

17-423/723: Designing Large-scale Software Systems

Designing Interface Specifications

Feb 5, 2025

Learning Goals

- Describe the role and importance of an interface specification
- Describe the structure and meaning of a specification
- Describe four different dimensions that must be considered while designing a specification

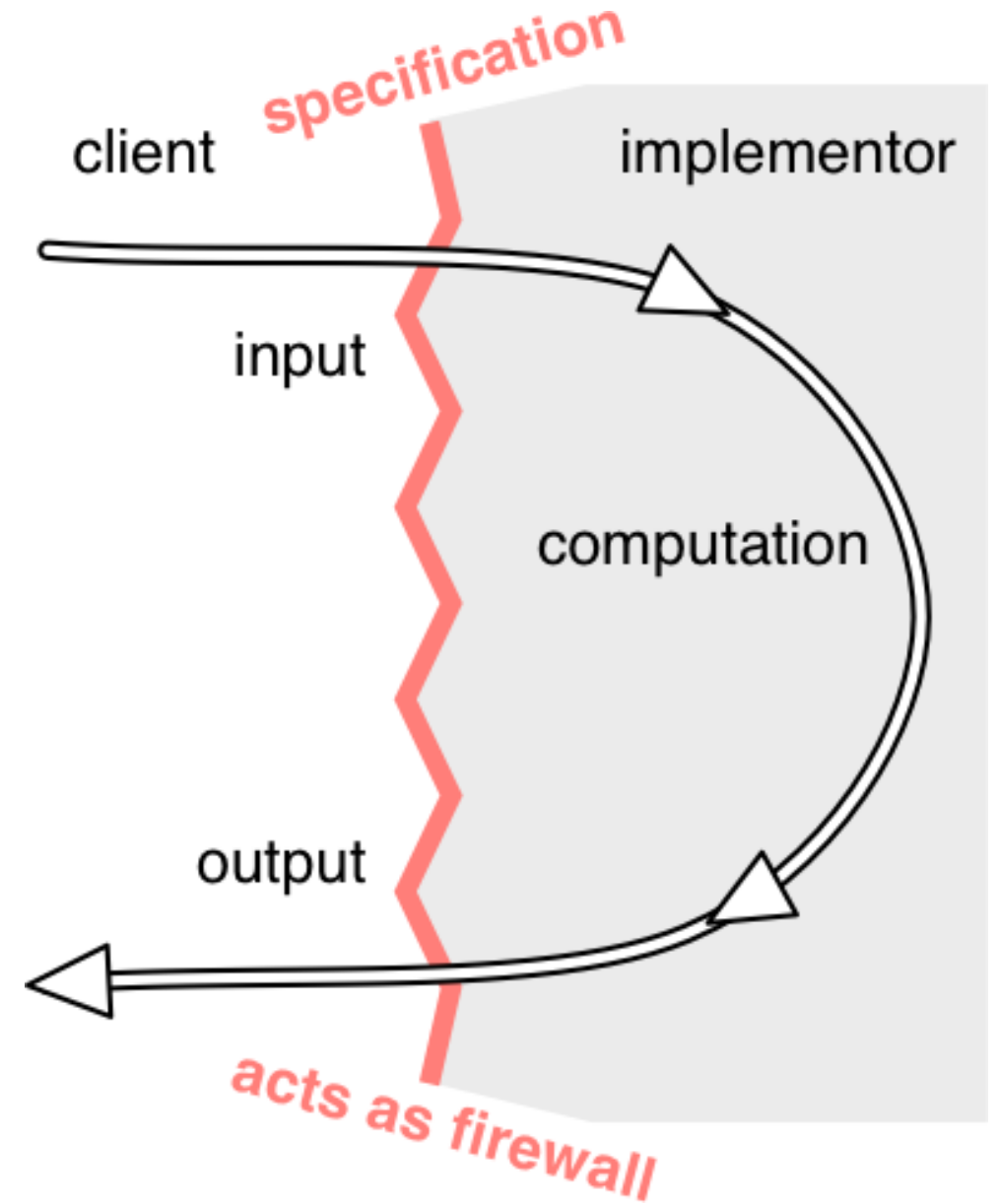
Interface Specifications

Specification

- A statement of a desired behavior or quality attribute of a software system
- Functional specification
 - “The scheduling system must provide a way for the patient to modify an existing appointment”
- Quality attribute specification
 - “The system must be able to handle additional 5000 users without a loss of latency” (scalability)
- **Interface specification**
 - Describes a piece of functionality or a service that a component is expected to deliver to its clients
 - **Today’s focus!**

