Abstract Data Types I

Spring 2013
Today’s Topics

Abstract Data Types
Representation Independence
Representation Exposure
Synonyms for Abstraction

**Abstraction is a software engineering principle**
- Spans many ideas & different ways of looking at the why or what
- Different names/ideas each emphasize a different aspect of abstraction
- Same central idea

**Abstraction**
- Omit or hide low-level details with simpler high-level idea

**Modularity**
- Using small components to design a larger system, with each component being self-contained, implementable, reusable, and testable on its own
Synonyms for Abstraction

**Encapsulation**
- Building components/modules in a way that shields them from bugs in other parts of the program. Modules responsible for managing their internal state/behavior.

**Separation of Concerns**
- Making each concern/feature/task the responsibility of a single module, and preventing a single module from too much functionality. Conversely, prevent feature/task being spread out among too many modules.

**Information Hiding**
- Hiding implementation details of each component from the rest of the system, so internal details may be changed without impacting rest of design.
User-defined Types: History

Early languages had built-in types & procedures

- Allow users define new procedures, but not new types
- E.g. instead of `addVec(Vector a, Vector b)` users needed to do `addVec(int a1, int a2, int b1, int b2)`
- Later languages allowed combining built-in types into `structures`

Built-in types forced users to think about & manipulate `machine representation` of data.

Abstract types

- Allow users to create new types, not just new procedures
- Theory from Liskov, Hoare, Guttag & many others
Data Abstraction

An abstract data type is characterized by operations that can be performed on it.
- Number = something that can be added/multiplied/etc
- String = something that can be concatenated etc

Key idea
- Instead of thinking about how a type stores values, programmers only need to think about operations on the type

Result: Many modern programming languages blur distinction between user-defined & language-defined types
Classifying Types

We’ve already seen one way to characterize types

**Mutable**

- Includes operations that alter the object such that other subsequent operations on the same object give different results
- E.g. StringBuilder from last week’s lectures

**Immutable**

- Object is not designed to change once created
- Often, new objects created instead of changing existing one
- E.g. String
Classifying Operations on Types

Four major kinds of operations.

Constructors
- Create new objects of a type
- May take arguments, but not of the same type being created

Producers
- Create new objects from old objects of the same type
- E.g. concat() creates a new String object from two existing ones
Classifying Operations on Types

Four major kinds of operations.

Mutators

➢ Alter the object
➢ E.g. Java List objects have an add() operation

Observers

➢ Take as input an object of the abstract type and return a different kind of object
➢ E.g. The size() method of Java Lists
Classifying Operations on Types

Four major kinds of operations.

For an abstract type T and other types t

Constructor: \( t^* \rightarrow T \)
Producers: \( T^+, t^* \rightarrow T \)
Mutators: \( T^+, t^* \rightarrow \text{void} \)
Observers: \( T^+, t^* \rightarrow t \)

Useful classification, but not perfect

- In particular, some operations may be mutator+observer or mutator +producer
- In general, mutator is a stronger term than others
- Operation that mutates the object is mutator, regardless of what it returns
Classifying Operations on Types

Example: Java’s double

Creators
- Double literals (1.2, 3.0, etc)

Producers
- Arithmetic operations (+, -, *, etc)

Observers
- Comparison operators (> , ==, etc)

Mutators
- None (why?)
Classifying Operations on Types

Example: Java List

Creators
- ArrayList constructor, LinkedList constructor, Collections.singletonList()

Producers
- Collections.unmodifiableList()

Observers
- size(), get()

Mutators
- add(), remove(), addAll(), Collections.sort()
Abstract Data Types: The Main Point

Now we can create new types that behave (almost) like built-in types, but with powerful domain-specific operations

Necessary to design our types with safety, power, and coherence in mind

- Careful design will prevent many kinds of bugs
- Prevent clients from inadvertently misusing the type
Designing Abstract Data Types

Need to choose a good set of operations & their behavior

Rules of thumb

- Better to have few, simple operations that can be combined in powerful ways than many complex operations.
- Each operation should have succinct, well-defined purpose, instead of lots of special cases.
- Set of operations should be adequate for what clients want to do & should convey information clients likely to want.
- If a type is generic it should contain no domain-specific operations. If a type is domain-specific, it should not have too general operations.
Choosing a Representation

For the client of a type, operations are the important thing, but if designing a type, must think about representation.

