

DbC: design by contract

- Concepts introduced by Bertrand Meyer for the Eiffel language in 1980's
- A *contract* of a procedure or object is a pair (precondition, postcondition) such that
 - precondition** : which proof obligations required by the invoker expressing input conditions or the initial state of an object
 - postconditions** benefits expected when the procedure terminates or the final state of an object
- a contract is violated if who invokes does not respect the preconditions or if the postconditions are not satisfied

```
class interface ACCOUNT

create
    make

feature

    balance: INTEGER
    ...

    deposit (sum: INTEGER)
        -- Deposit sum into the account.
        require
            sum >= 0
        ensure
            balance = old balance + sum

    withdraw (sum: INTEGER)
        -- Withdraw sum from the account.
        require
            sum >= 0
            sum <= balance - minimum_balance
        ensure
            balance = old balance - sum

    may_withdraw ...

end -- ACCOUNT
```

Objects in Dafny

- an object is an instance of a class
- it is accessed by reference and is stateful
- the class defines state components of an object, *fields*, and operations, *methods*: *members* of the class.

(Objects) Class invariants

A *class invariant* is a condition that defines the valid states (members values) of the objects (class instances).

Besides pre/post conditions each method, inside a class, has to respect the class invariant:

- Each constructor needs to ensure that all invariants hold at the end of its execution.
- Each method needs to ensure that all invariants hold at the end of its execution, assuming that they hold in the beginning.

```
class Account
{
    var balance: int;

    constructor(b:nat)
    ensures balance==b
    {
        balance := b;
    }
    method Withdraw(val: int)
        requires Valid() && balance >= 0 && balance >= val
        ensures Valid() && balance == old(balance) - val
        modifies this
    {
        balance := balance - val;
    }
    method AddCredit (val: int)
        requires Valid() && val >= 0;
        ensures Valid() && balance == old(balance) + val
        modifies this
    {
        balance := balance + val;
    }
    ghost predicate Valid()
        reads this
    {
        balance >= 0
    }
}

class
class Stack
```

```

{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    constructor (cap: nat)
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate isEmpty()
    {
        size == 0
    }
    predicate isFull()
    {
        size == elems.Length
    }
    method push(x : T)
    {
        elems[size] := x;
        size := size + 1;
    }
    function top() : T
    {
        elems[size-1]
    }
    method pop()
    {
        size := size-1;
    }
}

```

Objects

- fields: `elems`, `size`
- constructor: called when objects are created (initialization)
- methods: `method`
- initialization: `var s:= new Stack(3)`

Stack

```

method {:verify false} testStack()
{

```

```

var s := new Stack(3);
assert s.isEmpty();
s.push(1);
s.push(2);
s.push(3);
assert s.top() == 3;
assert s.isFull();
s.pop();
assert s.top() == 2;
}

```

Objects

- classes (or *trait*) can extend other classes
- Methods can be declared static
- An object is referred by `this`
- and their members by `.".": this.size` (but `this` can be omitted)
- A variable of a class (instance) can be `null` if declared with `?` after the class name
- construct can have names
- The type of all classes is `object`
- Immutable fields can be declared with `const`
- Classes may include `predicates` or `functions`.

```

class A {
    constructor()
}

method main(){
var a: A := new A();
var o:object = a;
var n:A? = null;
}

```

Class invariant- Valid

- predicate `Valid()` describes the class invariant
- `this` invariant is required (precondition) for all methods/functions (except the constructors)
- and needs to hold at the end of the execution (postcondition) by all constructors and modifiers (any method)
- with the attribute `:autocontracts` Dafny adds automatically `requires/ensures` for `Valid()` and other required contracts.

framing for objects

- Object are mutable and stored in the *heap*
- use *dynamic frames*
- It is necessary to state for *function* (or *predicate*) which parts are **reads** and for *method* which parts are modified **modifies** ()
- if only one field of an object is read that can be expressed with ‘: **reads** this’**Value**
- Side note:
- The name Dafny comes from a permutation of some letters in ”DYNAmic Frames”

