Construction and Verification of Software

2018 - 2019

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 5 - State Change and Type States

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Changing state & Framing

Set ADT (growable)

```
 class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    method Grow() returns (na:array<int>)
        requires RepInv()
        ensures size < na.Length
        ensures fresh(na)
        ensures forall k::(0<=k<size) ==> na[k] == a[k];
    {
        na := new int[a.Length*2];
        var i := 0;
        while (i<size)
            decreases size-i
            invariant 0 <= i <= size ;
            invariant forall k::(0<=k<i) ==> na[k] == a[k];
        {
            na[i] := a[i];
            i := i + 1;
        }
    }
```

Set ADT (growable)

```
 class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    method add(x:int)
        modifies this,a;
        requires RepInv()
        ensures RepInv()
    {
        var i := find(x);
        if (i < 0) {
            if (size == a.Length)
            { a := Grow(); }
            a[size] := x;
            size := size + 1;
      }
    }
```

Set ADT (growable)

```
class ASet {
    method del(x:int)
        modifies this,a;
        requires RepInv()
        ensures RepInv()
    {
        var i:int := find(x);
        if (i >= 0) {
            assert a[i] == x && forall j::(0<=j<i) ==> a[j] == old(a[j]); // <<<<<<
            var pos:int := i;
            while (i < size-1)
                modifies a;
                decreases size - 1 - i
                invariant …
            {
                a[i] := a[i+1];
                i := i + 1;
            }
            size := size - 1;
        }
    }

}
```

Further hints on invariants

- We illustrate a famous issue related to using formal logic to reason about dynamical systems, the so-called "**frame-problem**".
- There is no "purely logical" way of inferring what does not change after an action, we need in each case to specify for each action not only what changes, but also what has not (remains stable).
- E.g. this arises in reasoning about programs

```
 {x.val() == a && y.val() == 0 }

     x.inc() { x.val() == a+1 && y.val() == ?}
```

- How do we know changing x affects y or not?

Some hints on invariants

- Historically, the “frame problem” appeared while using logic to reason about robot actions.

- Consider the “action” axioms:

  Paint(t, x, c) ==> Colour(t+1, x, c)

  Move(x, p) ==> Pos(t+1, x, p)

- We would like the following implication to hold

  Colour(2, cube, red) && Pos(2,cube,6) && Paint(2, cube, blue)

  ==> Colour(3, cube, blue) && Pos(3,cube,6)

Some hints on invariants

- Unfortunately, there is no way to derive

  Colour(2, cube, red) && Pos(2,cube,6) && Paint(2, cube, blue)

  ==> Colour(3, cube, blue) **&& Pos(3,cube,6)**

- Painting has nothing to do with moving, and we would like to avoid having to explicitly say that.

- But "inertial" assumptions are always domain specific, and usually one needs to add "frame axioms" to assert what doesn't change.

- While Hoare Logic does not need frame axioms, since there is no aliasing or invisible side effects.

Set ADT (growable)

```
class ASet {
    method del(x:int)
        modifies this,a;
        requires RepInv()
        ensures RepInv()
    {
      var i:int := find(x);
      if (i >= 0) {
          var pos:int := i;
          while (i < size-1)
            modifies a;
            decreases size - 1 - i
            invariant pos <= i <= size-1
            invariant unique(a,0,i) && unique(a,i+1,size)
            invariant forall j::(0 <= j < pos) ==> a[j] == old(a[j])
            invariant forall j::(pos <= j < i) ==> a[j] == old(a[j+1])
            invariant forall j::(i+1 <= j < size) ==> a[j] == old(a[j])
          {
              a[i] := a[i+1];
              i := i + 1;
          }
          size := size - 1;
      }
```

Some hints on invariants

- In the previous code, we consider the invariants

```
invariant 0 <= pos <= i <= size-1;

invariant unique(a,0,i) && unique(a,i+1,size);

invariant forall j::(0<=j<pos) ==> a[j] == old(a[j]) ;

invariant forall j::(pos<=j<i) ==> a[j] == old(a[j+1]) ;

invariant forall j::(i+1<=j<size) ==> a[j] == old(a[j]);
```

- The method body assumes that all components of this and a can be modified as a side effect, so Dafny does not add any frame principles: we need to include the necessary ones in invariants.

Set ADT (growable)

```
class ASet {
…
    method Grow() returns (na:array<int>)
        requires RepInv()
        ensures size < na.Length
        ensures fresh(na)
        ensures forall k::(0<=k<size) ==> na[k] == a[k];
    {
        na := new int[a.Length*2];
        var i := 0;
        while (i<size)
            decreases size-i
            invariant 0 <= i <= size ;
            invariant forall k::(0<=k<i) ==> na[k] == a[k];
        {
            na[i] := a[i];
            i := i + 1;
        }
    }
}
```

Some hints on invariants

- In the previous code for grow, no frame conditions where added, since there is no **modifies** clause !
- In general, we need only add frame conditions for data that is declared to be subject to change by some **modifies** declaration.

Some hints on invariants

- Another way, much simpler, of expressing frame conditions, is to add **modifies** declarations directly in the loop, close to the invariants
- A **modifies** clause in a while loop overrides the method modifies clause, making it more precise.
- **Only** the instance variables mentioned in the loop modifies clause will be assumed to be changed by the loop.
- The other variables will be framed out, and Dafny will automatically know that they will not change.
- This will often save frame conditions.

