Interface Specifications

Readings:
Barbara Liskov and John Guttag, *Program Development in Java: Abstraction, Specification, and Object Oriented Design*
Ch 9 Specifications

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

Module and Interfaces

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

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

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
SumVector Example

```cpp
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;
}
```

Replace Example

```cpp
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;
        }
    }
}
```

Substring Example

```cpp
#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 tl = text.length();
    int wl = word.length();
    for (int tIndex = 0; tIndex < tl; tIndex++) {
        int wIndex = 0;
        for (int ti = tIndex; ti < tl && wIndex < wl && text[ti] == word[wIndex]; ti++) {
            wIndex++;
        }
        if (wIndex == word.length() - 1)
            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

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

```cpp
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 class** 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).

**Class Example with Hidden State**

```cpp
class Account {
    // Specification fields:
    //   ActNo = unique id of Account
    //   balance = amount of money owed for phone services
    //   fee = monthly fee
public:
    explicit Account ( const AccountNo& num);
    // ensures: initializes this to an Account whose
    //   ActNo == num
    //   balance == 0
    //   fee == 30
    virtual void bill ();
    // requires: ??
    // modifies: ??
    // ensures: ??
    virtual void print() const;
    // modifies: cout
    // ensures: cout = cout@pre + this
}
```

**Example of Derived Class**

```cpp
class CheapAccount : public Account {
    // Specification fields:
    //   minutes = number of minutes called since the last bill()
    //   freemin = number of free minutes per billing period
    //   rate = charge of nonfree calls per-minute
public:
    explicit CheapAccount ( const AccountNo& num);
    // ensures: initializes this to a CheapAccount whose
    //   ActNo == num
    //   balance == 0
    //   minutes == 0
    //   fee == 30
    //   freemin == 200
    //   rate == 1
    virtual void bill();
    // requires: ??
    // modifies: ??
    // ensures: ??
}
```
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.


What are the Conforming Implementations?

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

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

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

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

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
   \[ \text{requires } B \Rightarrow \text{requires } A \]
2. A's postconditions are equal to or stronger (promise more) than B's postconditions
   \[ \text{ensures } B \Rightarrow (\text{ensures } A \land \text{returns } A) \Rightarrow (\text{ensures } B \land \text{returns } B) \]
3. A modifies the same or more objects
   \[ \text{requires } B \Rightarrow (\text{modifies } B \subseteq \text{modifies } A) \]
4. A throws the same or fewer exceptions
   \[ \text{requires } B \Rightarrow (\text{throws } A \subseteq \text{throws } B) \]
Checking Preconditions

**Obligation:** The client is responsible for ensuring that the preconditions are met before calling our code.

**Best practice:**
1. Check precondition if it is easy to do so; throw exception if not satisfied.
2. Try to detect problem and report error.

What You Should Get From This

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

**Comprehension**
- Specification considerations: specification restrictiveness / generality, comparing specifications, preconditions vs. exceptions

**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.
- Determining whether one specification is stronger than another.