Interface Specification

- **Contract** between a client and a component
- **For clients:**
 - Describes what a client needs to know to use the component
 - Describes what is expected as the output, given an input
 - Hides implementation details (secrets!)
- **For implementors:**
 - Describes implementation tasks to be fulfilled by developers (or LLMs)
 - Hides possible uses of the component by clients (**Q. Why is this good?**)



Interface Specifications in Practice

Java Collections API

OVERVIEW	PACKAGE	CLASS	USE	TREE	DEPRECATED	INDEX	HELP
PREV CLASS	NEXT CLASS	FRAMES	NO FRAMES	ALL CLASSES			
SUMMARY: NESTED	FIELD	CONSTR	METHOD	DETAIL: FIELD	CONSTR	METHOD	

compact1, compact2, compact3
java.util

Class HashSet<E>

java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractSet<E>
 java.util.HashSet<E>

Method Summary

All Methods

Instance Methods

Concrete Methods

Modifier and Type

Method and Description

boolean

add(E e)

Adds the specified element to this set if it is not already present.

void

clear()

Removes all of the elements from this set.

Object

clone()

Returns a shallow copy of this HashSet instance: the elements then

boolean

contains(Object o)

Returns true if this set contains the specified element.

boolean

isEmpty()

Returns true if this set contains no elements.

Iterator<E>

iterator()

Returns an iterator over the elements in this set.

Interface Specifications in Practice

Python Docstrings

```
def add(num1, num2):  
    """  
    Add up two integer numbers.  
  
    This function simply wraps the ``+`` operator, and does not  
    do anything interesting, except for illustrating what  
    the docstring of a very simple function looks like.  
  
    Parameters  
    -----  
    num1 : int  
        First number to add.  
    num2 : int  
        Second number to add.  
  
    Returns  
    -----  
    int  
        The sum of ``num1`` and ``num2``.  
  
    See Also  
    -----  
    subtract : Subtract one integer from another.  
  
    Examples  
    -----  
    >>> add(2, 2)  
    4  
    >>> add(25, 0)  
    25  
    >>> add(10, -10)  
    0  
    """
```

Interface Specifications in Practice

Swagger Petstore

1.0.7

OAS 2.0

[Base URL: petstore.swagger.io/v2]

