Design by Contract
Modularity
Abstract Data Types (ADTs)

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

YORK UNIVERSITÉ UNIVERSITY

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. Methodology of Design by Contract (DbC)
2. Criterion of *Modularity*, Modular Design
3. *Abstract Data Types* (*ADTs*)

Terminology: Contract, Client, Supplier

- A *supplier* implements/provides a service (e.g., microwave).
- A *client* uses a service provided by some supplier.
  - The client is required to follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
  - If instructions are followed, the client would **expect** that the service does what is guaranteed (e.g., a lunch box is heated).
  - The client does not care how the supplier implements it.
- What then are the *benefits* and *obligations* os the two parties?

	benefits	*obligations*
CLIENT	obtain a service	follow instructions
SUPPLIER	assume instructions followed	provide a service

- There is a *contract* between two parties, violated if:
  - The instructions are not followed. [Client's fault]
  - Instructions followed, but service not satisfactory. [Supplier's fault]

Client, Supplier, Contract in OOP (1)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */
  } }
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();]
    m.heat(obj);
  } }
```

Method call ***m**.heat(obj)* indicates a client-supplier relation.

- **Client**: resident class of the method call [`MicrowaveUser`]
- **Supplier**: type of context object (or call target) **`m`** [`Microwave`]

Client, Supplier, Contract in OOP (2)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */ }
}
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();
    m.heat(obj);
} }
```

- The *contract* is *honoured* if:

  Right **before** the method call :

  - State of `m` is as assumed: `m.on==true` and `m.locked==ture`
  - The input argument `obj` is valid (i.e., not explosive).

  Right **after** the method call : `obj` is properly heated.

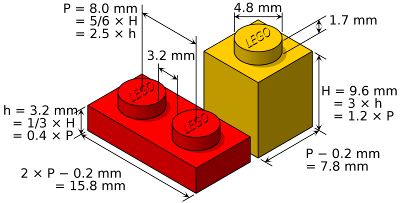
- If any of these fails, there is a *contract violation*.
  - `m.on` or `m.locked` is `false` ⇒ `MicrowaveUser`'s fault.
  - `obj` is an explosive ⇒ `MicrowaveUser`'s fault.

    A fault from the client is identified ⇒ Method call will not start.
  - Method executed but `obj` not properly heated ⇒ `Microwave`'s fault

What is a Good Design?

- A "good" design should *explicitly* and *unambiguously* describe the *contract* between **clients** (e.g., users of Java classes) and **suppliers** (e.g., developers of Java classes).
  We call such a contractual relation a *specification*.
- When you conduct *software design*, you should be guided by the "appropriate" contracts between users and developers.
  - Instructions to **clients** should *not be unreasonable*.
    e.g., asking them to assemble internal parts of a microwave
  - Working conditions for **suppliers** should *not be unconditional*.
    e.g., expecting them to produce a microwave which can safely heat an explosive with its door open!
  - You as a designer should strike proper balance between **obligations** and **benefits** of clients and suppliers.
    e.g., What is the obligation of a binary-search user (also benefit of a binary-search implementer)? [The input array is sorted.]
  - Upon contract violation, there should be the fault of **only one side**.
  - This design process is called *Design by Contract (DbC)*.

