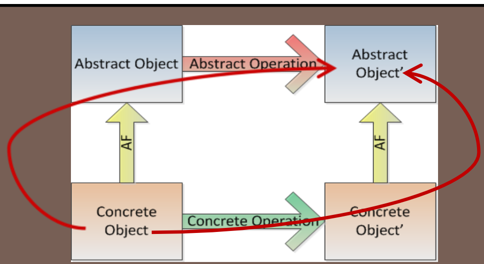


CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
ABSTRACT DATA TYPES II

Autumn 2011

Kinds of ADT operations (abstract)

creators & producers	mutators	observers
make new values of an ADT ○ Creators return new ADT values (analogous to constructors) – effects not modifies ○ Producers are operations on the type that return new values	modify a value of the ADT (without affecting reference equality; that is, == still holds)	return information to distinguish among values of an ADT

- Mutable ADTs: creators, observers, and mutators
- Immutable ADTs: creators, observers, and producers

Three examples

- A primitive type as an (immutable) ADT
- A mutable type as an ADT
- An immutable type as an ADT

Primitive data types are ADTs

- `int` is an immutable ADT
  - creators `0, 1, 2, ...`
  - producers `+ - * / ...`
  - observer `Integer.toString(int)`
- Peano showed we only need one creator for basic arithmetic – why might that not be the best programming language design choice?

Poly: overview

```
/**
 * A Poly is an immutable polynomial with
 * integer coefficients.  A typical Poly is
 *           c0 + c1x + c2x2 + ...
 **/
class Poly { …
```

- Overview states whether mutable or immutable
- Defines abstract model for use in specs of operations
  - Often difficult and always vital! Appeal to math if appropriate
  - Give an example (reuse it in operation definitions)
- State in specification is abstract not concrete (in the `Poly` spec above, the coefficients are the abstract state)

Poly: creators

```
// effects: makes a new Poly = 0
public Poly()

// effects: makes a new Poly = cx^n
// throws: NegExponent when n < 0
public Poly(int c, int n)
```

- New object, not part of pre-state: in `effects`, not `modifies`
- Overloading: distinguish procedures of same name by parameters (Ex: two `Poly` constructors)

Poly: observers

```
// returns: the degree of this: the largest
//    exponent with a non-zero coefficient; if
//    no such exponent exists, returns 0
public int degree()

// returns: the coefficient of
//    the term of this whose exponent is d
public int coeff(int d)

// Poly x = new Poly(4, 3);
// x.coeff(3) returns 4
// x.degree() returns 3
```


Notes on observers

- Observers **`return`** values of other types to discriminate among values of the ADT
- Observers *never* **`modify`** the abstract value
- They are generally described in terms of **`this`** – the particular object being worked on (also known as the receiver or the target of the invocation)

Poly: producers

```
// returns: this + q (as a Poly)
public Poly add(Poly q)

// returns: the Poly = this * q
public Poly mul(Poly q)

// returns: -this
public Poly negate()

// Poly x = new Poly(4, 3);
// Poly y = new Poly(3, 7);
// Poly z = x.add(y);
// z.degree() returns 7
// z.coeff(3) returns 4
// (z.negate()).coeff(7) returns -3
```


Notes on producers

- Common in immutable types like `java.lang.String`
  - Ex: `String substring(int offset, int len)`
- No side effects
  - That is, they can affect the program state but cannot have a side effect on the existing values of the ADT

IntSet, a mutable datatype

```
// Overview: An IntSet is a mutable, unbounded
// set of integers { x1, ..., xn }.
class IntSet {

  // effects: makes a new IntSet = {}
  public IntSet()
  …
```


IntSet: observers

```
// returns: true if x ∈ this
//          else returns false
public boolean contains(int x)

// returns: the cardinality of this
public int size()

// returns: some element of this
// throws: EmptyException when size()==0
public int choose()
```

IntSet: mutators

```
// modifies: this
// effects:  this_post = this_pre ∪ {x}
public void add(int x)     // insert an element

// modifies: this
// effects:  this_post = this_pre - {x}
public void remove(int x)
```


Notes on mutators

- Operations that modify an element of the type
- Rarely modify anything other than **`this`**
  - Must list **`this`** in modifies clause if modified
- Typically have no return value
- Mutable ADTs may have producers too, but that is less common

Quick Recap

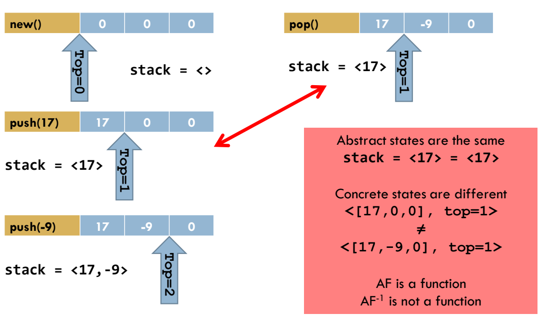
- The examples focused on the abstraction specification – with no connection at all to a concrete implementation
- To connect them we need the abstraction function (AF), which maps values of the concrete implementation of the ADT (for 331, instances of a Java class) into abstract values in the specification
- The representation invariant (RI) ensures that values in the concrete implementation are well-defined – that is, the RI must hold for every element in the domain of the AF