<https://petstore.swagger.io/v2/swagger.json>

This is a sample server Petstore server. You can find out more about Swagger

REST API Doc

store Access to Petstore orders

GET

/store/inventory Returns pet inventories by status

POST

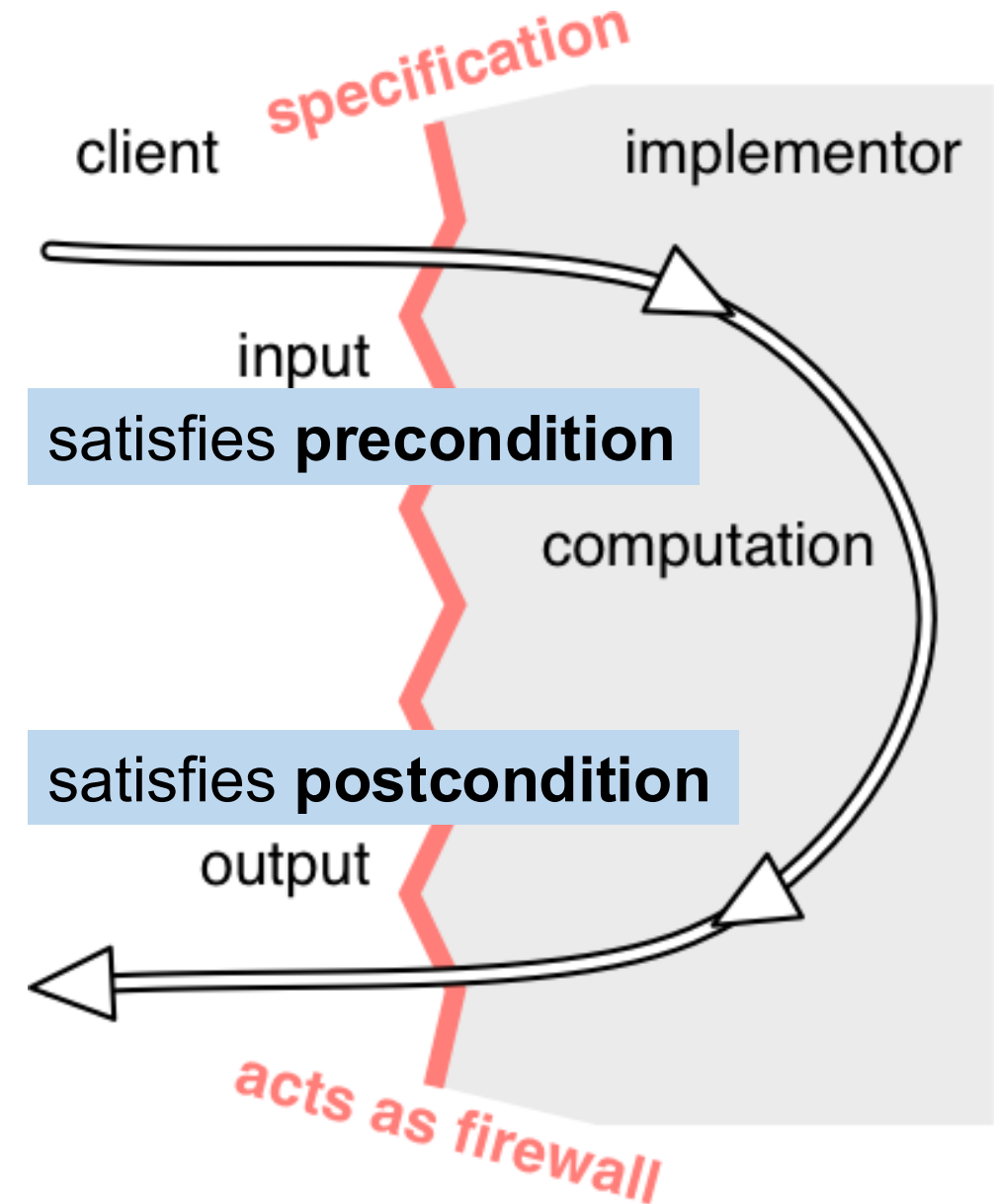
/store/order Place an order for a pet

Parameters

Name	Description
body * required	order placed for purchasing the pet
object (body)	<div>Example Value Model</div> <pre>{ "id": 0, "petId": 0, "quantity": 0, "shipDate": "2025-02-03T21:44:39.405Z", "status": "placed", "complete": true }</pre> <div>Parameter content type</div> <div>application/json</div>

