Representation Invariant, Abstraction Function

Reading:
Barbara Liskov and John Guttag, *Program Development in Java: Abstraction, Specification, and Object Oriented Design*
Ch. 5.5 Aids to Understanding Implementation

Today's lecture benefits from slides by Michael Ernst (UWash) and Daniel Jackson (MIT)
Refresher: Interface Specifications

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

- valid uses of operations (preconditions)
- expected outcomes of operations (postconditions)
- expressed in terms of the module’s abstract values
representation invariant $R$

- defines the set of valid concrete values of ADT’s implementation

abstraction function $A$

- interprets legal concrete values as ADT’s abstract values
Set Represented as an Array

class Set {
public:
    static const int SET_MAX_SIZE = 10;
    class SetSizeExceeded {};
Set () : elements_(), size_(0) {}
void insert (Data* d) {
    if ( !member (d) ) {
        if ( size_ == SET_MAX_SIZE ) throw SetSizeExceeded();
        elements_[size_] = d; size_++;
    }
}
void remove (Data* d) {
    for (int i = 0; i < size_; i++)
        if ( elements_[i] == d ) {
            elements_[i] = elements_[size_-1]; size_--;
            return;
        }
}
bool member (Data* d) {
    for (int i = 0; i < size_; i++)
        if ( elements_[i] == d ) return true;
    return false;
}
Data* elements_[SET_MAX_SIZE];
int size_;
Linked List

- List
  - Header
- Node
  - Next
  - Prev
  - Element
- Data
class LinkedList {
    class Node {
        public:
            Data* element_;  
            Node* prev_;  
            Node* next_;  
            Node (Data* e, Node* p, Node* n) :
                element_(e), prev_(p), next_(n) {}  
    };
    public:
        LinkedList();  
        ...  
    private:
        Node* header_;  
            int size_;  
};

LinkedList::LinkedList () : size_(0),
        header_( new Node( nullptr, nullptr, nullptr, nullptr) ) {
            header_-->prev = header_-->next = header;  
}
void LinkedList::insert (Data* o) {
    Node* e = new Node (o, header->prev, header);
    e->prev->next = e;
    e->next->prev = e;
    size++;
}

Rep Invariants of Complex Types

If the representation uses other ADTs, it may refer to them either by their specification fields or by their operations.

Example:

```cpp
class Account {
    int balance_;  
    vector<Trans> transactionHistory_; 
    ...
};
```

Suppose the Trans ADT has a specification field `amount` that is accessible by the operation `amount()`.
Example Representation Invariants

Structural invariants of the data structure chosen:

- **trees cannot have cycles** $\left( n_1.(\text{left}+\text{right})^* \neq n_1 \right)$
- **two tree nodes cannot share the same descendent node**
  \[ n_1 \neq n_2 \Rightarrow n_1.(\text{left}+\text{right})^* \cap n_2.(\text{left}+\text{right})^* = \emptyset \]
- **in doubly-linked list**, $n_1.\text{next}.\text{prev} = n_1$

Value invariants induced by data structure / algorithms chosen:

- **no duplicate data elements:**
  \[ \text{for all nodes } n_1, n_2: (n_1 \neq n_2 \Rightarrow n_1.\text{data} \neq n_2.\text{data}) \]
- **for all nodes** $n_1$ **in the tree that have a left subtree**, $n_1.\text{left}.\text{data} < n_1.\text{data}$
Example Representation Invariants

Redundant fields are kept in sync:
• list.size = num of elements in list
• for all graph nodes n1,n2:
  n1 != n2 ⇒ (n1 ∈ n2.edges ⇒ n2 ∈ n1.edges)

Domain-specific value invariants:
• balance >= 0
• denom != 0
• 1<= length of bcode <=3  &&
  first char of bcode is capital letter  &&
  second and third char of bcode (if present) is a capital letter or a digit

Values that would otherwise cause runtime errors:
• transactionHistory_ != NULL
• list.header != NULL
• 0 <= index < 52
Where is the Error?

How can we find an error in our ADT code that is exposed by surprising behaviour in the client code?

```c
// client code
IntSet s;
s.insert(42);
s.insert(42);
s.delete(42);
if ( s.member(42) )
   // print “wrong”; ← “wrong” is printed
else
   // print “right”;
```

Is the error in `insert`?
Is the error in `delete`?
How can we know?
Inductive Reasoning

An invariant is a property that is true for the entire program—which in the case of an invariant about an object, reduces to the entire lifetime of the object.

**Structural induction:** If an invariant of an ADT is
   (1) established by constructor;
   (2) preserved by mutators; and
   (3) no representation exposure occurs,
then the invariant is true of all instances of the ADT.
Representation Exposure

• Non-private data members.