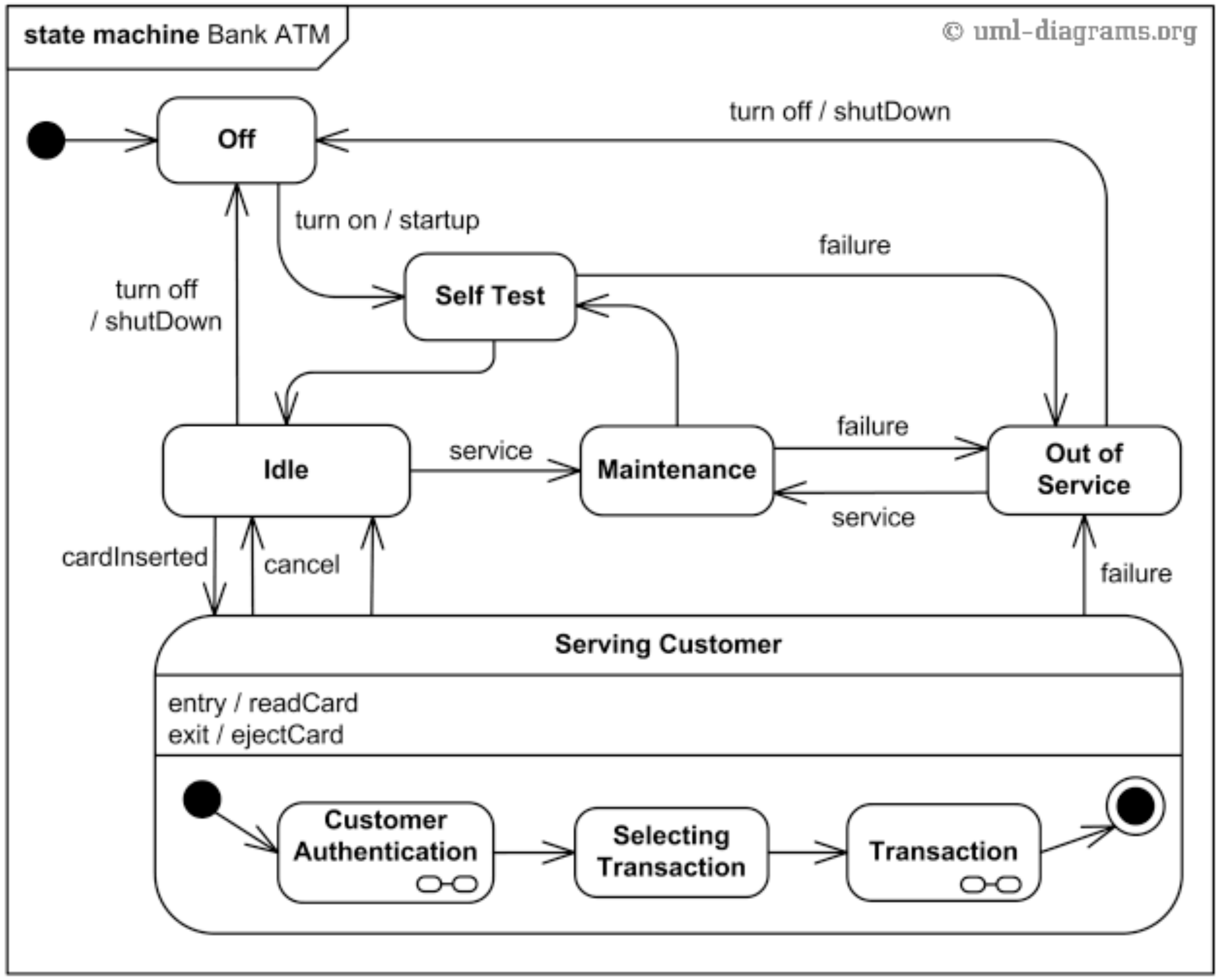
Key Points

- The ADT operations pre / post conditions must always preserve the representation invariant
- Other operations (private helper methods) do not need to preserve the invariant, they are need to know about the ADT implementation details
- The ADT pre / post conditions should avoid referring to the concrete state, to preserve information hiding
- To do that, you may expose ghost variables
- Alternatively, use also some form of **typestate**, enough to express rich dynamic constraints (next lecture)

ADT States

UML State Transition Diagrams

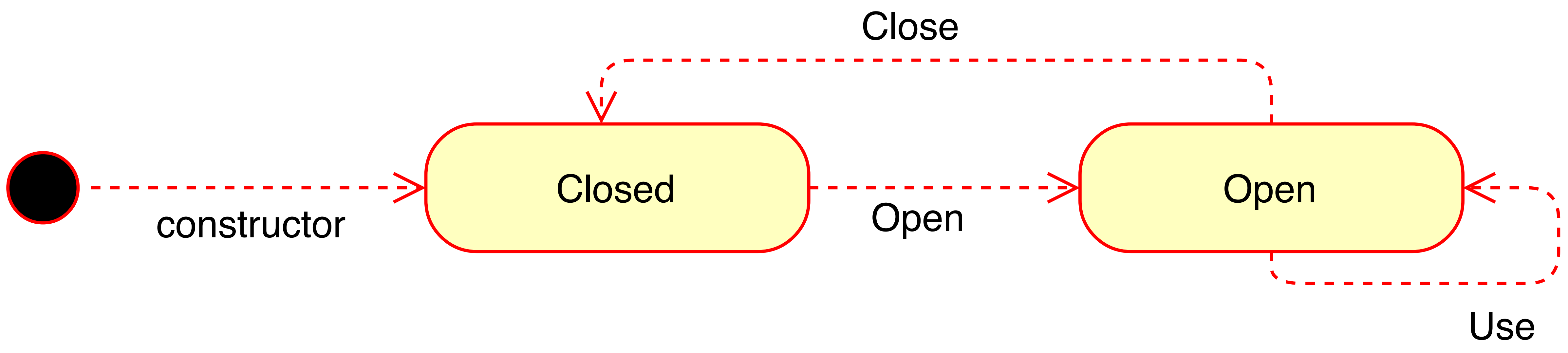
- Typically the connection between a state and the domain of the values for an object are based on conventions / written in documentations.
- Operations are state transitions in a state diagram.
- If a state is formally connected to conditions over the state of an object, the correction of state transitions may be mechanically checked



https://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html?context=stm-examples

TypeStates

- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- We illustrate using a general Resource object with the following state diagram



TypeStates

- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- We illustrate using a general Resource object.
    - A Resource must first be created and starts on the closed state
    - A Resource can only be used after being Opened
    - A Resource may be Closed at any time
    - A Resource can only be Opened if it is in the Closed state, and Closed if it is in the Open state
- We define two abstract states (`ClosedState()` and `OpenState()`)

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { … }

    function ClosedState():bool
    reads this
    { …  }

    constructor ()
    ensures  ClosedState();
    { … }

 …

}
```

TypeStates define an abstract layer, visible to clients that can be used to verify resource usage.

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { … }

    function Clos
    reads this
    { …  }

    constructor (
    ensures  Clos
    { … }

  …

}
```

```
method UsingTheResource()
{
    var r:Resource := new Resource();
    r.Open(2);
    r.Use(2);
    r.Use(9);
    r.Close();
}
```

Legal usage of resource, according to protocol!