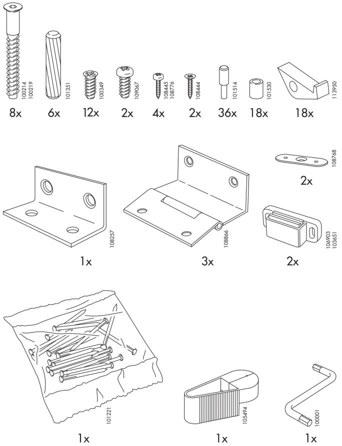
Modularity (1): Childhood Activity



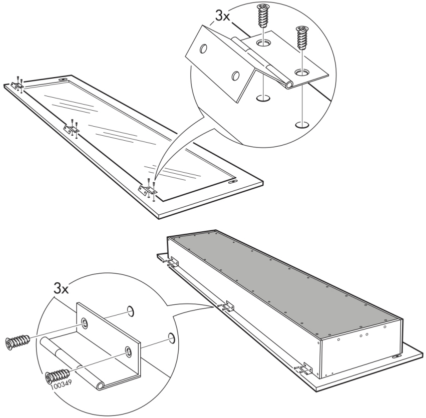
(INTERFACE) SPECIFICATION || (ASSEMBLY) ARCHITECTURE

Sources: `https://commons.wikimedia.org` and `https://www.wish.com`

Modularity (2): Daily Construction



(INTERFACE) SPECIFICATION

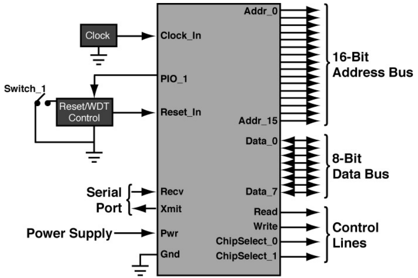


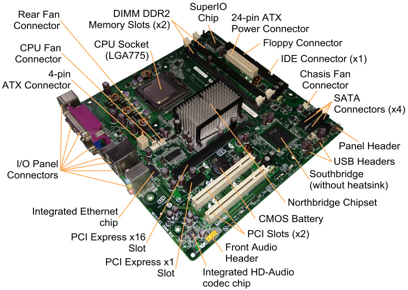
(ASSEMBLY) ARCHITECTURE

Source: `https://usermanual.wiki/`

Modularity (3): Computer Architecture

Motherboards are built from functioning units (e.g., *CPUs*).





(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

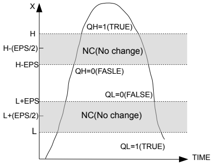
Sources: `www.embeddedlinux.org.cn` and `https://en.wikipedia.org`

Modularity (4): System Development

Safety-critical systems (e.g., *nuclear shutdown systems*) are built from *function blocks*.

```
   (* DECLARATION *)
      +---------+
      | LIMITS_ |
      | ALARM   |
REAL--|H      QH|--BOOL
REAL--|X       Q|--BOOL
REAL--|L      QL|--BOOL
REAL--|EPS      |
      +---------+
```

```
FUNCTION_BLOCK LIMITS_ALARM
  VAR_INPUT
    H   : REAL; (* High limit     *)
    X   : REAL; (* Variable value *)
    L   : REAL; (* Lower limit    *)
    EPS : REAL; (* Hysteresis     *)
  END_VAR
  VAR_OUTPUT
    QH : BOOL; (* High flag    *)
    Q  : BOOL; (* Alarm output *)
    QL : BOOL; (* Low flag     *)
  END_VAR
END_FUNCTION_BLOCK
```



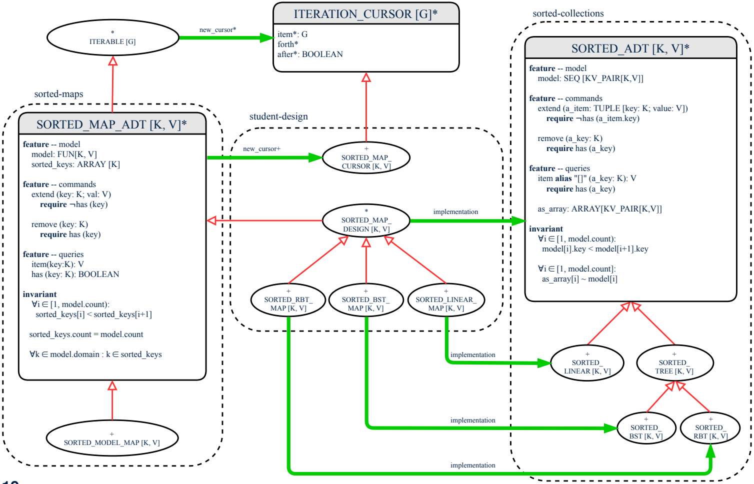

```
           (* Function block body in FBD language *)
                              HIGH_ALARM
                            +------------+
                            | HYSTERESIS |
X------------------------+--|XIN1       Q|--+----------QH
                 +---+ w2|  |            |  |
H----------------| - |------|XIN2        |  |
             +---|   |   |  |            |  |
             |   +---+   |  |            |  |
             +--------------|EPS         |  |  +-----+
      +---+w1|           |  +------------+  +--| >=1 |
EPS --| / |--|           |                     |     |--Q
2.0 --|   |  |           |    LOW_ALARM     +--|     |
      +---+  |           |  +------------+  |  +-----+
             |   +---+ w3|  | HYSTERESIS |  |
L ---------------| + |------|XIN1       Q|--+----------QL
             |   |   |   |  |            |
             +---|   |   +--|XIN2        |
             |   +---+      |            |
             +--------------|EPS         |
                            +------------+
```

(INTERFACE) SPECIFICATION || (ASSEMBLY) ARCHITECTURE

Sources: `https://plcopen.org/iec-61131-3`

Modularity (5): Software Design

Software systems are composed of well-specified *classes*.