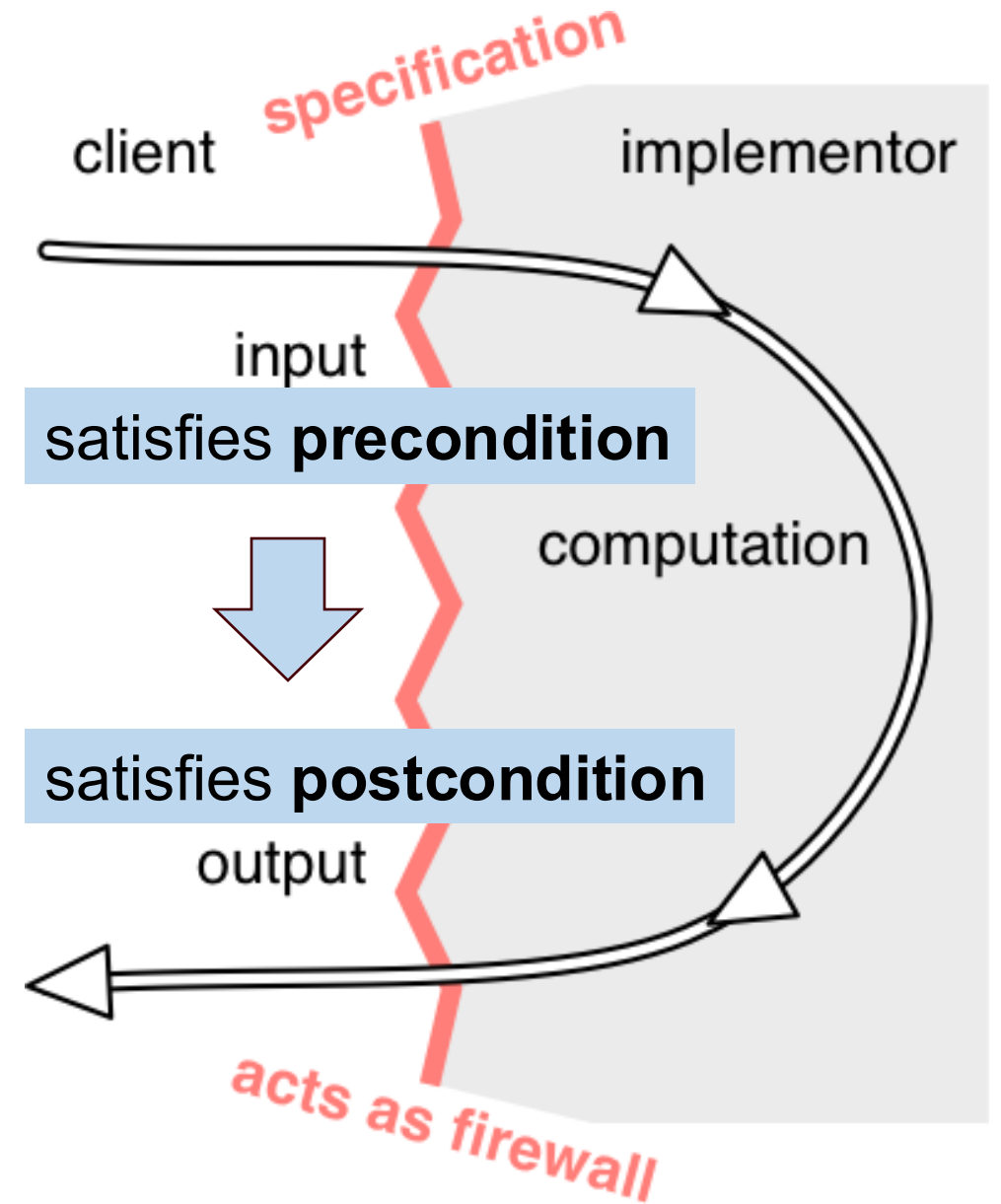
Specification: Elements

- Each specification of a function is associated with **pre- & post-conditions**
- **Pre-condition**
 - What the component **expects from the client**, expressed as a condition over the function input and/or component state
- **Post-condition**
 - What the component **promises to deliver**, as a condition over the function output and/or component state