• Client has access to object inside the representation, through a different path.
  • e.g., object passed into operation is incorporated into representation, despite being accessible via another reference

• Operation outputs an object that is part of representation.
  • outputs internal collection
  • outputs iterator that points to member of internal collection
Checking Representation Invariants

Rule of thumb: check representation invariant
- on exit of constructor
- on entry and on exit of accessors and mutators

```c
void remove (Data* d) {
  checkRep();
  for (int i = 0; i < size_; i++)
    if (elements_[i] == d) {
      elements_[i] = elements_[size_-1]; size--;
      break;
    }
  checkRep();
}
void checkRep() {
  for (int i = 0; i < size_; i++) {
    assert (findIndex( elements_[i] ) == i);
  }
  for (int i = size_; i < SET_MAX_SIZE; i++) {
    assert (elements_[i] == nullptr);
  }
}
```
An abstraction function $A$ explains the correlation between concrete values in an ADT’s implementation and the ADT’s abstract values.

- maps concrete values to abstract values
Idioms for Expressing Abs Functions

Set Comprehension. \( \{ x \mid P(x) \} \) denotes the set of all elements \( x \) that satisfy the property \( P \).

Abstraction: a set \( s \) of integers in the range 0..255, with specification field \( \text{elems} \) of elements

Representation \( r \): a bit vector \( \text{bitvec} \) where value \( i \) is in the set iff \( \text{bitvec}[i] \) is true.

\[
s.\text{elems} = \{ i \mid \text{bitvec}[i] \}
\]

Recursion. Most appropriate for recursive representations.

Abstraction: a set \( s \) with specification field \( \text{elems} \)

Representation \( r \): a tree \( t \), with fields \( \text{val} \) (the value of the tree node), \( \text{left} \) (the left subtree), and \( \text{right} \) (the right subtree)

\[
s.\text{elems} = (t.\text{val} == \text{nullptr}) \ ? \ {} : \ {t.\text{val}} \cup t.\text{left.elems} \cup t.\text{right.elems}
\]
Idioms for Expressing Abs Functions

Projection. Sometimes it is easiest to divide the AF into pieces

Abstraction: a sequence sq of Data elements with spec field elems
Representation r: an array eltarray, with a max field that specifies
the highest index that points to a sequence element.

\[
\text{size}(sq) == \text{max} \quad \text{&&}
\text{for } i:1..\text{size}(sq) : \text{sq.elems}[i] = \text{eltarray}[i]
\]

By example. Last resort

Abstraction: a sequence sq of Data elements with spec field elems
Representation r: a linked list p, with an empty header node, and
next field that leads to the next node in the sequence.

\[
\text{sq.elems} =
\text{if } p.\text{size} == 0 \text{ then } <> \quad // \text{empty list}
\text{else } < p->\text{header}->\text{next}->\text{element}, \]
\[ p->\text{header}->\text{next}->\text{next}->\text{element}, \ldots >
\]

(sequence of length p.size, whose ith element =
\[ p->\text{header}->\text{next}^i->\text{element})\]
Benevolent Side Effects

Different implementation of member:

```cpp
bool member (Data* d) {
    int i = findIndex(d);
    if ( i == -1) return false;

    // move-to-front optimization
    swap ( elements_[0], elements_[i] );

    return true;
}
```

Move-to-front speeds up related lookup requests. Mutates the concrete value, but not the abstract value.

Example: \( \text{AF}(5, 20, 10) = \text{AF}(10, 20, 5) = \{5, 10, 20\} \)
Card Example

```cpp
enum Suit { CLUB, DIAMOND, HEART, SPADE };
enum Rank { ACE, TWO, THREE, FOUR, FIVE, ..., TEN, JACK, QUEEN, KING };

//**************************************************************************************************
// Card represents an immutable playing card
//
// Specification fields:
//  suit: {Clubs, Diamonds, Hearts, Spades}
//  rank: {Ace, 2, 3, ..., 10, Jack, Queen, King}
//**************************************************************************************************

class Card{
public:
    Card(Suit s, Rank r);
    // ensures: initializes *this so that getSuit()==s and getRank()==r

    Suit suit() const;
    // returns: *this.suit  -- "suit" refers to the spec field

    Rank rank() const;
    // returns: *this.rank  -- "rank" refers to the spec field

private:
    int index_;

    void checkRep() const;
    // ensures: no change if *this satisfies rep invariant; else raise assertion

bool operator==(const Card& c1, const Card& c2);
// returns: true if c1.suit==c2.suit && c1.rank==c2.rank; else false
```
Take Aways

**Recognition**
- Motivation and uses of representation invariant
- Know where to perform runtime checking of a representation invariant
- Motivation and uses of abstraction functions

**Comprehension**
- Argue the pros and cons of including runtime checking of representation invariants in software that is released

**Application**
- Specify the representation invariant of an ADT implementation
- Use inductive reasoning to argue informally that an operation is correct
- Specify the abstraction function of an ADT implementation