

Abstraction and Interface Specification

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Feb. 6, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 What is an interface?
- 2 Why interfaces?
- 3 How to develop and use interfaces in Java?

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Programming assignment 1 is posted on your lab Canvas page.

Two Questions to Answer Today

- 1 What is an interface specification?
- 2 Why does it need abstraction?

Recall: Information Hiding

- ▶ Idea of hiding unnecessary details from users
- ▶ An interface facilitates information hiding and provides a separation of concerns between class users and implementers
- ▶ Users can ignore the classes and their implementation details and depend on just interfaces

Interface Specification

- ▶ For users to depend on just interfaces, interfaces must contain what users need to know and that goes beyond operation names (methods) and parameters
 - ▶ We use contracts to specify these

Interface Specification

- ▶ It is a contract expressed in an `interface`
- ▶ Why does it need abstraction?

Interface Specification

- ▶ It is a contract expressed in an `interface`
- ▶ Why does it need abstraction?
 - ▶ We need to hide the details of the implementation
 - ▶ We don't know how it could be implemented

Recall: Grid Example

- ▶ We have an interface: `Grid`
 - ▶ Stores the characters in a grid
 - ▶ Provides methods such as `addMarker`, `whatsAtPos`, and other useful methods
- ▶ We have 2 implementations of the interface:
 - 1 `GridFast` uses a 2D array of characters
 - ▶ Very fast to check a particular position
 - ▶ Very memory intensive
 - 2 `GridMem` uses a `List` of `<character, row, column>`
 - ▶ Takes more time to check a particular position
 - ▶ Much more memory efficient (until grid fills up)
- ▶ How the grid is implemented is different, but since they implement the same interface, they have the same `addMarker` and `whatsAtPos` methods!

Interface Specification

- ▶ It is a contract expressed in an `interface`
- ▶ Why does it need abstraction?
 - ▶ We need to hide the details of the implementation
 - ▶ We don't know how it could be implemented
 - ▶ We can't refer to the underlying private data structure.
 - ▶ That is different for each implementation
 - ▶ We keep our specification general and abstract so it will fit any implementation
 - ▶ The only variables we know exist are ones declared as `public static final` (constants).

Interface Specification

- ▶ Our interface specification can refer to general abstract values as long as they don't enforce a specific implementation
- ▶ We can refer to a number of rows, and a number of columns in our grid as long as we don't enforce a way that data is stored
 - ▶ Maybe they are stored as `private` variables
 - ▶ Maybe they are stored as the length of the array
- ▶ Our specification should give enough detail so it is clear what is expected of the implementation, but not enforce a specific implementation
 - ▶ Hard balance to find

Interface Specification

- ▶ Should have a description of what the `interface` represents
 - ▶ Will get more formal eventually
- ▶ **Initialization ensures:** what will be *true* after initialization (after calling the constructor)
 - ▶ The constructor is not in the `interface`
 - ▶ This is how we can give information about what we expect the constructor to do
 - ▶ Ensures consistency in our implementations

Interface Specification

- ▶ **Defines:** Allows us to define some concepts for our specification¹
 - ▶ What information do we expect our implementations to have?
 - ▶ Allows us to hint at `private` variables that may be available
 - ▶ Again, must be generic and abstract, not tied to any implementation
- ▶ **Constraints:** Allows us to add constraints in some of the specifications.
 - ▶ Usually tells us how the defined concepts relate to our `interface`
 - ▶ Similar to **invariants**

¹However, it does not force a specific implementation.

Grid Interface

```
/**  
Grid is abstractly a 2 dimensional grid of characters.  
Indexing starts at 0  
  
Initialization ensures:  
    Grid contains only blank characters and  
    is MAX_SIZE x MAX_SIZE or smaller  
  
Defines:        number_of_rows:  Z  
                   number_of_columns:  Z  
  
Constraints:    0 < number_of_rows <= MAX_SIZE  
                   0 < number_of_cols <= MAX_SIZE  
  
*/  
public interface Grid {  
    public static final int MAX_SIZE = ...;  
    ...  
}
```

Grid Interface

```
public interface Grid {  
    ...  
  
    /**  
        @pre    0 <= r < number_of_rows and  
                0 <= c < number_of_columns  
        @post   self = #self, except a will appear in  
                row r, column c  
    */  
    void placeChar(int r, int c, char a);  
}
```

Grid Interface

- ▶ This interface specification does not know whether the implementation will use a 2D array, or a List, or something else.
- ▶ It does not need to use the implementation's private variables to write its contract.
- ▶ The contracts are still general and abstract

Note: In your specifications, you might find the need to distinguish between the parameter's incoming and outgoing values. In this case, use: `#<variable_name>` to indicate that it is the incoming value and `<variable_name>` to refer to the outgoing value.

Example: $x = \#x + 1$

Primitive Types and Abstraction

- ▶ If our parameters are primitive types like `int` and `double` then it is obvious how to express in mathematical notation.
- ▶ Though these types are represented internally with bits and bytes, users can view them *abstractly* like mathematical variables.
 - ▶ **Important Caveat:** Values are bounded in computing!

