Introduction to Software Engineering

ECSE-321

Unit 12 – Low Level Design

Objectives (Part I)

- To explain how visibility rules make program entities accessible through their names
- To show how references and aliases can extend access beyond visibility
- To emphasize the importance of limiting visibility and not extending access for hiding information
- To introduce cases where extending access is permissible

Entities, Names, and Visibility

A **program entity** is anything in
a program that is treated as a unit.

A **name** is an identifier
bound to a program entity.

A program entity is **visible** at a point in a program text if it can be referred to by name at that point; the portion of a text over which an entity is visible is its **visibility**.

Visibility Example

```
File: package1/PublicClass.java
package package1;
public class PublicClass{
   private String privateAttribute;
   String packageAttribute;
      public void method() {
      String localVariable;
      ...
      // point A
      ...
   }
   ...
   // point B
   ...
} // end package1.PublicClass
------------------------------------

File: package1/PackageClass.java
package package1;
class PackageClass{
   ...
   // point C
   ...
} // end package1.PackageClass
File: package2/PackageClass.java
package package2;
import package1.*;
class PackageClass{
   ...
   // point D
   ...
} // end package2.PackageClass
```

Types of Visibility

- *Local*—Visible only within the module where it is defined
- *Non-local*—Visible outside the module where it is defined, but not visible everywhere in a program
- *Global*—Visible everywhere in a program
- An entity is **exported** from the module where it is defined if it is visible outside that module.

Object-Oriented Feature Visibility

- *Private*—Visible only within the class where it is defined
  - A type of local visibility
- *Package*—Visible in the class where it is defined as well as classes in the same package or namespace
  - A form of non-local visibility
- *Protected*—Visible in the class where it is defined and all sub-classes
  - A form of non-local visibility
- *Public*—Visible anywhere the class is visible
  - A form of non-local or global visibility

Accessibility

A program entity is **accessible** at a point in a program text if it can be used at that point.

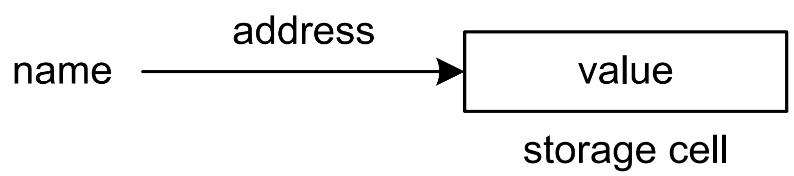
- A program entity is accessible wherever it is visible.
- A program entity may also be accessible where it is not visible.

Variables

A **variable** is a programming language device for storing values.

Variables have attributes:

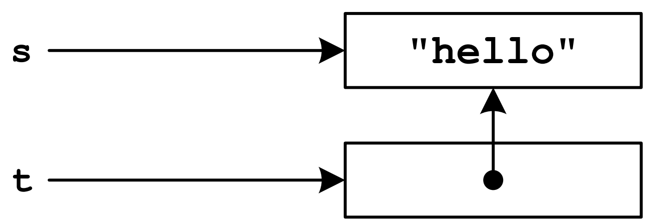
- Name
- Value
- Address



References

A **reference** is an expression that evaluates to an address where a value is stored.

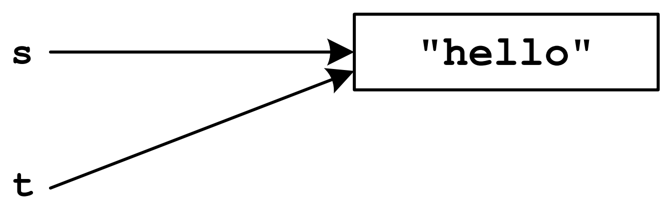
```
string s = "hello";
string *t = &s;
```



Aliases

An **alias** is a variable with the same address as another variable.

```
string s = "hello";
string &t = s;
```



Extending Access Beyond Visibility

- References and aliases can make variables accessible where they are not visible
  - Passing a reference as an argument
  - Returning a reference from a sub-program
- This practice is *extending access beyond visibility*
  - Generally it is a bad practice

Information Hiding and Access

- The key technique for hiding information is to restrict access to program entities as much as possible.
  - Limiting visibility—Use scope and visibility markers to restrict visibility
  - Not extending access—Avoid using references and aliases to extend visibility
- A **defensive copy** is a copy of an entity held by reference passed to or returned from another operation.