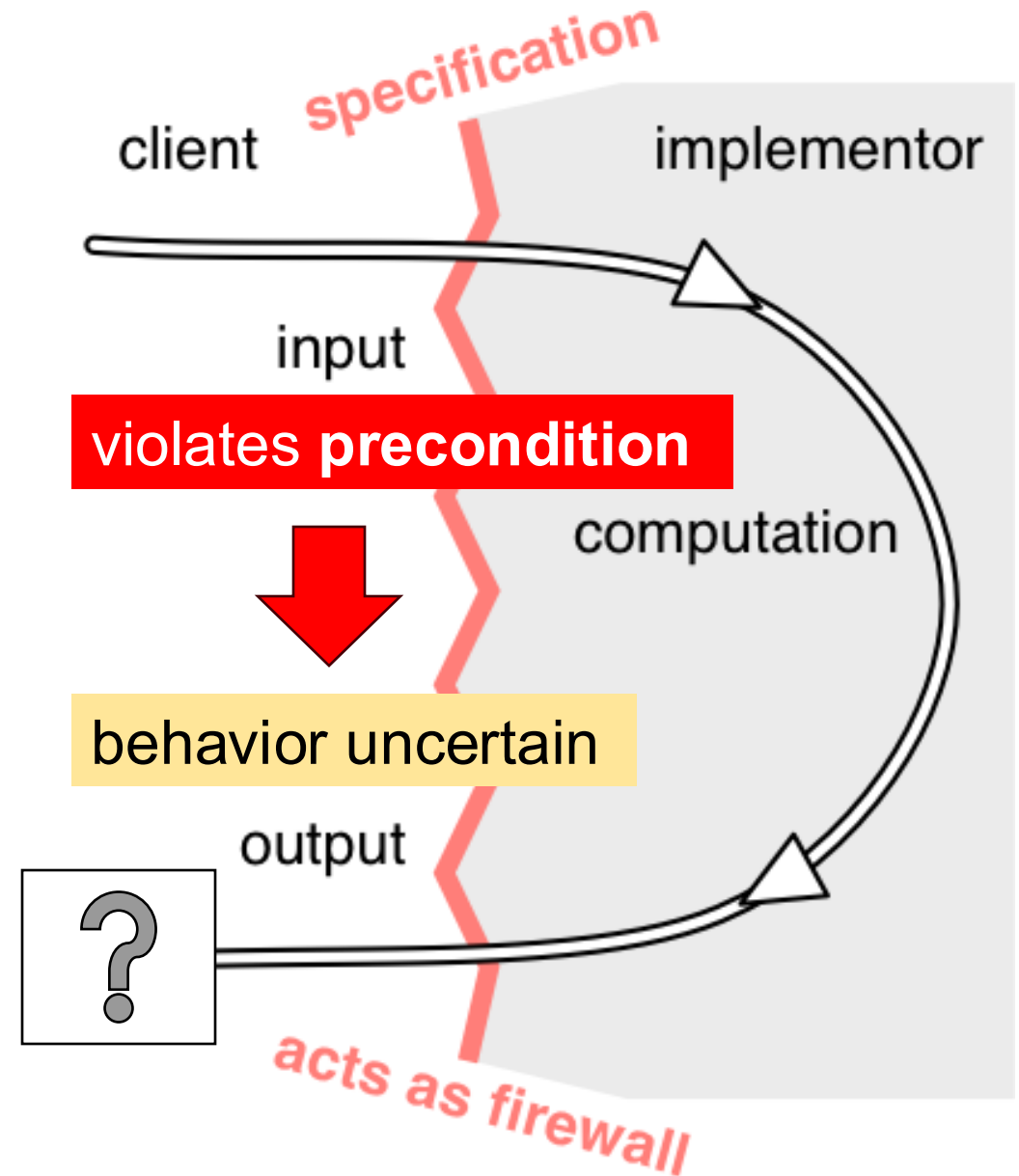
Specification: Meaning

- **Pre-condition** \Rightarrow **Post-condition**
(i.e., logical implication)
- If the client satisfies the pre-condition, the component promises to satisfy post-condition



Specification: Meaning

- **Pre-condition** \Rightarrow **Post-condition**
(i.e., logical implication)
- If the client satisfies the pre-condition, the component promises to satisfy post-condition
- But if the client violates the pre-condition, the component can behave in an arbitrary way!
 - Logically, “false implies anything”
 - **Q. Why is this reasonable?**



Example: Specifying Array Find

```
static int find(int[] arr, int val)
```

requires: val occurs exactly once in arr

effects: returns index i such that arr[i] = val

- A specification of a “find” function
- By convention, we will label pre- & post-conditions as **requires** and **effects**
- **Meaning**: If “val” occurs exactly once in “arr”, then it returns index “i” such that arr[i] = val
 - If “val” occurs zero times or more than once, then “find” may return anything

Specification as an Implementation Set

- Specification defines a **set of possible implementations**
- Given a pre- & post-condition, any implementation that fulfills the requirement “**pre-condition** \Rightarrow **post-condition**” is a valid implementation of the specification

Example: Implementing Array Find

```
static int find(int[] arr, int val) {  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] == val) return i;  
    }  
    return arr.length;  
}
```

```
static int find(int[] arr, int val) {  
    for (int i = arr.length - 1; i >= 0; i--) {  
        if (arr[i] == val) return i;  
    }  
    return -1;  
}
```