The AF is a **function**

- Why do we map concrete to abstract rather than vice versa?
- It's not a function in the other direction.
  - Ex: lists **`[a,b]`** and **`[b,a]`** both represent the set **`{a,b}`** in **`CharSet`**
- It's not as useful in the other direction – we can manipulate abstract values through abstract operations

Brief example

Abstract stack with array and "top" index implementation



Writing an abstraction function

- The domain: all representations that satisfy the rep invariant
- The range: can be tricky to denote
  - For mathematical entities like sets: relatively easy
  - For more complex abstractions: give them fields
    - AF defines the value of each "specification field"
- The overview section of the specification should provide a way of writing abstract values
  - A printed representation is valuable for debugging

Checking the rep invariant

```
public void delete(Character c) {
  checkRep();
  elts.remove(c)
  checkRep();
}
…
/** Verify that elts contains no duplicates. */
/*  throw an exception if it doesn't */
private void checkRep() {
  for (int i = 0; i < elts.size(); i++) {
    assert elts.indexOf(elts.elementAt(i)) == i;
  }
}
```

From Friday's CharSet example

Alternative

- `repOK()` returns a `boolean`
- Callers of `repOK()` check the return value
- Why do this instead of `checkRep()`?
- More flexibility if the representation is invalid

Checking rep invariants

- Should code always check that the rep invariant holds?
  - Yes, if it's inexpensive (in terms of run-time)
  - Yes, for debugging (even when it's expensive)
  - It's quite hard to justify turning the checking off
  - Some private methods need not check – why?

Practice defensive programming

- Assume that you will make mistakes – if you're wrong in this assumption you're (a) superhuman and (b) ahead of the game anyway
- Write code designed to catch them
  - On entry: check rep invariant *and* check preconditions
  - On exit: check rep invariant *and* check postconditions
- Checking the rep invariant helps you discover errors
- Reasoning about the rep invariant helps you avoid errors
  - Or prove that they do not exist!
  - More about reasoning later in the term

Representation exposure redux

- Hiding the representation of data in the concrete implementation increases the strength of the specification contract, making the rights and responsibilities of both the client and the implementer clearer
- Defining the fields as `private` in a class is *not* sufficient to ensure that the representation is hidden
- *Representation exposure* arises when information about the representation can be determined by the client

Representation exposure: example

```
Point p1 = new Point();
Point p2 = new Point();
Line line = new Line(p1,p2);
p1.translate(5, 10);  // move point p1
```

- Is `Line` mutable or immutable?
- It depends on the implementation!
  - If `Line` creates an internal copy: immutable
  - If `Line` stores a reference to `p1,p2`: mutable
- So, storing a mutable object in an immutable collection can expose the representation

Ways to avoid rep exposure

- Exploit immutability – client cannot mutate

```
Character choose() { // Character is immutable
  return elts.elementAt(0);
}
```

- Make a copy – mutating a copy in the client is OK

```
List<Character> getElts() {
  return new ArrayList<Character>(elts);
}
```

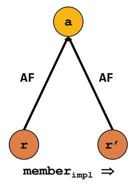
- Make an immutable copy – client cannot mutate

```
List<Character> getElts() {
  return Collections.unmodifiableList<Character>(elts);
}
```


Benevolent side effects: example

- Alternative implementation of **member** – an observor

```
boolean member(Character c1) {
  int i = elts.indexOf(c1);
  if (i == -1)
    return false;
  // move-to-front to
  // speed up repeated member tests
  Character c2 = elts.elementAt(0);
  elts.set(0, c1);
  elts.set(i, c2);
  return true;
}
```



- Mutates rep, but not abstract value – AF maps both `r` and `r'` to abstract value `a`
- Nor does it violate the rep invariant
- Arguably, the client can learn something about the representation – at the same time, this is a relatively benign case of rep exposure

A half-step backwards

- Why focus so much on invariants (properties of code that do not – or are not supposed to – change)?
- Why focus so much on immutability (a specific kind of invariant)?

- Software is complex – invariants/immutability etc. allow us to reduce the intellectual complexity to some degree
- That is, if we can assume some property remains unchanged, we can consider other properties instead
- Simplistic to some degree, but reducing what we need to focus on in a program can be a huge benefit

Next steps

- Assignment 2(a)
  - Due tonight 11:59PM
- Assignment 2(b)
  - Out tomorrow AM
  - Due Friday 11:59PM
- Lectures (swap from original plan)
  - Subtyping/subclassing (W)
  - Modular design (F)