Example: Stack

- ▶ A **Stack** allows entries to be inserted and removed in **LIFO** (last-in-first-out) order

How to Specify this Interface?

```
public interface IntegerStack {  
  
    void push(Integer x);  
    Integer pop();  
    int depth();  
    ...  
  
}
```

How to Specify this Interface?

```
/**  
  Stack is abstractly a string of integers.  
  
  Initialization ensures:  
    Stack contains nothing, i.e., an empty string <>.  
  
  Constraints: length of a self <= MAX_DEPTH  
*/  
public interface IntegerStack {  
    public static final int MAX_DEPTH = ...;  
    ...  
}
```

How to Specify this Interface?

```
public interface IntegerStack {  
    ...  
  
    /**  
        @pre    length of self, |self| < MAX_DEPTH  
        @post    self = x added to the front of #self  
    */  
    void push(Integer x);  
  
    /**  
        @pre    length of self, |self| > 0  
        @post    self = pop() removed from the front of #self  
    */  
    Integer pop();  
  
    ...  
}
```

How to Specify this Interface Formally?

Future Discussion

```
public interface IntegerStack {  
    ...  
  
    /**  
        @pre    |self| < MAX_DEPTH  
        @post   self = <x> o #self  
    */  
    void push(Integer x);  
  
    /**  
        @pre    |self| > 0  
        @post   #self = pop() o self  
    */  
    Integer pop();  
  
    ...  
}
```

Dual Ideas of Specification

- ▶ Information hiding
 - ▶ Hide details unnecessary to use the `interface`
- ▶ Abstraction
 - ▶ Provide a “cover story” or explanation in user-oriented terms so they can understand the `interface`
- ▶ If there's not a formal mathematical notation to use, then we are stuck with an informal contract
- ▶ Informal $\neq$ Easy to Write

Specifications

- ▶ Straightforward descriptions
 - ▶ `Push` pushes an object on a stack
 - ▶ How much do they help?
 - ▶ Carefully write the contracts. Rely on the interface description and invariants
- ▶ Use of metaphors
 - ▶ A `Queue` is like a line at a fastfood restaurant
 - ▶ Do they generalize?
- ▶ Use of private implementation details
 - ▶ Remember information hiding!
 - ▶ Users shouldn't have to depend on implementation details

Implementing our Specification

- ▶ Our interface has our specification and contracts
- ▶ Our implementation has our code and our `private` variables
- ▶ We need to tie the two together
 - ▶ Add **invariants** that use the implementation details
 - ▶ We are exposing more information, but the client will not look at these unless they need to know implementation details
 - ▶ The client is not the intended audience
- ▶ Invariants may not be enough

Correspondences

- ▶ **Correspondences** allow us to tie the concepts specified in the interface to the `private` data in the implementation
 - ▶ Ties the abstraction to the implementation
- ▶ Correspondences do reveal information
- ▶ Correspondences and the contracts in the implementation are helpful for:
 - ▶ The implementers of the implementation
 - ▶ Multiple programmers could work on the implementation
 - ▶ Their understanding of the implementation affects how they write the code
 - ▶ Clients with specific restrictions that require them to know implementation details
 - ▶ Performance tradeoffs

Correspondences

- ▶ Correspondences tell us how the private data implements the interface specification.
- ▶ Consider our `Grid` interface and `GridFast` implementation...

Grid Interface

```
/**
Grid is abstractly a 2 dimensional grid of characters.
Indexing starts at 0

Initialization ensures:
    Grid contains only blank characters and
    is MAX_SIZE x MAX_SIZE or smaller

Defines:      number_of_rows:  Z
               number_of_columns:  Z
Constraints:  0 < number_of_rows <= MAX_SIZE
               0 < number_of_cols <= MAX_SIZE
*/
public interface Grid {
    public static final int MAX_SIZE = ...;
    ...
}
```

GridFast Implementation

```
/**
 * @invariant 0 < numRows <= MAX_SIZE AND
 *             0 < numCol <= MAX_SIZE
 *
 * @correspondence number_of_rows = numRows AND
 *                 number_of_columns = numCol AND
 *                 self = myGrid[0...numRow-1][0...numCol-1]
 */
public class GridFast implements Grid {

    private char[][] myGrid;
    private int numRows;
    private int numCol;

    // Need to add contracts for the constructor!
    public GridFast(int r, int c) {
        myGrid = new char[r][c];
        numRows = r;
        numCol = c;
        ...
    }
    ...
}
```

Correspondence

- ▶ We now know how the implementation is connected to the interface specification
- ▶ We do not need to include preconditions and postconditions for any methods specified in the interface
 - ▶ We can use the correspondence to map them to the preconditions and postconditions in the **interface**
- ▶ Do need contracts for any additional methods
 - ▶ Including the constructor
 - ▶ Should be in terms of the interface specification
 - ▶ Client may see these
- ▶ Add invariants for our **private** data
 - ▶ Helps tie back to the interface specification
 - ▶ Similar to **Constraints**

Design Process

- ▶ First fully design the `interface`
 - ▶ What methods need to be available?
 - ▶ What `public static final` variables exist?
 - ▶ Don't worry about how they work yet, it's not the time for that
- ▶ Write the interface specification
 - ▶ What concepts exist for this class
 - ▶ Write the contracts for each method
 - ▶ Contracts and method names should refer to the concepts, not any `private` data
- ▶ Now design the class that will implement the `interface`
 - ▶ How will we represent the concepts in our `private` data
 - ▶ Write the correspondences to connect our `private` data to the concepts in the `interface`
 - ▶ Write your invariants
- ▶ Now write the code for each method based on
 - ▶ The contracts in the `interface`
 - ▶ The correspondences

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Programming assignment 1 is posted on your lab Canvas page.

Summary

- 1 What is an interface specification?
- 2 Why does it need abstraction?