Q. Do these functions behave the same or differently?

Example: Specifying Array Find

```
static int find(int[] arr, int val)
```

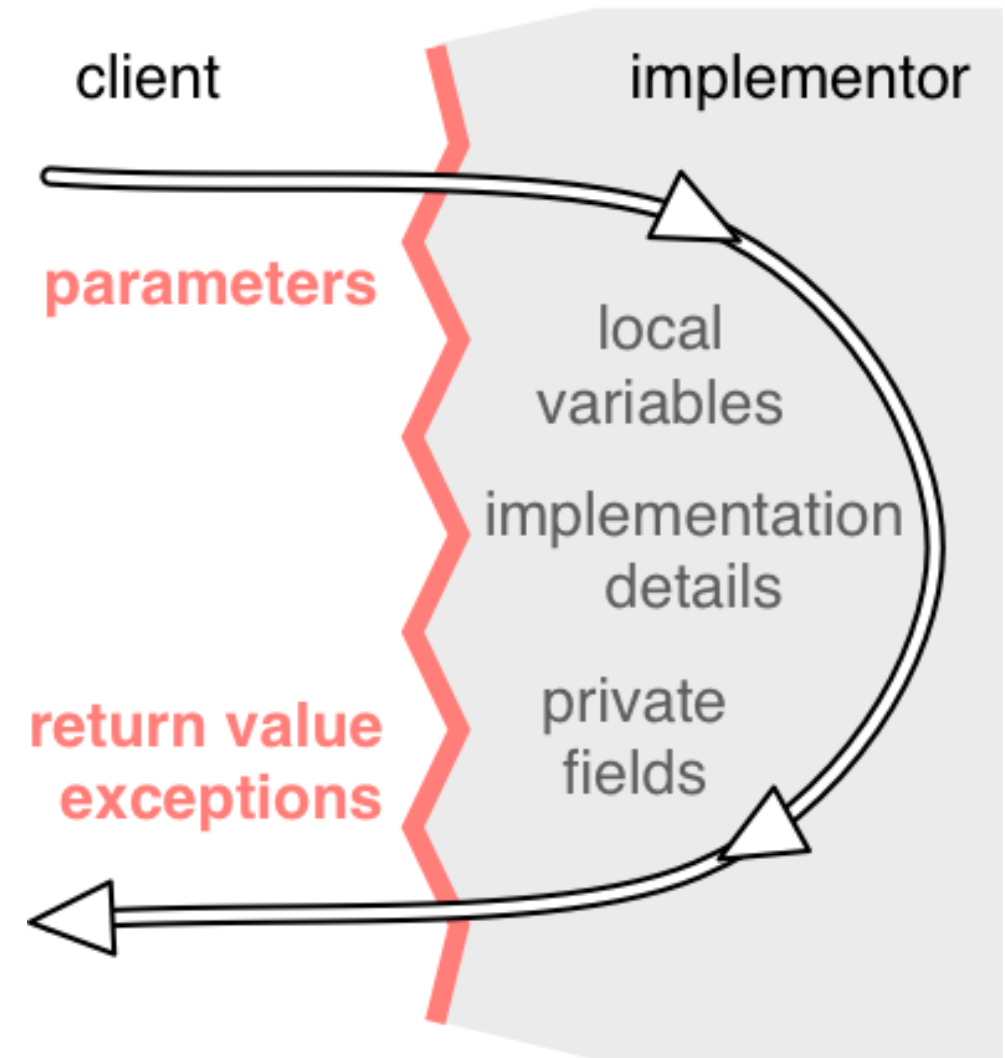
requires: val occurs exactly once in arr

effects: returns index i such that arr[i] = val

- A specification of the “find” function
- The two versions of “find” are both valid implementations of this specification!
 - As far as the client is concerned, they have the same behavior
 - One could be substituted with the other, without affecting the client’s code

Specification Must Hide Unnecessary Details

- What can appear inside the pre- & post-conditions?
- Recommended practice
 - Pre-conditions should only mention input parameters of a function (**Q. Why not output?**)
 - Post-conditions should only mention the input & output parameters
 - They should avoid mentioning **hidden/private** fields in the component (**Q. Why not?**)
 - If necessary, instead refer to **publicly visible** fields/functions



Specification Must Hide Unnecessary Details

```
public class Account {  
    private String accountID;  
    private int currBalance; // in cents  
  
    public void deposit(int dollars)  
        requires: nothing  
        effects: increase currBalance by (dollars)*100  
        { ... // implementation }  
}
```

- Q. What's undesirable about this specification of “deposit”?
- Q. How would you improve this?