Information Hiding Heuristics

Limit visibility.

- Make program entities visible in the smallest possible program region.
- Restrict the scope of declarations to the smallest possible program region.
- Make attributes at least protected and preferably private.
- Make helper operations at least protected and preferably private.
- Avoid global visibility.
- Avoid package visibility.

Information Hiding Heuristics…

Don’t extend access.

- Don’t initialize attributes with references passed to the class—make defensive copies instead.
- Don’t pass or return references to attributes—pass or return defensive copies instead.
- Don’t pass parameters by reference.
- Don’t make aliases

Exceptions

Two cases when access may be extended beyond visibility:

- Modules must share an entity to collaborate
  - Example: a shared queue
- Some other design goal is of greater importance than information hiding
  - Example: performance constraints

Summary

- Program entities are usually accessible through their names by being visible in various parts of a program text.
- Entities may also be accessed through references or aliases.
- Information hiding dictates that visibility be limited and that access not be extended beyond visibility.
- Occasionally this rule can be violated to achieve other goals.

Objectives (Part 2)

- To present operation specifications and their contents
- To present design by contract for declarative specification of operation behavior
- To introduce minispecs and pseudocode for algorithm specification
- To introduce data structure diagrams for data structure specification
- To survey design finalization

Operation Specification (Op-Spec)

Structured text stating an operation's interface and responsibilities

- *Class or module*—Identifies the operation
- *Signature*—Operation name, names and types of parameters, return type, and perhaps more (syntax)
- *Description*—Sentence or two
- *Behaviour*—Semantics and pragmatics
- *Implementation*—Optional

Behavior Specification

- *Procedural*—Describes an algorithm for transforming inputs to outputs
  - An **algorithm** is a sequence of steps that can be performed by a computer.
- *Declarative*—Describes inputs, outputs, calling constraints, and results without specifying an algorithm

Declarative Specification Advantages

- More abstract because they ignore implementation details—more concise
- Focus on the interface, not the internals
- Do not bias programmers towards a particular implementation as procedural specifications might (what vs how)

Design by Contract

A **contract** is a binding agreement between two or more parties.

An **operation contract** is a contract between an operation and its callers.

Contract Rights and Obligations

- The caller
  - Is obliged to pass valid parameters under valid conditions, and
  - Has the right to delivery of advertised computational services.
- The called operation
  - Is obliged to provide advertised services, and
  - Has the right to be called under valid conditions with valid parameters.

Assertions

An **assertion** is a statement that must be true at a designated point in a program.

Assertions state caller and called operation right and obligations.

Preconditions and Postconditions

A **precondition** is an assertion that must be true at the initiation of an operation.

A **postcondition** is an assertion that must be true upon completion of an operation.

- Preconditions state caller obligations and called operation rights.
- Postconditions state caller rights and called operation obligations.

Operation Specification Example

Signature	public static int findMax(int[] a) throws IllegalArgumentException
Class	Utility
Description	Return one of the largest elements in an int array.
Behavior	pre: (a != null) && (0 < a.length) post: for every element x of a, x <= result post: throws IllegalArgumentException if preconditions are violated

Class Invariants

A **class invariant** is an assertion that must be true of any class instance between calls of its exported operations.

Class invariants augment every exported operation's contract.

What to put in Assertions

- Preconditions:
  - Restrictions on parameters
  - Conditions that must have been established before the call
- Postconditions
  - Relationships between parameters and results
  - Restrictions on results
  - Changes to parameters
  - Responses to violated preconditions
- Class invariants
  - Restrictions on attributes
  - Relationships among attributes
- State empty assertions as “true” or “none.”

Developing Op-Specs

- Don’t make detailed op-specs early in mid-level design
  - The design is still fluid and many details will change
- Don’t wait until the end of design
  - Details will have been forgotten
  - Probably will be done poorly
- Develop op-specs gradually during design, adding details as they become firm

Algorithm Specification

- Specify well-known algorithms by name.
- Use a **minispec**, a step-by-step description of how an algorithm transforms its inputs to output.
- Write minispecs in **pseudocode**, English augmented with programming language constructs.

Pseudocode Example

```
Inputs:  array a, lower bound lb, upper bound ub,
    search key
Outputs: location of key, or -1 if key is not
    found
lo = lb
hi = ub
while lo <= hi and key not found
   mid = (lo + hi) / 2
   if        ( key = a[mid] ) then key is found
   else if ( key < a[mid] ) then hi = mid-1
   else                            lo = mid+1
if key is found then return mid
else                 return -1
```

Data Structures

A **data structure** is scheme for storing data in computer memory.