What do we mean by representation?

- Actual data structures used internally to support operations on an abstract data type
- In practice, the collection of fields of the Java object, where each field is a Java type (including of the same type as the ADT object)
Choosing a Representation

Suppose Java had no List interface. Let's design a List ADT

Represent as linked list
- Each entry in the list is an object called Entry
- List itself points to first object
- Each entry points to next entry

Represent as an array
- A List of Foo objects is just represented by an array of type Foo[]

Choice of representation is important in ADT design, even if clients never see it.
Representation Independence

Suppose we know List is implemented with an array (elementData) as the representation.

Vector v = ...
List l = new List ();
v.copyInto (l.elementData);

Like all hacks, this will work temporarily.

What happens if the implementer changes representation to a linked list?
Suppose we know List is implemented with an array (elementData) as the representation.

Vector v = ...
List l = new List ();
v.copyInto (l.elementData);

Suppose internally, List keeps track of mySize and size() just returns the instance variable mySize.

What now?
Representation Independence

The use of an abstract type should be independent of its representation

- Changes in representation should not impact any client code
- Requires all operations to have fully-specified preconditions, post conditions, and frame conditions

Representation exposure is bad

- Code outside of the class can modify the internal representation

Java has (some) language features to prevent representation exposure

- E.g. private or final instance variables
- But not sufficient to prevent all representation exposure
Preserving Invariants

A good ADT must preserve its own invariants, and prevent clients from breaking them.

Invariant – property of a program or object that is always true

- Immutability is one we’ve seen before
- Other invariants may express relationships between instance variables etc.

Preserving invariants makes reasoning about the use of the ADT easier

- Only need to reason about ADT itself, not its use (since use cannot violate invariant)
- E.g. String immutability means we don’t worry about disallowed modifications to Strings in our programs
Preserving Invariants: Immutability

```java
public class Transaction {
    // want this to be an immutable class
    public int amount;
    public Calendar date;

    public Transaction(int amount, Calendar date) {
        this.amount = amount;
        this.date = date;
    }
}
```

Transaction t = new Transaction(10, new Calendar());
t.amount += 10;

Total representation exposure, so clients directly access fields & can change them.
Preserving Invariants: Immutability

```java
public class Transaction {
    private final int amount;
    private final Calendar date;

    public Transaction(int amount, Calendar date) {
        this.amount = amount;
        this.date = date;
    }

    public int getAmount() { return amount; }

    public int getDate() { return date; }
}
```

Use access control to prevent representation exposure

➢ But is representation actually not exposed?
Preserving Invariants: Immutability

```java
public class Transaction {
    private final int amount;
    private final Calendar date;

    public Transaction(int amount, Date date) {
        this.amount = amount;
        this.date = date;
    }

    public int getAmount() { return amount; }
    public int getDate() { return date; }
}

/** @return a transaction of same amount as t, one month later */
public static Transaction makeNextPayment(Transaction t) {
    Calendar d = t.getDate();
    d.add(Calendar.MONTH, 1);
    return new Transaction(t.getAmount(), d);
}
```
Preserving Invariants: Immutability

/** @return a transaction of same amount as t, one month later */
public static Transaction makeNextPayment(Transaction t) {
    Calendar d = t.getDate();
    d.add(Calendar.MONTH, 1);
    return new Transaction (t.getAmount(), d);
}

get\_Date() returns reference to the same object as t’s date, so mutation changes the t’s date
Preserving Invariants: Immutability

```java
public Calendar getDate() {
    return (Calendar) date.clone();
}
```

**We’ll use defensive copying to prevent this exposure**

Are we done now?
Preserving Invariants: Immutability

```java
/**
 * @return a list of 12 monthly payments of identical amounts */
 public static List<Transaction> makeYearOfPayments(int amount) {
    List<Transaction> list = new ArrayList<Transaction>();
    Calendar date = new GregorianCalendar();
    for (int i=0; i<12; i++) {
        list.add(new Transaction(amount, date));
        date.add(Calendar.MONTH, 1);
    }
    return list;
}
```

Still have rep exposure.
➢ Do defensive copying in constructor

ADT design must be careful with respect to mutability
Structural Induction

How to establish invariants?