Counter

```
class Counter {  
var Value: int;  
  
predicate Valid()  
reads this'Value  
{  
Value >= 0  
}  
constructor Init ()  
ensures Value == 0  
ensures Valid()  
{ Value := 0; }  
method GetValue() returns (x: int)  
modifies this  
requires Valid()  
ensures x == Value  
{x := Value; }
```

Counter

```
method Inc ()  
modifies this  
requires Valid()  
ensures Value == old(Value)+1  
ensures Valid()  
{Value := Value + 1; }  
  
method Dec ()  
modifies this
```

```

requires Value > 0 && Valid()
ensures Value == old ( Value ) -1 && Valid()
{Value := Value -1;}
}

```

Allocation of new objects

- if during the execution of a method new objects are allocated in the *heap* that should be indicated using `fresh`.

```

method NewSorted(n: nat) returns (v: array<int >)
ensures fresh ( v )
ensures Sorted ( v )
ensures v.Length = n
{}

```

Invariant of Stack

```

ghost predicate Valid()
reads this
{
size <= capacity == elems.Length
}

constructor (cap: nat)
    requires cap > 0
    ensures fresh(elems)
    ensures Valid()
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }

predicate isEmpty()
    reads this
    {
        size == 0
    }

predicate isFull()
    reads this
    {
        size == elems.Length
    }

```

```

method push(x : T)
    modifies this, elems
    requires Valid() && size < capacity
{
    elems[size] := x;
    size := size + 1;
}

function top(): T
    requires Valid() && size > 0
    reads this, elems
{
    elems[size-1]
}

method pop()
    requires Valid() && size > 0
    modifies this
    ensures Valid()
    ensures elems[..size] == old(elems[..size-1])
{
    size := size-1;
}
}

:autocontracts

class{:autocontracts} Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    ghost predicate Valid()
    reads this
    { size <= capacity == elems.Length }
    constructor (cap: nat)
    ensures cap == capacity && size == 0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate method isEmpty()
    {

```

```

        size == 0
    }
    predicate method isFull()
    {
        size == elems.Length
    }

:autocontracts

method push(x : T)
requires size < capacity
modifies this, elems
ensures elems[..size]==old(elems[..size])+[x]
{
    elems[size] := x;
    size := size + 1;
}

function top(): T
requires size > 0
{
    elems[size-1]
}
method pop()
requires size > 0
modifies this
ensures elems[..size] == old(elems[..size-1])
{
    size := size-1;
}
}

```

Inheritance

A subclass cannot break the contract of their superclass, but

- precondition can be weakened
- Postcondition can be strengthened

Example: The class `Circle` inherits from the `Shape` class.

Traits (abstract superclass)

```

trait Shape
{
    var center: (real, real);

    ghost predicate Valid()
        reads this

    function getSizeX(): real
        requires Valid()
        reads this

    function getSizeY(): real
        requires Valid()
        reads this

    method resize(factor: real)
        requires factor > 0.0 && Valid()
        modifies this
        ensures Valid()
        ensures getSizeX() == factor * old(getSizeX())
        ensures getSizeY() == factor * old(getSizeY())
        ensures center == old(center)
}

class Circle extends Shape
{
    var radius: real;
    predicate Valid()
        reads this
    {
        radius > 0.0
    }
    constructor Circle(center: (real, real), radius: real)
        requires radius > 0.0
        ensures this.center == center && this.radius == radius && Valid()
    {
        this.center := center;
        this.radius := radius;
    }
    function getSizeX(): real
        requires Valid()
        reads this
    {
        radius
    }
    function getSizeY(): real