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { … }

    function Clos
    reads this
    { …  }

    constructor (
    ensures  Clos
    { … }

  …

}
```

```
method UsingTheResource()
{
    var r:Resource := new Resource();
    r.Close();
    r.Open(2);
    r.Use(2);
    r.Use(9);
    r.Close();
    r.Use(2);
}
```

Illegal usage of resource, according to protocol!

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;

    function OpenState():bool
    reads this
    { h != null && 0 < size == h.Length }

    function ClosedState():bool
    reads this
    { h == null && 0 == size  }

    constructor ()
    ensures  ClosedState();
    { h := null; size := 0; }

  …

}
```

TypeStates define an abstract layer, that may be defined with relation to the representation type (and invariants) and be used to verify the implementation.

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;
 …
    method Open(N:int)
    modifies this
    requires ClosedState() && N > 0
    ensures  OpenState() && fresh(h)
    {
        h, size := new int[N], N;
    }

    method Close()
    modifies this
    requires OpenState()
    ensures  ClosedState()
    {
        h, size :=null, 0;
    }
```

Method Implementations represent state transitions, and must be implemented to correctly ensure the soundness of the arrival state (assuming the departure state)

Resource

```
class Resource {

    var h:array?<int>;
    var size:int;
 …

    method Use(K:int)
    modifies h;
    requires OpenState();
    ensures  OpenState();
    {
       h[0] := K;
    }