- Must ensure the invariant is true when the object is constructed
- All other operations on the object must preserve invariant
- But also want to avoid representation exposure

Structural Induction

- If an invariant of an abstract data type is
  1. Established by its constructors
  2. Preserved by producers, mutators, and observers, and
  3. No representation exposure occurs

Then the invariant is true for all instances of the abstract data type.
Complex Invariants

Not all invariants are about immutability

- And immutable objects don’t mean internal methods are barred from modifying internal state

Example: Implementing Java’s String

- We choose as a representation: private char[] a
- Decide to implement a method maxChar which returns the character in the string that is latest in alphabetical order (or ASCII order)
Complex Invariants

/**
 * Return the character in the string with the highest ASCII code.
 * @return the character, or 0 for an empty string
 */

public char maxChar() {
    char ch = 0;
    for (int i=0; i<a.length; i++) {
        if (a[i] > ch)
            ch = a[i];
    }
    return ch;
}

Unfortunately, this needs to iterate through the entire string. So let’s optimize.
Complex Invariants

```java
private char actualMaxChar;
private boolean maxCharSet;

/**
 * Return the character in the string with the highest ASCII code.
 * @return the character, or 0 for an empty string
 */
public char maxChar() {
    if (maxCharSet)
        return actualMaxChar;
    char ch = 0;
    for (int i=0; i<a.length; i++) {
        if (a[i] > ch)
            ch = a[i];
    }
    actualMaxChar = ch;
    maxCharSet = true;
    return ch;
}
```

Is the immutability of our String class violated by adding this method?
Complex Invariants

**Invariant**
- Contents of array \( a \) are never modified after object initialization, and if \( \text{maxCharSet} \) is true, then \( \text{actualMaxChar} \) is the character with the highest ASCII value.

**Established by constructors?**
- Constructor assigns array \( a \) a value, so first part of invariant is true
- Must set \( \text{maxCharSet} \) to false in constructor for invariant to be true

**Preserved by producers, mutators, observers?**
- \( \text{maxChars} \) only reads from \( a \)
- If \( \text{maxCharSet} \) is true, we don’t change anything—so invariant is true
- If \( \text{maxCharSet} \) is false, we set \( \text{actualMaxChar} \) then set \( \text{maxCharSet} \) to true, so invariant preserved
Complex Invariants

Invariant

- Contents of array a are never modified after object initialization, and if maxCharSet is true, then actualMaxChar is the character with the highest ASCII value.

No representation exposure occurs?

- All storage is private, and only mutable field is a, which we never return or store a reference to anywhere else

Hand-wavy version of structural induction

- But this is the kind of reasoning one must do for ADTs
- Benefit: establish representation independence
  - Ready for changed implementations
  - Our changes won’t impact clients
Testing & Rep Independence

Contextual equivalence

- Two classes are contextually equivalent, if, given the same arbitrary sequence of method calls, we get the same results.

In other words, can test if multiple implementations actually representation independent

- Just have (many) test cases that compare outputs for arbitrary sequences of operations.
- Also compare initial states by testing output of constructors.
- Definitely not a formal way to verify representation independence (or correctness!)
- But useful way to think when considering whether your implementation is representation independent.
Summary

Abstract Data Types allow clients to think about types in terms of operations allowed on them

Enable designers to create types with implementations that can change, but need representation independence

Four major kinds of operations

Invariant preservation: difficult to do, complex to reason about, but important to avoid representation exposure and/or broken objects
Next Lecture

Java Interfaces
More on theory of abstract data types
Abstraction Functions