```

```

    requires Valid()

    reads this
{
    radius
}
}

```

Subclasse

```

method resize(factor: real)
    requires factor != 0.0 && Valid()
    modifies this
    ensures center == old(center)
    ensures radius == abs(factor) * old(radius)
    ensures Valid()
{
    radius := abs(factor) * radius;
}

function abs(x: real): real
{
    if x >= 0.0 then x else -x
}
}

```

See that

$$\begin{aligned}
 factor > 0.0 &\rightarrow factor \neq 0.0 \\
 radius == abs(factor) * old(radius) &\rightarrow getSizeX() == factor * old(getSizeX())
 \end{aligned}$$

Abstract data types (ADT)

The verification of a ADT (or other class) needs three definitions:

Representation (*footprint/dynamic frame*) set of objects of the *heap* associated to each instance and can be modified/created: set that includes also the instance and field objects (if any)

```
ghost var Repr: set<object>
```

Model ADT Specification for verification (functional)

Invariant identify the valid instances of the model: `Valid()`

```
ghost predicate Valid()
reads this, Repr
```

even if `this` in `Repr` that only happens if `Valid()` returns `true`, thus `reads` has to include `this`.

Abstraction/ Model

- Contracts must be abstract and do not reflect the internal implementation
- For instance data structures as sets or priority queues can be represented using arrays, hash, binary trees, etc.
- but contracts should not reflect those structures
- Abstract state representations can be,
 - **ghost** variables that internally are associated with the concrete implementation
 - **ghost** functions, that allow a abstract view of the internal state

Linked lists

```
type A = int
class Node<A> {
    ghost var Repr: set<Node<A>>;
    ghost var Model: seq <A>
    var data: A;
    var next: Node?<A>;
    ghost predicate Valid()
        reads this , Repr
    {
        this in Repr && (if next == null then
            Repr == {this} && model == [data]
            else next in Repr && Repr == {this} + next.Repr
            && Model == [data] + next.Model
            && this !in next.Repr
            && next.Valid())
    }
}
```

Linked lists

```
constructor(x: A, next: Node?<A>)
    requires next != null ==> next.Valid()
    ensures Valid()
    ensures Model == [x] + (if next == null then []
        else next.Model)
    ensures fresh(Repr - (if next == null then {}
        else next.Repr))
{ this.data := x;
    this.next := next;
    if next == null {
```

```

    this.Repr := { this };
    this.Model := [data];
} else {
    this.Repr := {this, next} + next.Repr;
    this.Model := [data] + next.Model; } }
}

```

More about *framing*: fresh

- `fresh(0)` states that objects 0 are newly allocated (in the *heap*) by the instance or method
- state in `Valid()` the components of `Repr`
- In the `constructor` state

```
ensures Valid() && fresh(Repr - {this})
```

- in the methods that modify mutable

```
requires Valid()
modifies Repr
ensures Valid() && fresh(Repr - {old(Repr)})
```

- These can be omitted using `:autocontracts` but only for simple cases.

Summary on Dynamic Frames

Representation Set ghost var `Repr`: set<object>

Invariant ghost predicate `Valid()`

```

    reads this, Repr
    ensures Valid() ==> this in Repr
    { this in Repr && a.Valid() && a in Repr && b.Valid() && b.Repr <= Repr
      && this !in b.Repr  &&
    //
    }

```

for each **a** that is a field object with a simple frame and each **b** field object with a dynamic frame. Other disjointness conditions may apply

Constructor constructor ()

```

    ensures Valid() && fresh(Repr)
    {
    //
    new;
    Repr := {this,a,b} + b.Repr
    //
    }

```

where **b** is a constituent object with a dynamic frame.

```
Functions function F(x: X): Y
    requires Valid()
    reads Repr
```

Methods A mutating method has Repr as its write frame

```
method M(x: X) returns (y: Y)
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
```

Summary of techniques to deal with heap updates

- Explicit representation sets are programmer-defined. Examples include dynamic frames and the Dafny tool.
- Implicit representation sets are derived from predicates in specialized logics. Examples include separation logics and the VeriFast and Viper tools.
- Predefined representation sets are derived from predefined heap topologies (commonly, a tree topology). Examples include the ownership relations in Spec#, VCC and regions in whyML.