 …

}
```

No execution errors are caused by misusing the representation type. Notice that states are `RepInv()` variants, essential to execute different method.

TypeStates

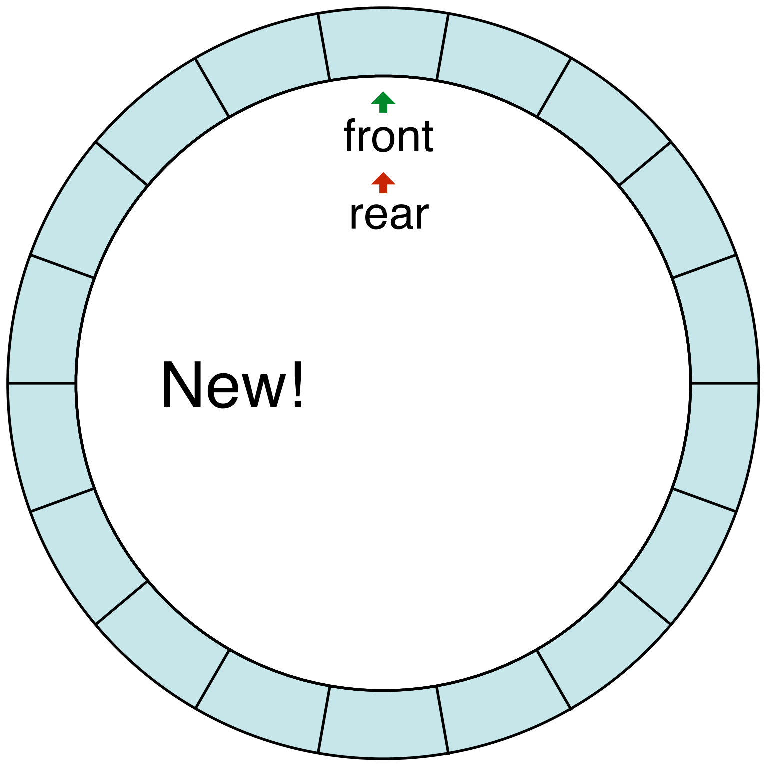
- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- It is often enough to expose TypeState assertions to ensure ADT soundness and no runtime errors
- In general, full functional specifications in terms the abstract state is too expensive and should be only adopted in high assurance code
- However, TypeState assertions are feasible and should be enforced in all ADTs:
- The simplest TypeState is the RepInv (no variants/less specific).

Key Points

- Software Design Time
  - Abstract Data Type
  - What are the Abstract States / Concrete States?
  - What is the Representation Invariant?