How do we design a “good” specification?

Factors in Designing Specifications

- Deterministic vs. under-determined
- Declarative vs. operational
- Strong vs. weak
- General vs. restrictive

Deterministic vs. Under-determined

- A specification of a function is **deterministic** if, for any given input, it allows exactly one possible output.
- A specification is **under-determined** if, for some input, it allows multiple possible outputs.

Recall: Specification of Find

```
static int find(int[] arr, int val)
```

requires: val occurs exactly once in arr

effects: returns index i such that arr[i] = val

- An example of a **deterministic** specification
 - Only one return value is possible for any given input

Recall: Specification of Find

```
static int find(int[] arr, int val)
```

requires: val occurs exactly once in arr

effects: returns index i such that arr[i] = val

Spec ver1

```
static int find(int[] arr, int val)
```

requires: val occurs in arr

effects: returns index i such that arr[i] = val

Spec ver2

- **Q. Is the second specification (ver2) deterministic or under-determined? Why?**

Recall: Implementations of Find

```
static int find(int[] arr, int val) {  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] == val) return i;  
    }  
    return arr.length;  
}
```

```
static int find(int[] arr, int val) {  
    for (int i = arr.length - 1; i >= 0; i--) {  
        if (arr[i] == val) return i;  
    }  
    return -1;  
}
```

**These are both valid
implementations of
Spec ver1 & ver2!**

Deterministic vs. Under-determined

- A specification of a function is **deterministic** if, for any given input, it allows exactly one possible output.
- A specification is **under-determined** if, for some input, it allows multiple possible outputs.
- An **under-determined** specification is ambiguous and can result in behaviors that are “surprising” to the client
 - The client can’t rely on what output the function will return
- In general, **deterministic** specifications are preferable
 - **Design consideration:** For a given input, are multiple inputs possible? If so, how do I modify the pre- or post-condition to make it deterministic?

Declarative vs. Operational

- An **operational** specification describes **how** a function achieves its post-condition through a series of steps
- A **declarative** specification describes **what** a function achieves without saying **how**

Declarative vs. Operational: Example

```
static int find(int[] arr, int val)
```

requires: val occurs in arr

effects: examines a[0],a[1],..., in turn and returns
the index of the 1st element equal to val

- An example of an operational specification
 - **Q. What is undesirable about this specification?**
 - Expose details about how the function is implemented internally
 - Unnecessarily constrains the set of possible implementations

Declarative vs. Operational: Example

```
static int find(int[] arr, int val)
```

requires: val occurs in arr

effects: examines a[0],a[1],..., in turn and returns
the index of the 1st element equal to val

Operational

```
static int find(int[] arr, int val)
```

requires: val occurs in arr

effects: returns index i such that arr[i] = val

Declarative

- Declarative specifications tend to:
 - Be shorter, easier to understand
 - Allow a larger set of implementations
 - Give more flexibility to the implementor!

Declarative vs. Operational

- An **operational** specification describes **how** a function achieves its post-condition through a series of steps
- A declarative specification describes **what** a function achieves without saying **how**
- **Operational** specifications tend to:
 - Expose details about how the function is implemented internally
 - Unnecessarily constrains the set of possible implementations
- **Declarative** specifications are preferable
 - **Design consideration:** Is the specification describing “how” something is done? If so, can we rewrite it to say only “what” it does?

Strong vs. Weak

- Let S1 and S2 be specifications with the same pre-condition
- S1 is **stronger** than S2 if S1 provides more guarantees about the output than S2 does
 - (Mathematically, S1's post-condition is **logically stronger** than S2's post-condition)

Strong vs. Weak: Example

```
static int find(int[] a, int val)  
  requires: val occurs at least once in a  
  effects: returns index i such that a[i] = val
```

Spec ver1

```
static int find(int[] a, int val)  
  requires: val occurs at least once in a  
  effects: returns lowest index i such that a[i] = val
```

Spec ver2

- Spec ver2 is stronger than ver1, since it provides stronger guarantees about the output
- How strong is “strong enough”?
 - Depends on the client’s requirements
 - To fulfill their own tasks, does the client rely on the index being the lowest?