Set specification

```
type T = int
class {:autocontracts} Set
{ ghost var elems: set<T>;
  constructor ()
    ensures elems == {}
  function size(): nat
    ensures size() == |elems|
  method insert(x : T)
    requires x !in elems
    ensures elems == old(elems) + {x}
  method delete(x: T)
    requires x in elems
    ensures elems == old(elems) - {x}
  function contains(x: T) : bool
    ensures contains(x) <==> x in elems
}
```

Set implementation

```
static const initialCapacity := 10;
var list: array<T>;
```

```

var used : nat;
ghost predicate Valid()
{
    used <= list.Length && list.Length >= initialCapacity
    && (forall i, j :: 0 <= i < j < used ==> list[i] != list[j])
    && |elems| == used && elems = (set x | x in multiset(list[..used]))
}
constructor ()
    ensures elems == {}
{
    list := new T[initialCapacity];
    used := 0;
    elems := {};
}

function size(): nat
    ensures size() == |elems|
{ used }

method insert(x : T)
    requires x !in elems & used < list.Length
    ensures elems == old(elems) + {x}
{
    list[used] := x;
    used := used + 1;
    elems := elems + {x};
}

```

Set implementation

```

method delete(x: T)
    requires x in elems
    ensures elems == old(elems) - {x}
{
    var i :| 0 <= i < used && list[i] == x;
    list[i] := list[used-1];
    used := used-1;
    elems := elems - {x};
}
function contains(x: T) : bool
    ensures contains(x) <==> x in elems
{
    exists i :: 0 <= i < used && list[i] == x
}

```

DEQUE: Double queue

- insert an element both from the front (`front`) and from the back of the queue (`back`)
- insertion: `push_back(x)`, `push_front(x)`
- delete: `pop_back(x)`, `pop_front(x)`
- inspection: `back(x)`, `front(x)`
- tests: `isEmpty()`, `isFull()`

Contracts

- we add `ghost` fields to describe the Deque in an abstract way:

```
ghost var elems: seq<T>; // front at head,  
// back at tail  
ghost const capacity: nat;
```

- Class invariant (predicate `Valid()`) ensures:
 - the consistence of the concrete variable
 - consistence of the abstract constraints with respect the abstract ones
 - defines the representation `Repr`

Specification

```
ghost var elems: seq<T>; // front at head, back at tail  
ghost const capacity: nat;  
ghost var Repr: set<object>  
ghost predicate Valid()  
    reads this, Repr  
    ensures Valid() ==> this in Repr  
constructor (capacity: nat)  
    ensures Valid() && fresh(Repr)  
    ensures elems == [] && this.capacity == capacity  
predicate isEmpty()  
    requires Valid()  
    reads Repr  
    ensures isEmpty() <==> elems == []  
predicate isFull()  
    requires Valid()  
    reads Repr  
    ensures isFull() <==> |elems| == capacity
```

```

function front() : T
    requires Valid() && !isEmpty()
    reads Repr
    ensures front() == elems[0]
method push_front(x : T)
    requires Valid() && !isFull()
    modifies Repr
    ensures Valid() && fresh (Repr - old(Repr))
    ensures elems == [x] + old(elems)
method pop_front()
    requires Valid() && !isEmpty()
    modifies Repr
    ensures Valid() && fresh (Repr - old(Repr))
    ensures elems == old(elems[1...])

function back() : T
    requires Valid() && !isEmpty()
    reads Repr
    ensures back() == elems[|elems| - 1]
method push_back(x : T)
    requires Valid() && !isFull()
    modifies Repr
    ensures Valid() && fresh (Repr - old(Repr))
    ensures elems == old(elems) + [x]
method pop_back()
    requires Valid() && !isEmpty()
    modifies Repr
    ensures Valid() && fresh (Repr - old(Repr))
    ensures elems == old(elems[..|elems|-1])