  - What is the Abstraction Mapping?
- Software Construction Time
  - Make sure constructor establishes the RepInv
  - Make sure all operations preserve the RepInv
    - they may assume the RepInv
    - they may require extra pre-conditions (e.g. on op args)
    - they may enforce extra post-conditions
  - Use assertions to make sure your ADT is sound

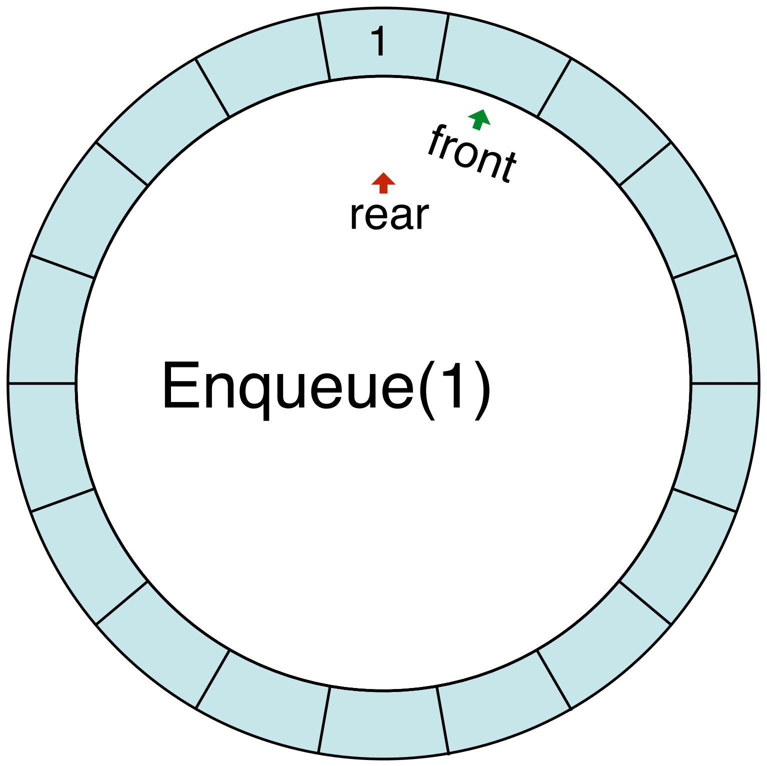
TypeStates - Queue

- An implementation using a circular buffer…



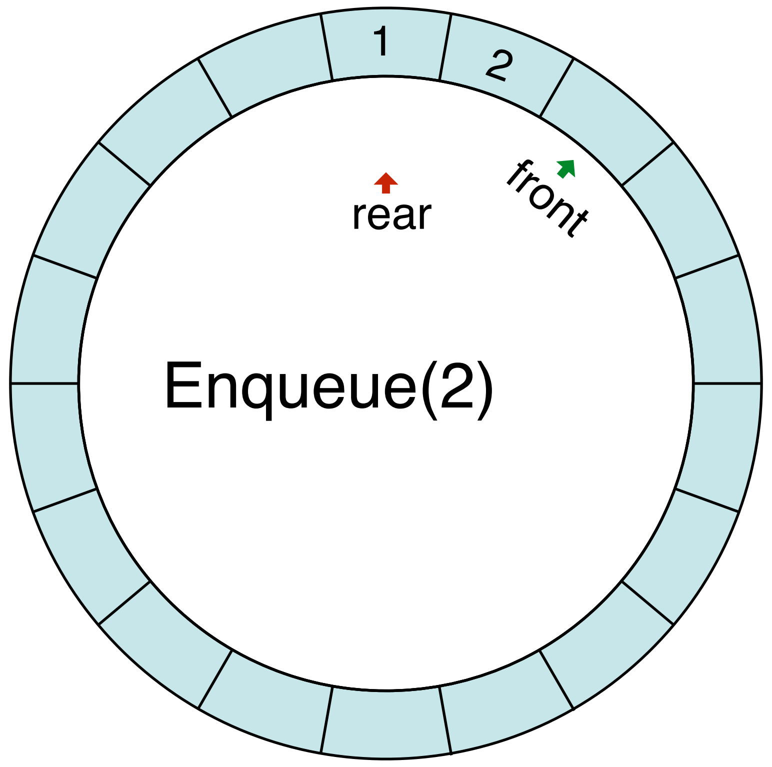
TypeStates - Queue

- An implementation using a circular buffer…



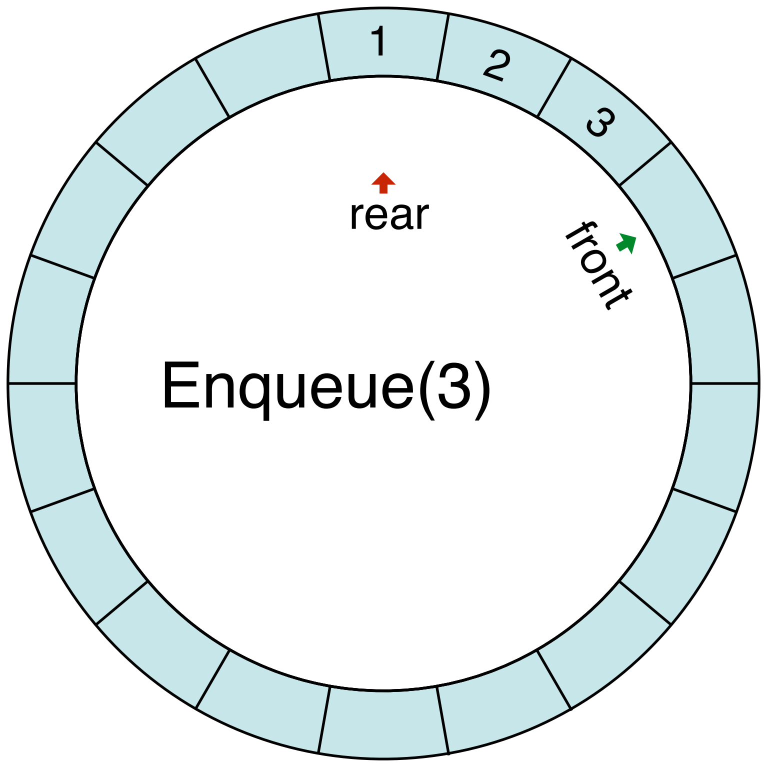
TypeStates - Queue

- An implementation using a circular buffer…



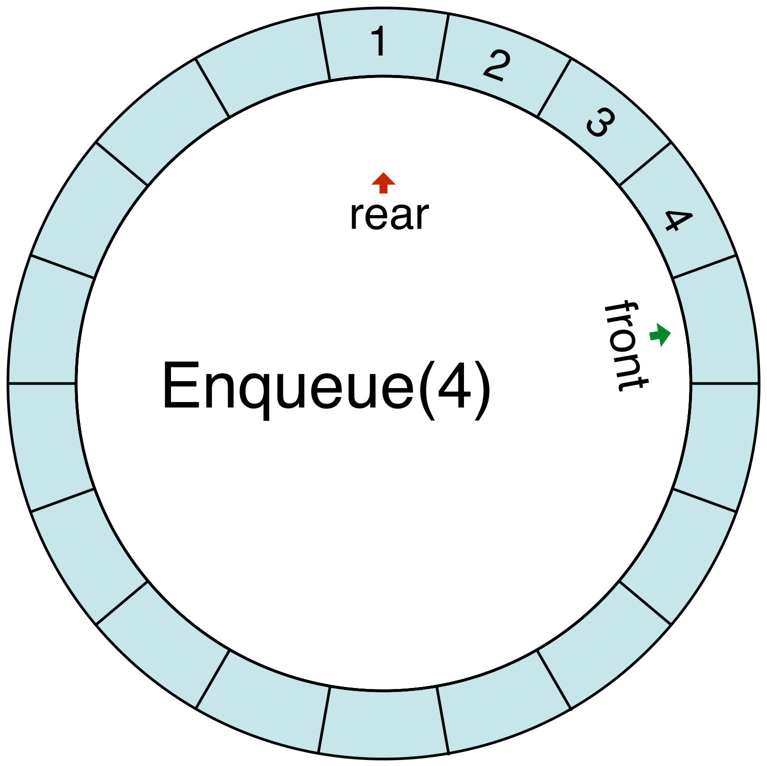
TypeStates - Queue

- An implementation using a circular buffer…



