CS 247: Software Engineering Principles

Interface Specifications

Readings:

Barbara Liskov and John Guttag, *Program Development in Java: Abstraction, Specification, and Object Oriented Design*

Modules and Interfaces

Module - a software component that encapsulates some design decision

e.g., function, class, package, library, component

Interface - abstract public description of some module

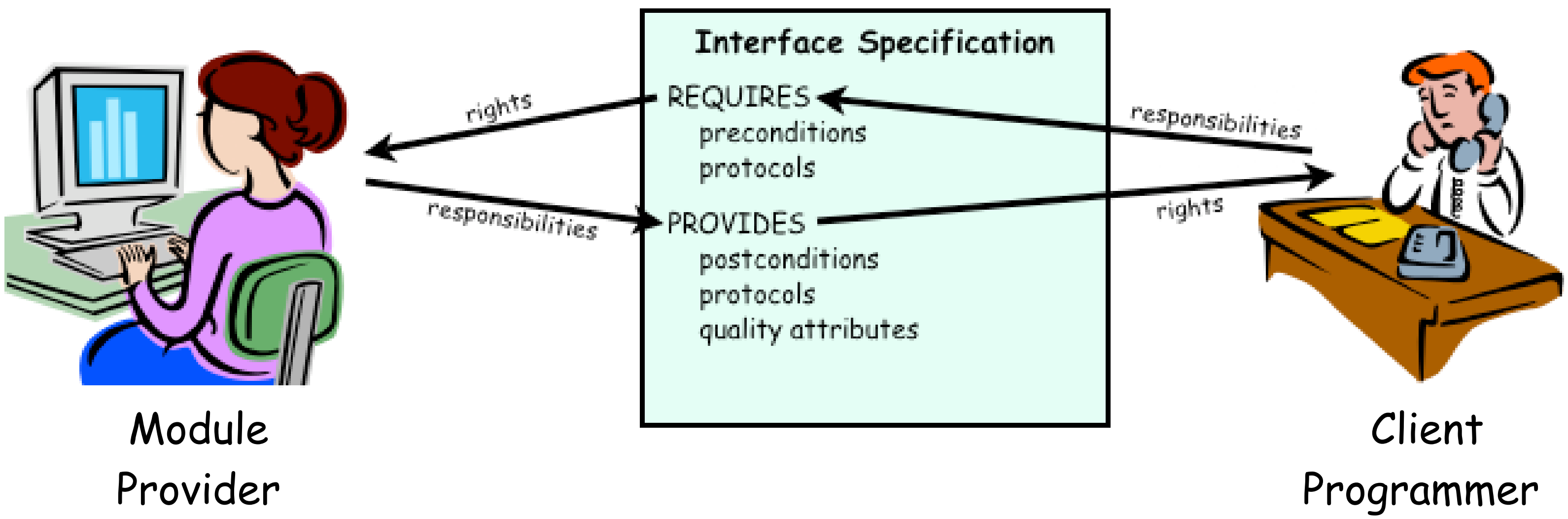
- supports information hiding (of module's details)
- reduces information overload (on client programmer)

Best Practice: An interface consists of

- a signature that specifies syntactic requirements
- a specification that describes the module's behaviour

Interface Specification

An interface specification is a contract between the module's provider and the client programmer, that documents each other's expectations.



- used to document the design of a future module
- used to document the correct usage of an existing module

CS 247 Interface Specifications

Preconditions: constraints that hold *before* the method is called (if not, anything goes):

// **requires:** necessary assumptions about the program state

Postconditions: constraints that hold *after* the method is called (assuming that the preconditions held):

// **modifies:** objects / variables that may be changed by the method

// **throws:** thrown exceptions, and conditions leading to exceptions

// **ensures:** (guaranteed) side effects on modified objects

// **returns:** describes return value

All expressions are over public variables and values

i.e., not the module's private variables

Example

```
int sumVector ( const vector<int>  & vect );
    // requires: ??
    // modifies: ??
    // ensures: ??
    // returns: ??
```

```
// return sum of vector elements
int sumVector( const vector<int> &vect ) {
  int sum = 0;
  for ( int i = 0; i < vect.size(); i++ ) {
    sum += vect[i];
    }

  return sum;
}
```

Another Example

```
int replace ( vector<int>  &vect, int oldElem, int newElem );
    // requires: ??
    // modifies: ??
    // ensures: ??
    // returns: ??
```

```
// replace element in vector; return position of new element
int replace ( vector<int> &vect, int oldElem, int newElem )
{
  for ( int i = 0; i < vect.size(); i++ ) {
    if (vect[i] == oldElem) {
      vect[i] = newElem;
      return i;
    }
  }
}
```

Yet Another Example

```
#include <string>

using std::string;

// check whether word is a substring of text
bool isSubstring( string text, string word ) {
  if ( text.length() == 0 ) return false;
  if ( word.length() == 0 ) return true;

  int wIndex = 0;
  for ( int tIndex = 0; tIndex < text.length(); tIndex++ ) {
    for ( int wIndex = 0; text[tIndex] == word[wIndex], wIndex++ ) {
      if ( wIndex == word.length() )
        return true;
    }
  }

  return false;
}
```