Strong vs. Weak: Example #2

```
static int find(int[] a, int val)
```

requires: nothing

effects: returns index i such that $a[i] = \text{val}$

Spec ver3

- **Q. What is wrong with ver3?**
 - The specification is too strong. In fact, there is no possible valid implementation for this specification!

Strong vs. Weak: Example #2

```
static int find(int[] a, int val)
  requires: nothing
  effects: returns index i such that a[i] = val
```

Spec ver3

```
static int find(int[] a, int val)
  requires: nothing
  effects: if val doesn't occur in a, returns -1
           else returns index i such that a[i] = val
```

Spec ver4

- Specification should be **as weak as** possible
 - Stronger specifications allow a smaller set of implementations & are harder to implement
 - Weaker specifications give more flexibility to the implementor

Strong vs. Weak

- Let S1 and S2 be specifications with the same pre-condition
- S1 is **stronger** than S2 if S1 provides more guarantees about the output than S2 does
 - (Mathematically, S1's post-condition is **logically stronger** than S2's post-condition)
- A specification should be **strong enough** to support the needs of the client
- A specification should also be **as weak as possible**, to provide as flexibility to the implementor
 - **Design consideration:** Is the specification providing more guarantees than needed? If so, how much can we relax them without breaking the client's code?

General vs. Restrictive

- Let S1 and S2 be specifications with the same post-condition
- S1 is **more general** than S2 if S1 puts less restrictions on the input than S2 does
 - (Mathematically, S1's pre-condition is **logically weaker** than S2's pre-condition)

General vs. Restrictive: Example

```
static int find(int[] a, int val)
```

requires: val occurs exactly once in a

effects: returns index i such that $a[i] = \text{val}$

Spec ver1

```
static int find(int[] a, int val)
```

requires: val occurs in a

effects: returns index i such that $a[i] = \text{val}$

Spec ver2

- Spec ver2 is more general than ver1, since it accepts a larger set of inputs
 - In ver1, the client must ensure that “val” occurs exactly once; ver2 imposes less burden on the client

General vs. Restrictive: Example #2

```
static int find(int[] a, int val)
```

requires: nothing

effects: **if** val doesn't occur in a, returns -1
 else returns index i such that a[i] = val

- Spec ver3 is most general (for the given post-condition)
 - Accepts any inputs; no burden on the client!
- But also shifts the burden onto the component to check input
 - Sometimes, this is undesirable, due to complexity or performance issues (e.g., consider a very large input array)
- A restriction of the pre-condition is sometimes necessary

General vs. Restrictive

- Let S1 and S2 be specifications with the same post-condition
- S1 is **more general** than S2 if S1 puts less restrictions on the input than S2 does
 - (Mathematically, S1's pre-condition is **logically weaker** than S2's pre-condition)
- A specification should be **as general as possible**
 - A pre-condition places burden on the client to satisfy it
 - Less restrictive it is, more applicable the function is
- A specification should be **restrictive** when necessary
 - **Design consideration:** What needs to be checked about the input? If the check is too expensive, can we restrict the pre-condition to rule out bad inputs?

Factors in Designing Specifications

- Deterministic vs. under-determined
- Declarative vs. operational
- Strong vs. weak
- General vs. restrictive

Exercise: Are these good specifications?

static Set union(Set s1, Set s2)

requires: “s1” and “s2” are non-empty

effects: returns a new set that contains the
elements from both “s1” and “s2”

static List sort(List l)

requires: nothing

effects: returns a new list that results from
applying merge sort to “l”

static String read(String filepath)

requires: filepath is not null

effects: opens the file at “filepath” and returns
the content of the file as a string

Interface Specifications: Takeaway

- A specification defines a contract between a component and its clients
- A specification defines a set of valid possible implementations
- A specifications should be **deterministic** rather than **under-determined**
- A specification should be **declarative** rather than **operational**
- A specification should be sufficiently **strong**, while being as **weak** as possible
- A specification should be as **general** as possible, while being **restrictive** when necessary

Summary

- Exit ticket!