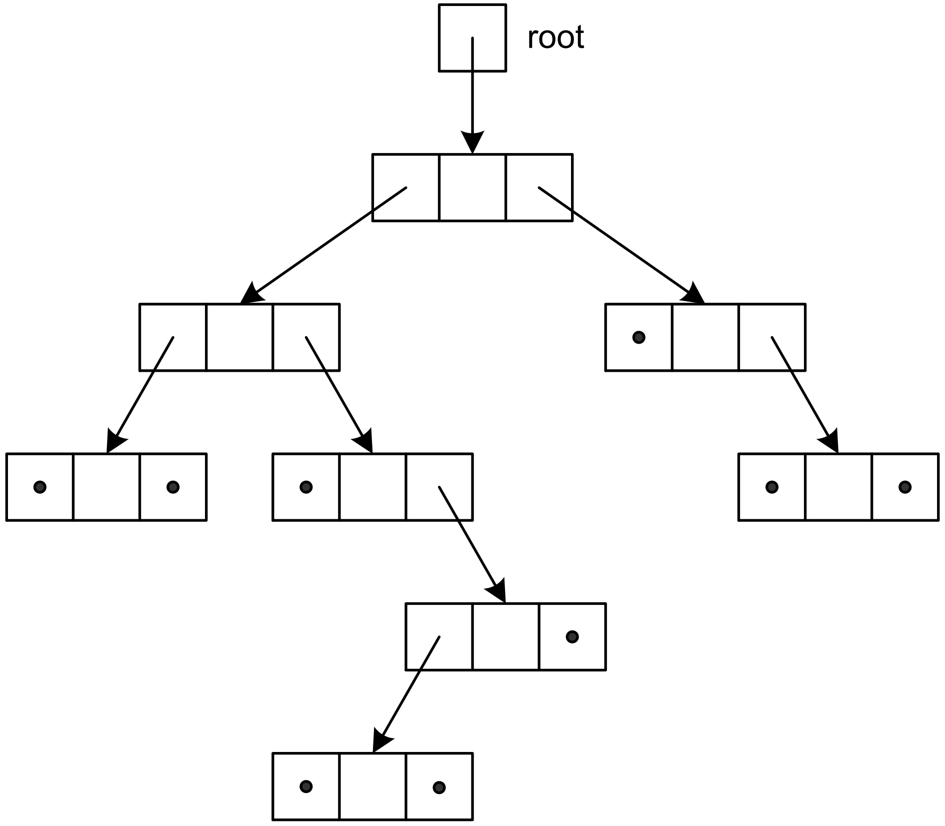
- *Contiguous implementation*—Values are stored in adjacent memory cells
- *Linked implementation*—Values are stored in arbitrary cells accessed using references or pointers

Data Structure Diagrams

- Rectangles represent memory cells, possibly with names
- Contiguous cells are represented by adjacent rectangles; cells may have indices
- Repeated elements are indicated by ellipses
- Linked cells are shown using arrows to represent pointers or references from one cell to another

Data Structure Diagram Example

Data Structure Diagram Example…



Data Structure Diagram Heuristics

- Label record fields only once.
- Use ellipses to simplify large, repetitive structures.
- Draw linked structures so that the pointers point down the page or from left to right.
- Identify unusual or additional symbols with a legend.

Design Finalization

- Low-level design specifications complete a design document.
- **Design finalization** is checking the design to make sure it is of sufficient quality and is well documented.
- This is the last step in the engineering design process.

Design Document Quality Characteristics

- *Feasibility*—Must be possible to realize the design
- *Adequacy*—Must specify a program that will meet its requirements
- *Economy*—Must specify a program that can be built on time and within budget
- *Changeability*—Must specify a program that can be changed easily

Design Document Quality Characteristics…

- *Well-Formedness*—Design must use notations correctly
- *Completeness*—Must specify everything that programmers need to implement the program
- *Clarity*—Must be as easy to understand as possible
- *Consistency*—Must contain specifications that can be met by a single product