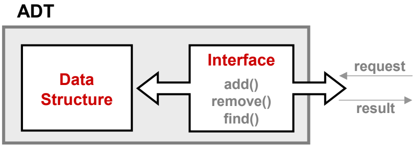


Design Principle: Modularity

- *Modularity* refers to a sound quality of your design:
  1. **Divide** a given complex ***problem*** into inter-related ***sub-problems*** via a logical/justifiable functional decomposition.
     e.g., In designing a game, solve sub-problems of: 1) rules of the game; 2) actor characterizations; and 3) presentation.
  2. **Specify** each ***sub-solution*** as a *module* with a clear **interface**: inputs, outputs, and **input-output relations**.
     - The UNIX principle: Each command does one thing and does it well.
     - In objected-oriented design (OOD), each class serves as a module.
  3. **Conquer** original ***problem*** by assembling ***sub-solutions***.
     - In OOD, classes are assembled via client-supplier relations (aggregations or compositions) or inheritance relations.
- A *modular design* satisfies the criterion of modularity and is:
  - ***Maintainable***: fix issues by changing the relevant modules only.
  - ***Extensible***: introduce new functionalities by adding new modules.
  - ***Reusable***: a module may be used in different compositions
- Opposite of modularity: A *superman module* doing everything.

Abstract Data Types (ADTs)

- Given a problem, decompose its solution into *modules*.
- Each *module* implements an *abstract data type (ADT)*:
  - filters out *irrelevant* details
  - contains a list of declared data and *well-specified* operations



- Supplier's Obligations:
  - Implement all operations
  - Choose the "right" data structure (DS)
- Client's Benefits:
  - Correct output
  - Efficient performance
- The internal details of an *implemented ADT* should be **hidden.**

Building ADTs for Reusability

- ADTs are *reusable software components*
  e.g., Stacks, Queues, Lists, Dictionaries, Trees, Graphs
- An ADT, once thoroughly tested, can be reused by:
  - Suppliers of other ADTs
  - Clients of Applications
- As a supplier, you are obliged to:
  - *Implement* given ADTs using other ADTs (e.g., arrays, linked lists, hash tables, etc.)
  - *Design* algorithms that make use of standard ADTs
- For each ADT that you build, you ought to be clear about:
  - The list of supported operations (i.e., *interface*)
    - The interface of an ADT should be *more than* method signatures and natural language descriptions:
    - How are clients supposed to use these methods? [*preconditions*]
    - What are the services provided by suppliers? [*postconditions*]
  - Time (and sometimes space) *complexity* of each operation

Why Java Interfaces ≈ ADTs (1)

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

It is useful to have:

- A *generic collection class* where the ***homogeneous type*** of elements are parameterized as `E`.
- A reasonably ***intuitive overview*** of the ADT.

Java 8 List API

Why Java Interfaces ≈ ADTs (2)

Methods described in a ***natural language*** can be ***ambiguous***:

E	`set(int index, E element)` Replaces the element at the specified position in this list with the specified element (optional operation).

set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

Parameters:

`index` - index of the element to replace

`element` - element to be stored at the specified position

Returns:

the element previously at the specified position

Throws:

`UnsupportedOperationException` - if the set operation is not supported by this list

`ClassCastException` - if the class of the specified element prevents it from being added to this list

`NullPointerException` - if the specified element is null and this list does not permit null elements

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this list

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())

Beyond this lecture...

1. **Q.** Can you think of more real-life examples of leveraging the power of ***modularity***?
2. Visit the Java API page:

   `https://docs.oracle.com/javase/8/docs/api`

   Visit collection classes which you used in EECS2030 (e.g., `ArrayList`, `HashMap`) and EECS2011.

   Q. Can you identify/justify <u>some</u> example methods which illustrate that these Java collection classes are **<u>not</u>** true ***ADTs*** (i.e., ones with well-specified interfaces)?
3. Constrast with the corresponding library classes and features in EiffelStudio (e.g., `ARRAYED_LIST`, `HASH_TABLE`).

   Q. Are these Eiffel features ***better specified*** w.r.t. obligations/benefits of clients/suppliers?

Index (1)

Index (2)