```

Test specification

```

var q := new Deque(3);
assert q.isEmpty();
q.push_front(1);
assert q.front() == 1;
assert q.back() == 1;
q.push_front(2);
assert q.front() == 2;
assert q.back() == 1;
q.push_back(3);
assert q.front() == 2;
assert q.back() == 3;
assert q.isFull();
q.pop_back();

```

```

assert q.front() == 2;
assert q.back() == 1;
q.pop_front();
assert q.front() == 1;
assert q.back() == 1;
q.pop_front();
assert q.isEmpty();

```

Implementation

The concrete variables are:

- Circular list `list` implemented with an array. one pointer `start` and the queue size `size` such that:
- `front:list[start]`
- `back: list[(start+size-1) mod list.Length]`

Valid()

```

class Deque {
    const list: array<T>;
    var start : nat;
    var size : nat;

    ghost predicate Valid()
        reads this, Repr
        ensures Valid() ==> this in Repr
        { this in Repr && list in Repr &&
          0<= size <=list.Length && 0<=start < list.Length
          && capacity ==list.Length
          && elems==if start+size <=list.Length then list[start..start+size]
          else
              list[start..] + list[..size-(list.Length-start)]
          {}}

```

Valid() with :autocontracts

```

class {:autocontracts} Deque {
    const list: array<T>;
    var start : nat;
    var size : nat;
    ghost predicate Valid()
    { 0 <= size <= list.Length && 0 <= start < list.Length
    && capacity == list.Length

```

```

    && elems == if start + size <= list.Length
    then list[start..start + size]
    else list[start..] + list[..size-(list.Length-start)] }
}

```

constructor()

```

constructor (capacity: nat)
  requires capacity > 0
  ensures Valid() && fresh(Repr)
  ensures elems == [] && this.capacity == capacity
{
  list := new T[capacity];
  start := 0;
  size := 0;
  this.capacity := capacity;
  elems := [];
  Repr := {this, list};
}

```

Repr will not change so `fresh()` can be omitted and we could define it with `const.`

constructor() with :autocontracts

```

constructor (capacity: nat)
  requires capacity > 0
  ensures elems == [] && this.capacity == capacity
{
  list := new T[capacity];
  start := 0;
  size := 0;
  this.capacity := capacity;
  elems := [];
}

```

All following methods aill be considered with `:autocontracts`

Tests

```

predicate isEmpty()
  ensures isEmpty() <==> elems == []
{
  size == 0
}

```

```

}

predicate  isFull()
    ensures isFull() <==> |elems| == capacity
{
    size == list.Length
}

back

function back() : T
    requires !isEmpty()
    ensures back() == elems[|elems| - 1]
{
    list[(start + size - 1) % list.Length]
}

method push_back(x : T)
    requires !isFull()
    ensures elems == old(elems) + [x]
{
    list[(start + size) % list.Length] := x;
    size := size + 1;
    elems := elems + [x];
}

back

method pop_back()
    requires !isEmpty()
    ensures elems == old(elems[..|elems|-1])
{
    size := size - 1;
    elems := elems[..|elems|-1];
}

front

function front() : T
    requires !isEmpty()
    ensures front() == elems[0]
{
    list[start]
}

```

front

```
method push_front(x : T)
    requires !isFull()
    ensures elems == [x] + old(elems)
{
    if start > 0
    {
        start := start - 1;
    }
    else
    {
        start := list.Length - 1;
    }
    list[start] := x;
    size := size + 1;
    elems := [x] + elems;
}
```

front

```
method pop_front()
    requires !isEmpty()
    ensures elems == old(elems[1..])
{
    if start + 1 < list.Length
    {
        start := start + 1;
    }
    else
    {
        start := 0;
    }
    size := size - 1;
    elems := elems[1..];
}
}
```