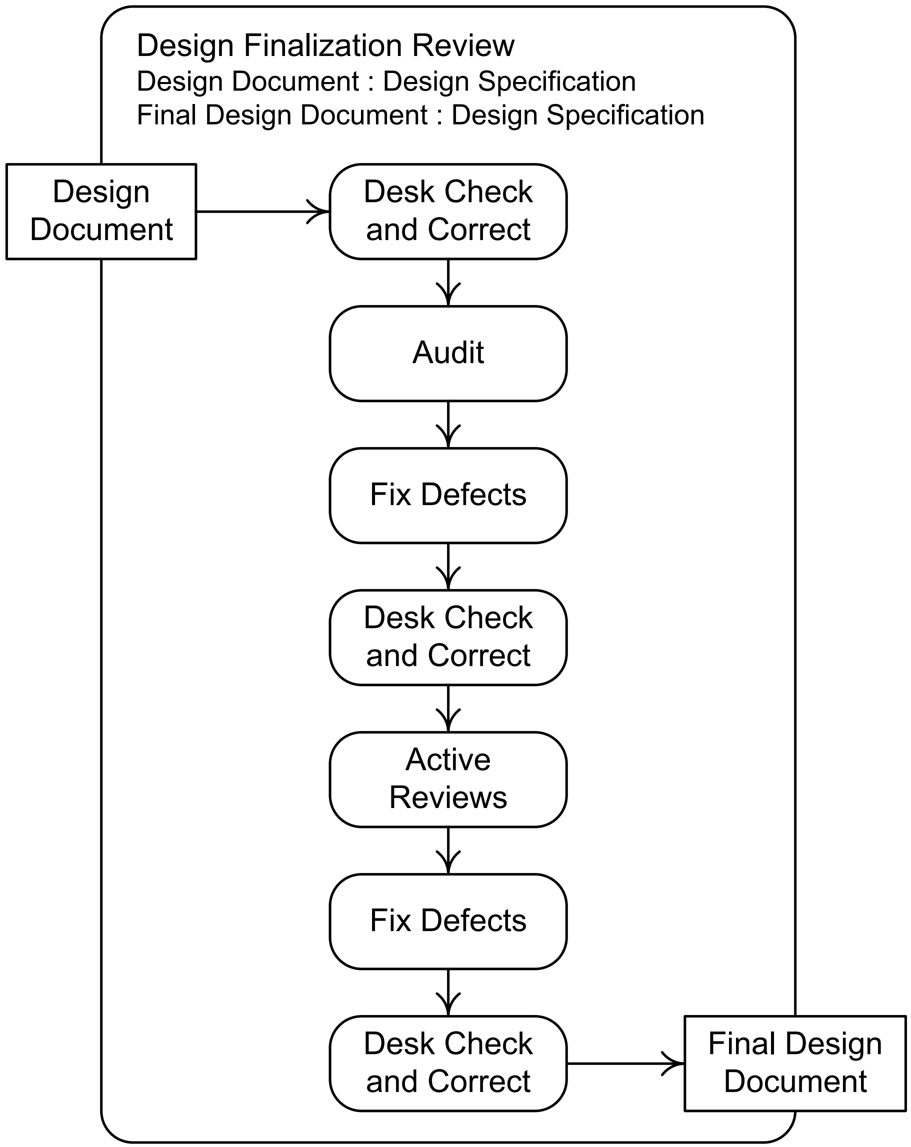
Critical Reviews

A **critical review** is an evaluation of a finished product to determine whether it is of acceptable quality.

Critical reviews can utilize

- Desk checks,
- Walkthroughs,
- Inspections,
- Audits, and
- Active reviews.

A Critical Review Process



Continuous Review

- A critical review that finds serious design defects may result in a return to a much earlier stage of design.
  - Expensive
  - Time consuming
  - Frustrating
- A policy of **continuous review** during the design process helps find faults early, avoiding the pain of finding them later.

Summary

- Operation specifications state design details about operations, including their
  - Class or module
  - Signature
  - Description
  - Behavior
  - Implementation
- Behavior can be specified declaratively or procedurally.

Summary…

- Declarative specification is done using operation contracts stated in assertions.
  - Preconditions state caller obligations and called operation rights.
  - Postconditions state caller rights and called operation obligations.
- Algorithms are specified in minispecs, often in pseudocode.
- Data structures are specified using data structure diagrams.

Summary…

- Design finalization is the last step of engineering design.
- The design document is checked in a critical review to ensure that it has all the requisite quality characteristics.
- A critical review that finds many defects can be a disaster that can be mitigated by conducting continuous reviews throughout the engineering design process.