Modularity
Abstract Data Types (ADTs)

EECS3311 A & E: Software Design
Fall 2020

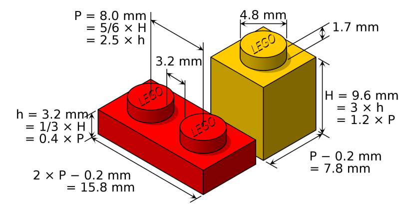
CHEN-WEI WANG

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. Criterion of *Modularity*, Modular Design
2. *Abstract Data Types* (*ADTs*)

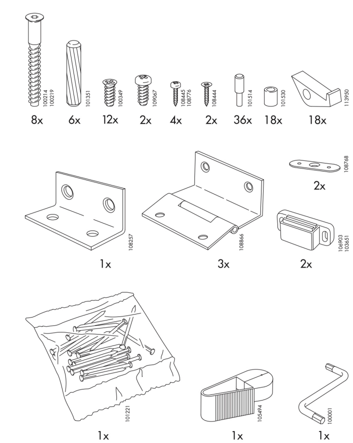
Modularity (1): Childhood Activity

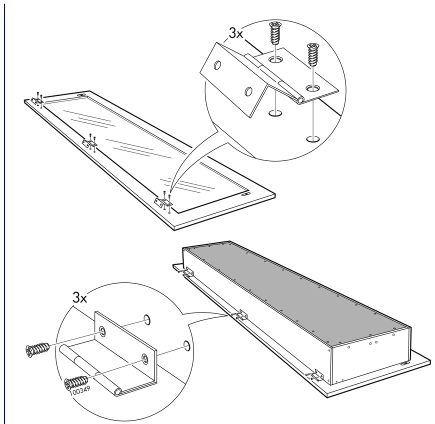


(INTERFACE) SPECIFICATION	(ASSEMBLY) ARCHITECTURE

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

Modularity (2): Daily Construction



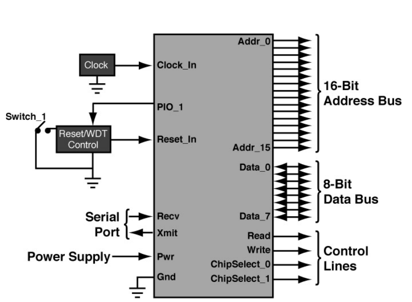


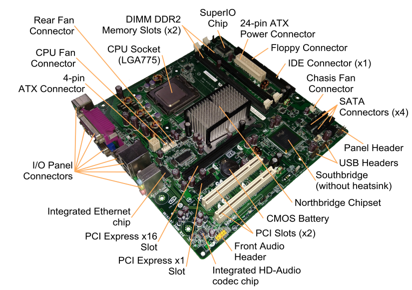
(INTERFACE) SPECIFICATION	(ASSEMBLY) ARCHITECTURE

Source: https://usermanual.wiki/

Modularity (3): Computer Architecture

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



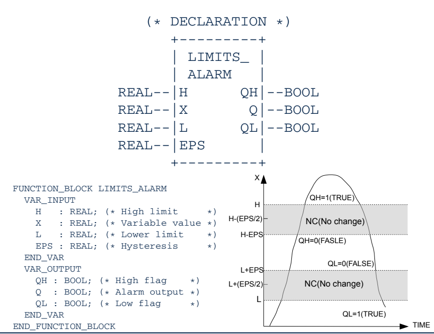


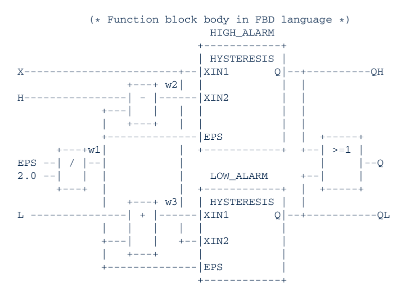
(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

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*.



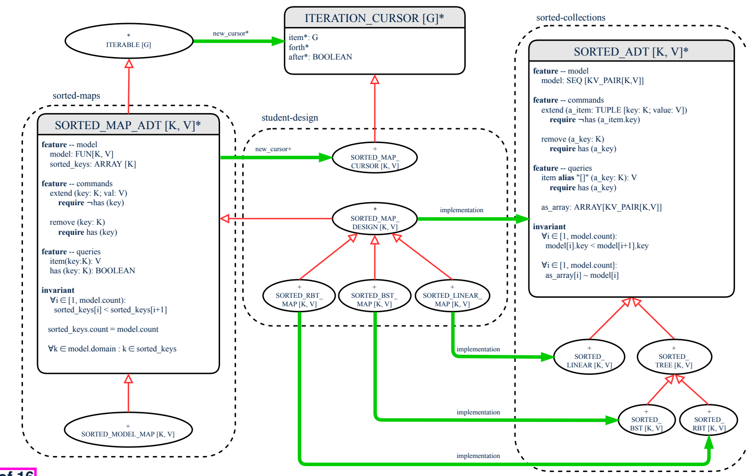


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

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

Modularity (5): Software Design

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

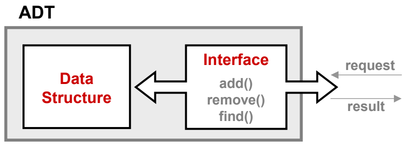


Design Principle: Modularity

- *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 Unacceptable 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 Unacceptable 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())

Why Eiffel Contract Views are ADTs (1)

```
class interface ARRAYED_CONTAINER
feature -- Commands
  assign_at (i: INTEGER; s: STRING)
      -- Change the value at position 'i' to 's'.
    require
      valid_index: 1 <= i and i <= count
    ensure
      size_unchanged:
        imp.count = (old imp.twin).count
      item_assigned:
        imp [i] ~ s
      others_unchanged:
        across
          1 |..| imp.count as j
        all
          j.item /= i implies imp [j.item] ~ (old imp.twin) [j.item]
        end
  count: INTEGER
invariant
  consistency: imp.count = count
end -- class ARRAYED_CONTAINER
```


Why Eiffel Contract Views are ADTs (2)

Even better, the direct correspondence from Eiffel operators to logic allow us to present a *precise behavioural* view.

ARRAYED_CONTAINER

feature -- Commands
assign_at (i: **INTEGER**; s: **STRING**)
-- Change the value at position 'i' to 's'.
require
valid_index: $1 \leq i \leq count$
ensure
size_unchanged: imp.count = (**old** imp.twin).count
item_assigned: imp[i] ~ s
others_unchanged: $\forall j : 1 \leq j \leq imp.count : j \neq i \Rightarrow imp[j] \sim (\textbf{old}\ imp.twin)[j]$

feature -- { NONE }
-- Implementation of an arrayed-container
imp: ARRAY[STRING]

invariant
consistency: imp.count = count

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 some example methods which illustrate that these Java collection classes are **not** 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)