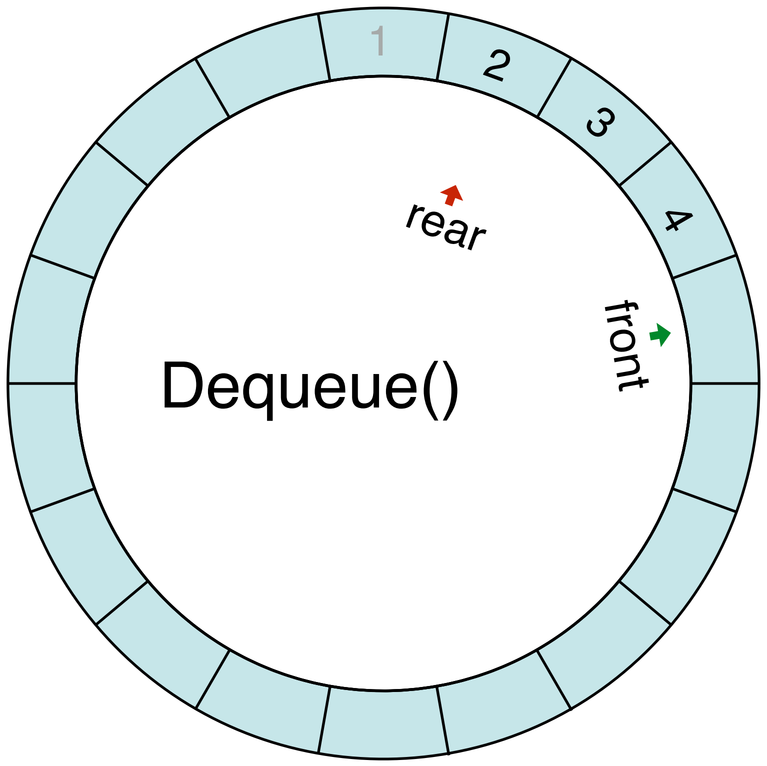
TypeStates - Queue

- An implementation using a circular buffer…



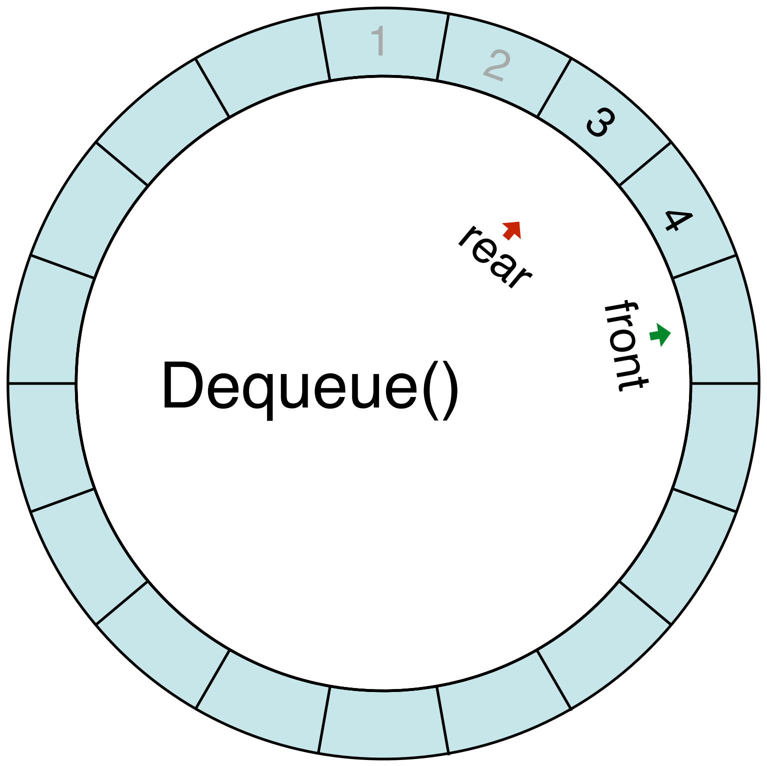
TypeStates - Queue

- An implementation using a circular buffer…



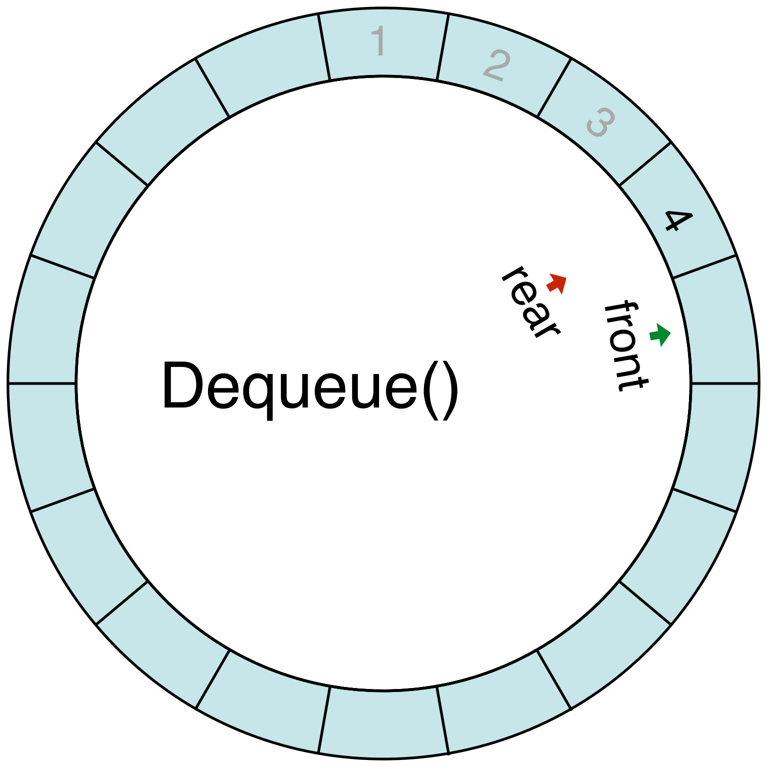
TypeStates - Queue

- An implementation using a circular buffer…



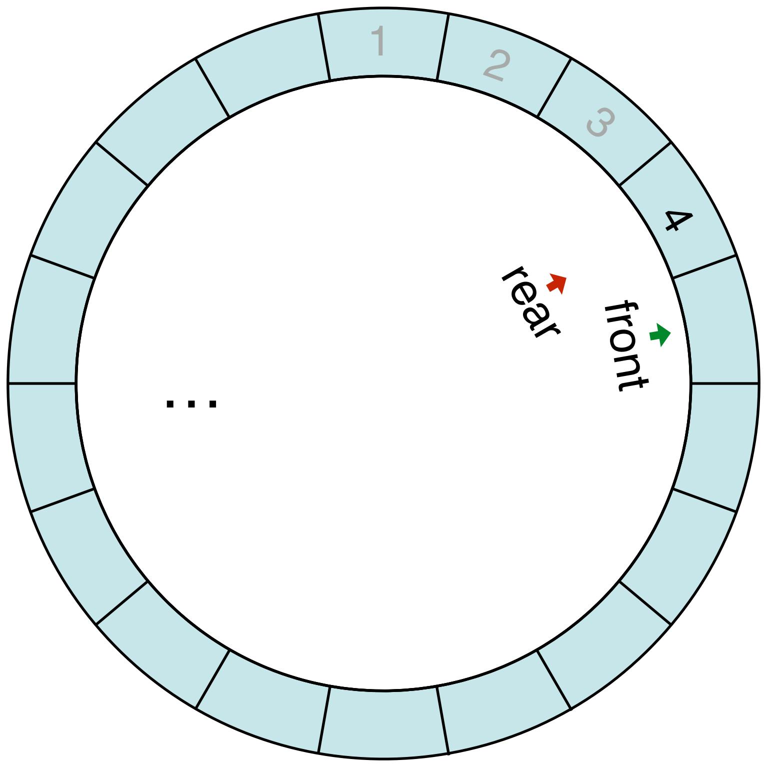
TypeStates - Queue

- An implementation using a circular buffer…



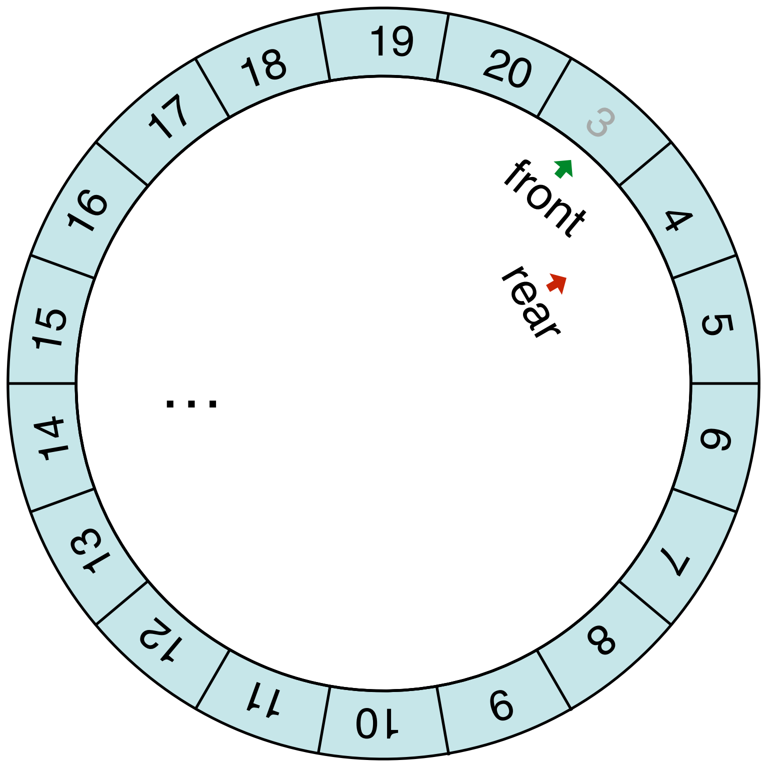
TypeStates - Queue

- An implementation using a circular buffer…



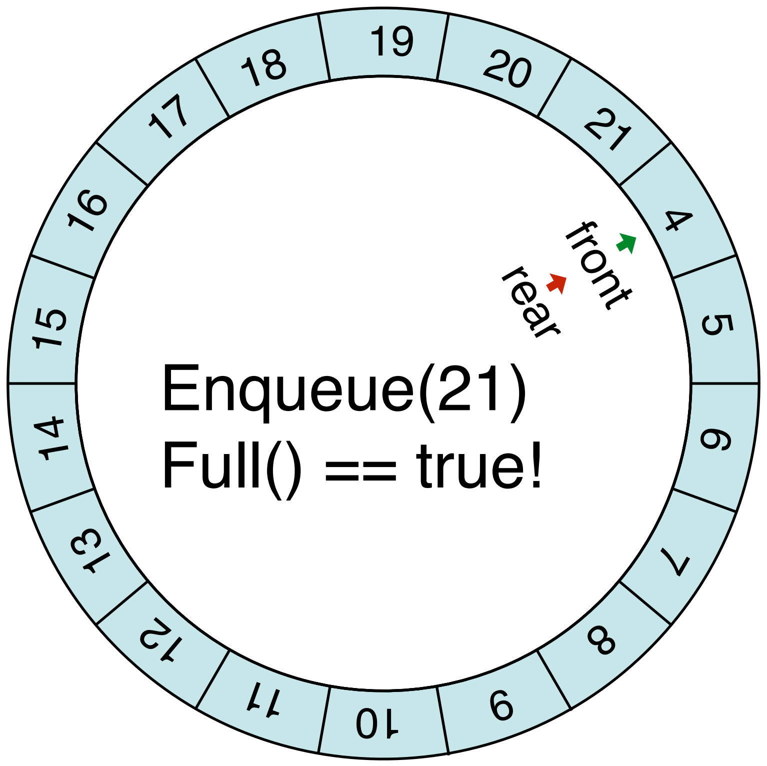
TypeStates - Queue

- An implementation using a circular buffer…



TypeStates - Queue

- An implementation using a circular buffer…



TypeStates - Queue

- An implementation using a circular buffer…