Specifying Exceptions

Interface specifications can supersede exception specifications

- lists all of the exceptions that can be thrown
- specifies the conditions under which each exception is thrown
- the precondition does ***not*** include the conditions that lead to a thrown exception

```
double quotient (int numerator, int denominator);
  // throws:  DivideByZero, if denominator = 0
  // returns: numerator / denominator
```

Class Example

```
class IntStack {
    // Specification Fields:
    //      top = top element of the stack

public:
  IntStack();
    // ensures: initializes this to an empty stack

  ~IntStack();
    // modifies: this
    // ensures: this no longer exists; memory is deallocated

  void push (int elem);
    // modifies: this
    // ensures: this = this@pre appended with elem; top == elem

  void pop ();
    // modifies: this
    // ensures: if this@pre is empty, then this is empty
    //          else this = this@pre with top removed

  int top();
    // requires: this is not empty
    // returns: top
```

Specifying Derivations

Derived classes inherit not only interface signatures, but also specifications.

We can specify a derived classes by either listing all of its specification fields (inherited and new), or by listing just the new fields.

When specifying an overridden method, it is best to provide the complete specification (rather than attempt to provide just extension).

Terminology

- An interface specification describes the behaviour of some software unit (e.g., function or class).
- An implementation satisfies a specification if it conforms to the described behaviour.
- The specificand set of a specification is the set of all conforming implementations.

We can ask whether an implementation conforms to a specification, or whether specification represents an implementation.

Source: Barbara Liskov and John Guttag, *Program Development in Java,* Addison-Wesley, 2001.

What are the Conforming Specifications?

```
int find ( const vector<int> &vec, int val ) {
  for ( int i=0; ; i++ )
    if ( vec[i]==val ) return i;
}
```

```
int find ( const vector<int> &vec, int val ) {
  for ( int i=0; i<vec.size(); i++ )
    if ( vec[i]==val ) return i;
  return -1;
}
```

```
int find ( const vector<int> &vec, int val ) {
  for ( int i=vec.size(); i=>0; i-- )
    if ( vec[i]==val ) return i;
  return vec.size();
}
```

Comparing Specifications

Specification A is stronger than specification B (A⇒B) iff

1. A's preconditions are equal to or weaker (less restrictive) than B's preconditions

   requires B ⇒ requires A

2. A's postconditions are equal to or stronger (promise more) than B's postconditions

   (requires B ⇒

   (ensures A ∧ returns A) ⇒ (ensures B ∧ returns B)

3. A modifies the same or fewer objects

   (requires B ⇒

   (modifies A ⊆ modifies B)

4. A throws the same or fewer exceptions

   (requires B ⇒

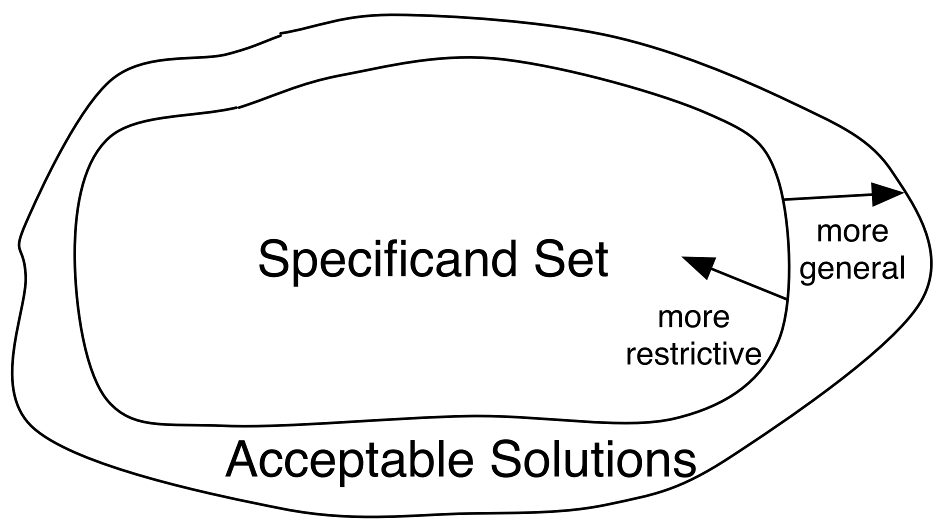
   (throws A ⊆ throws B)

How Precise Should a Specification Be?

A specification is sufficiently restrictive as long as it rules out all implementations that are unacceptable to the clients of the software module.

A specification is sufficiently general as long as it does not rule out desirable implementations.

Specificand set ⊆ Acceptable Solutions



Source: Barbara Liskov and John Guttag, *Program Development in Java,* Addison-Wesley, 2001.

Another Example

```
void sort ( list<int> &lst );
    // requires: ??
    // modifies: ??
    // ensures: ??
    // returns: ??
```

What You Should Get From This

Recognition

- Specification as a contract.
- Specification as documentation of correct usage.
- The specificand set of a specification

Comprehension

- Explain the pros and cons of specification alternatives.

Application

- Specifying the interface of a C++ method or class.
- Specifying the interface of a derived class.
- Determining whether a C++ program satisfies a specification.
- Implementing a C++ program that satisfies a specification.
- Justifying whether to check that a precondition has been met.
- Determining whether one specification is stronger than another.