Representation Invariant, Abstraction Function

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**

**Interface Specification**

**Refresher: Interface Specifications**

**ADT Documentation**

**Set Represented as an Array**

```java
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

```
class LinkedList {
  class Node {
    public:
      Data* element_;  // Data field
      Node* prev_;    // Node field
      Node* next_;    // Node field
      Node (Data* e, Node* p, Node* n) :
        element_(e), prev_(p), next_(n) {}  // Constructor
    }
  public:
    LinkedList();  // Constructor
    ...  // Other methods
  private:
    Node* header_;  // Node pointer
    int size_;     // Integer
  }

LinkedList::LinkedList () : size_(0), header_(new Node( nullptr, nullptr, nullptr)) {
  header_->prev = header_->next = header;  // Initialize header
}
```

Linked List (Revisited)

```
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:
```
class Account {
  int balance_;  // Integer field
  vector<Trans> transactionHistory_;  // Vector of Trans
  ...  // Other members
};
```

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 (\(n_1.(left+right)^* \neq n_1\))
- two tree nodes cannot share the same descendent node
  \(n_1\neq n_2 \Rightarrow n_1.(left+right)^* \cap n_2.(left+right)^* = \emptyset\)
- in doubly-linked list, \(n_1.next.prev = n_1\)

Value invariants induced by data structure / algorithms chosen

- no duplicate data elements:
  
  for all nodes \(n_1, n_2\):  
  \(n_1\neq n_2 \Rightarrow n_1.data \neq n_2.data\)

- for all nodes \(n_1\) in the tree that have a left subtree,
  \(n_1.left.data < n_1.data\)

Example Representation Invariants

Redundant fields are kept in sync:

- \(list.size = \text{num of elements in list}\)
- for all graph nodes \(n_1, n_2:\)
  \(n_1 \neq n_2 \Rightarrow (n_1 \in n_2.edges \Rightarrow n_2 \in n_1.edges)\)

Domain-specific value invariants:

- \(balance >= 0\)
- \(denom != 0\)
- \(1 <= \text{length of bcode} <= 3 \&\& \text{first char of bcode is capital letter} \&\& \text{second and third char of bcode (if present) is a capital letter or a digit}\)

Values that would otherwise cause runtime errors:

- \(\text{transactionHistory_} \neq NULL\)
- \(\text{list.header} \neq NULL\)
- \(0 <= \text{index} < 52\)

Where is the Error?

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

```cpp
// 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

Abstraction Function

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

  \[ s.\text{elems} = \{ t.\text{val} == \text{nullptr} \} ? \{ \} : \{ t.\text{val} \} \]

  \( t.\text{left.\text{elems}} \)

  \( t.\text{right.\text{elems}} \)
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.

$$
size(sq) == max \&\& \\
for i:1..size(sq) : sq.elems[i] = eltarray[i]
$$

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.

$$
sq.elems = \\
if p.size == 0 then <> // empty list \\
else p->header->next->element, \\
 p->header->next->next->element, ... > \\
(sequence of length $p.size$, whose $ith$ element = \\
 $p->header->next^i->element$)
$$

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: $AF(5, 20, 10) = AF(10, 20, 5) = \{5, 10, 20\}$
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