```
class Queue {
    // Representation type
    var a:array<int>;
    var front: int;
    var rear: int;
    var numberOfElements: int;

    // Representation invariant
    constructor(N:int)
        requires 0 < N
        ensures fresh(a)
    {
        a := new int[N];
        front := 0;
        rear := 0;
        numberOfElements := 0;
    }
    …
}
```

TypeStates - Queue

- What’s wrong with it? a RepInv is necessary to maintain front and rear within bounds…

```
class Queue {

    …
    method Enqueue(V:int)
        modifies this`front, this`numberOfElements, a
    {
        a[front] := V;
        front := (front + 1)%a.Length;
        numberOfElements := numberOfElements + 1;
    }

    method Dequeue() returns (V:int)
        modifies this`rear, this`numberOfElements, a
    {
        V := a[rear];
        rear := (rear + 1)%a.Length;
        numberOfElements := numberOfElements - 1;
    }
}
```

TypeStates - Queue

```
class Queue {
    // Representation type
    var a:array<int>;
    var front: int;
    var rear: int;
    var numberOfElements: int;

    // Representation invariant
    function RepInv():bool
        reads this
    { 0 <= front < a.Length && 0 <= rear < a.Length }

    constructor(N:int)
        requires 0 < N
        ensures RepInv()
        ensures fresh(a)
    {
        a := new int[N];
        front := 0;
        rear := 0;
        numberOfElements := 0;
    }
    …
}
```

TypeStates - Queue

```
class Queue {
    …
    method Enqueue(V:int)
        modifies this`front, this`numberOfElements, a
        requires RepInv()
        ensures RepInv()
    {
        a[front] := V;
        front := (front + 1)%a.Length;
        numberOfElements := numberOfElements + 1;
    }

    method Dequeue() returns (V:int)
        modifies this`rear, this`numberOfElements, a
        requires RepInv()
        ensures RepInv()
    {
        V := a[rear];
        rear := (rear + 1)%a.Length;
        numberOfElements := numberOfElements – 1;
    }
}
```

TypeStates - Queue

- Not enough… No runtime errors but the correct behaviour is not yet ensured…
  wrong values may be returned,
  valid elements maybe overwritten… right?

```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    q.Enqueue(1);
    r := q.Dequeue();   ????
    r := q.Dequeue();
    q.Enqueue(2);
    q.Enqueue(3);
    q.Enqueue(4);  ????
    q.Enqueue(4);
    q.Enqueue(4);
    q.Enqueue(5);
```

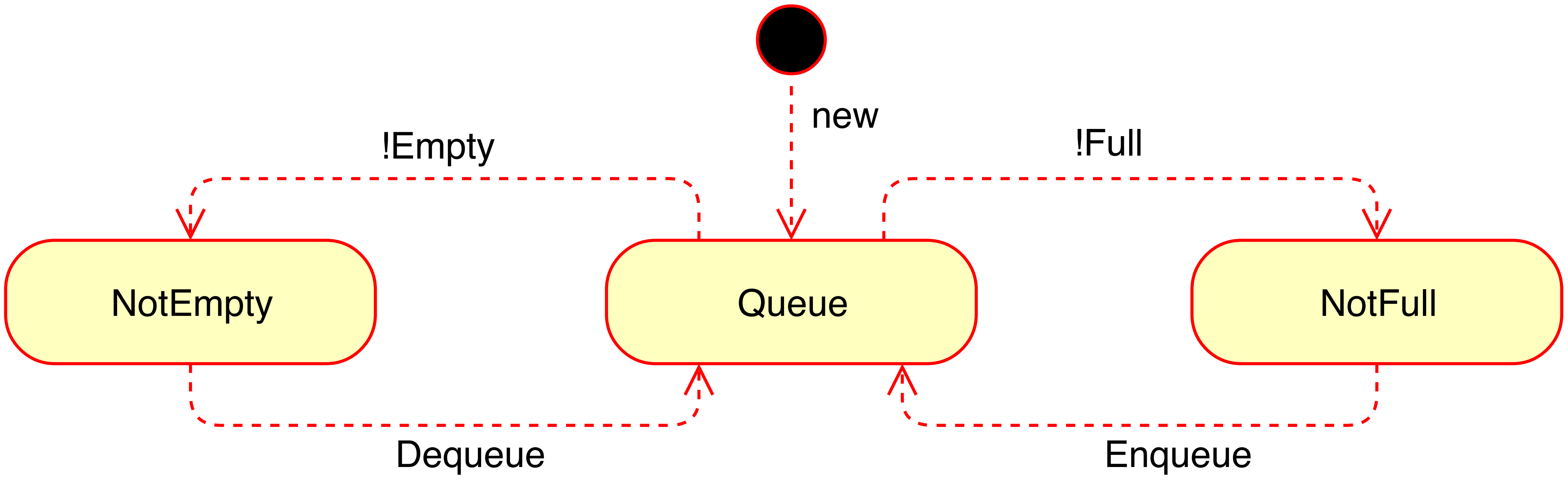
TypeStates - Queue

- RepInv must be refined to ensure that we stay inside the domain of valid queues…

