {0..59}
          TYPE Clock :
          <
               HH : Hour,
               MM : Minute,
               SS  : Second;
          >
```

Task/Exercise : Clock ADT

- Primitive for Clock.h
- Validator
  - Function **IsValid**(HH,MM,SS: integer) → boolean
  - { return true if 0≤HH≤23, and 0≤MM≤59, and 0≤MM≤59 }
- Constructor
  - Function **MakeClock**(HH, MN, SS: integer) → clock
  - { return clock created from input }

Task/Exercise : Clock ADT

- Selector
  - Function GetHour(c : clock) → hour

    → c.HH
  - Function GetMinute(c : clock) → minute
  - Function GetSecond(c : clock) → second

- Value changer
  - Procedure SetHour(In/Out c : clock, newHH: integer)

    c.HH ← newHH
  - Procedure SetMinute(In/Out c : clock, newMM: integer)
  - Procedure SetSecond(In/Out c : clock, newSS: integer)

Task/Exercise : Clock ADT

- Relational Operation
  - Function IsEqual (c1 : clock, c2 :clock) → Boolean

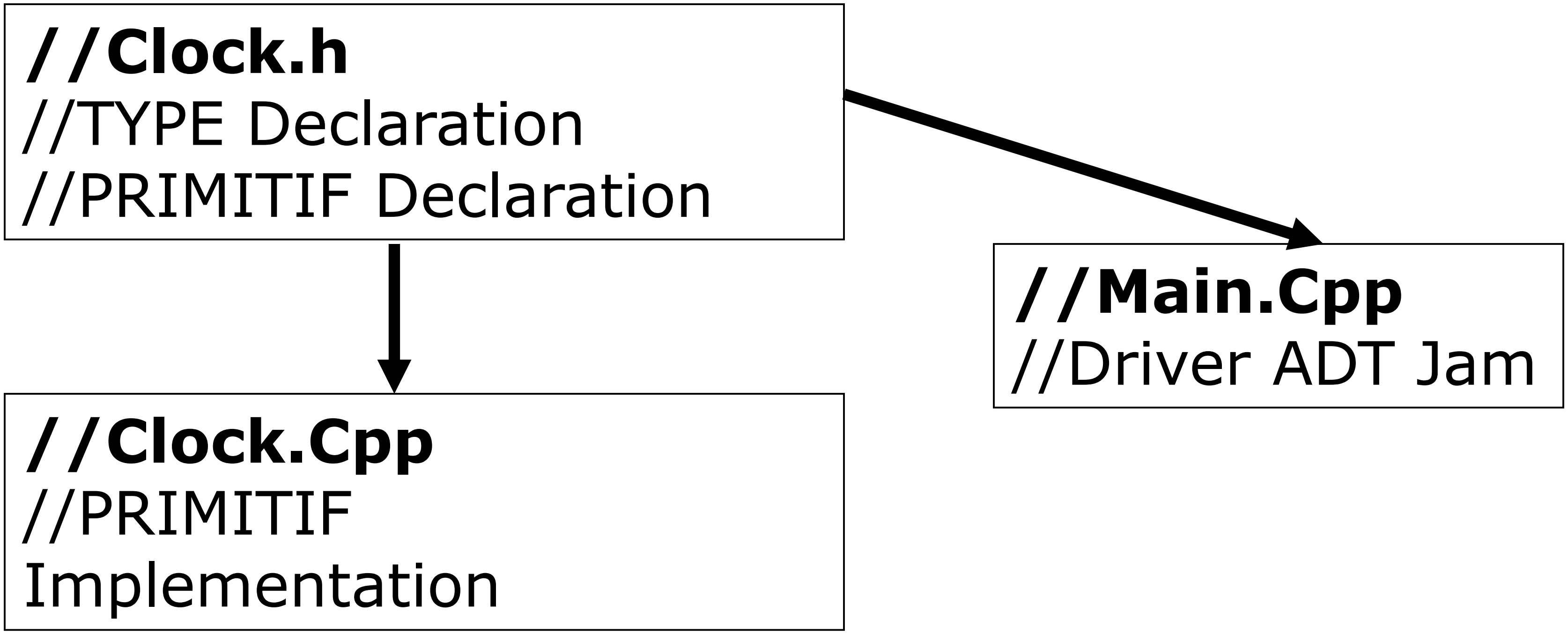
    → c1=c2
- Arithmetic Operation
  - Function AddClock (c1 : clock, c2 :clock) → clock
- Output Process
  - Procedure PrintClock (c : clock);

Task/Exercise : Clock ADT

- Create the Implementation of Clock ADT (Clock.cpp)
- Create the Driver application to try the implementation (Main.cpp)
  - Example :
  - c1 ← MakeClock(2,30,4)
  - c2 ← MakeClock(6,0,0)
  - c3 ← IsEqual (c1, c2)
  - output(GetHours(c1))

Clock ADT

Implementation Diagram of Clock ADT



//Clock.h
//TYPE Declaration
//PRIMITIF Declaration

//Clock.Cpp
//PRIMITIF Implementation

//Main.Cpp
//Driver ADT Jam

THANK YOU