```
function RepInv():bool
     reads this
 {
     0 <= front < a.Length &&
     0 <= rear < a.Length &&
     if front == rear then
       numberOfElements == 0 ||
       numberOfElements == a.Length
     else
       numberOfElements ==
         if front > rear
         then front - rear
         else front-rear+a.Length
 }
```

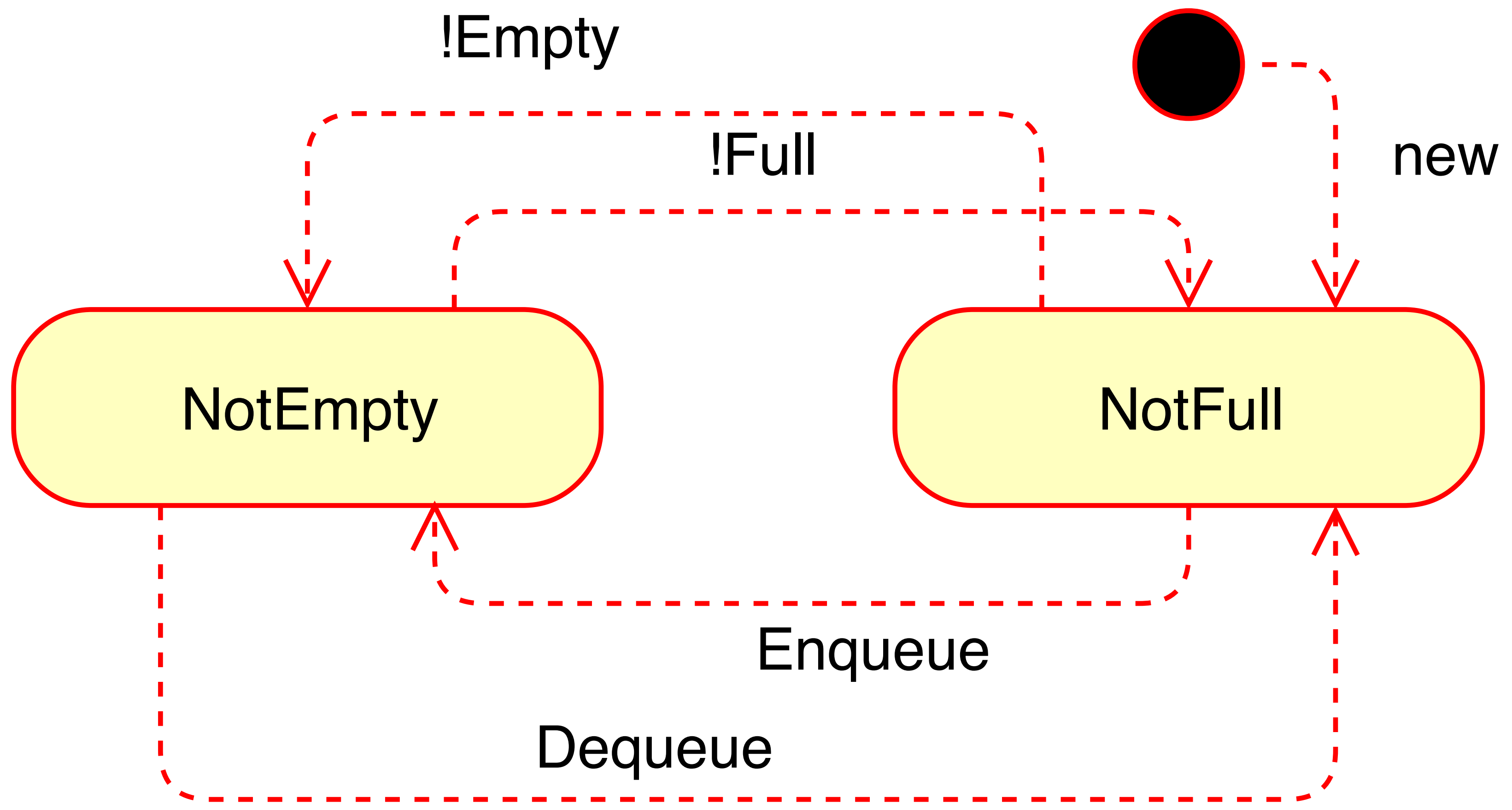
TypeStates - Queue

- Enqueue and Dequeue Operations are only valid in certain states… Obtained by dynamic testing operations.



TypeStates - Queue

- Enqueue and Dequeue Operations are only valid in certain states… Obtained by dynamic testing operations.



TypeStates - Queue

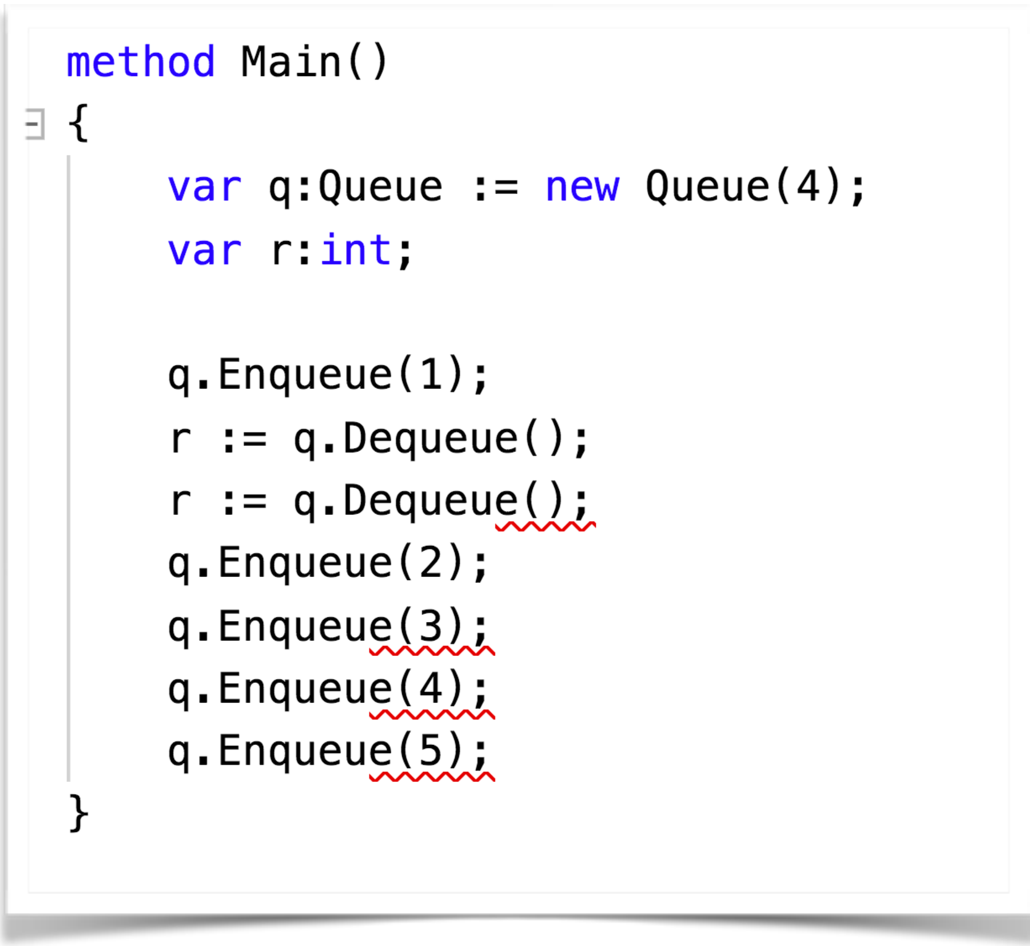
```
class Queue {
    …
    function NotFull():bool
        reads this
    { RepInv() && numberOfElements < a.Length }

    function NotEmpty():bool
        reads this
    { RepInv() && numberOfElements > 0 }

    constructor(N:int)
        requires 0 < N
        ensures NotFull()
        ensures fresh(a)
    { … }

    method Enqueue(V:int)
        modifies this`front, this`numberOfElements, a
        requires NotFull()
        ensures NotEmpty()
    { … }

    method Dequeue() returns (V:int)
        modifies this`rear, this`numberOfElements, a
        requires NotEmpty()
        ensures NotFull()
    { … }
```



```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    q.Enqueue(1);
    r := q.Dequeue();
    r := q.Dequeue();
    q.Enqueue(2);
    q.Enqueue(3);
    q.Enqueue(4);
    q.Enqueue(5);
}
```

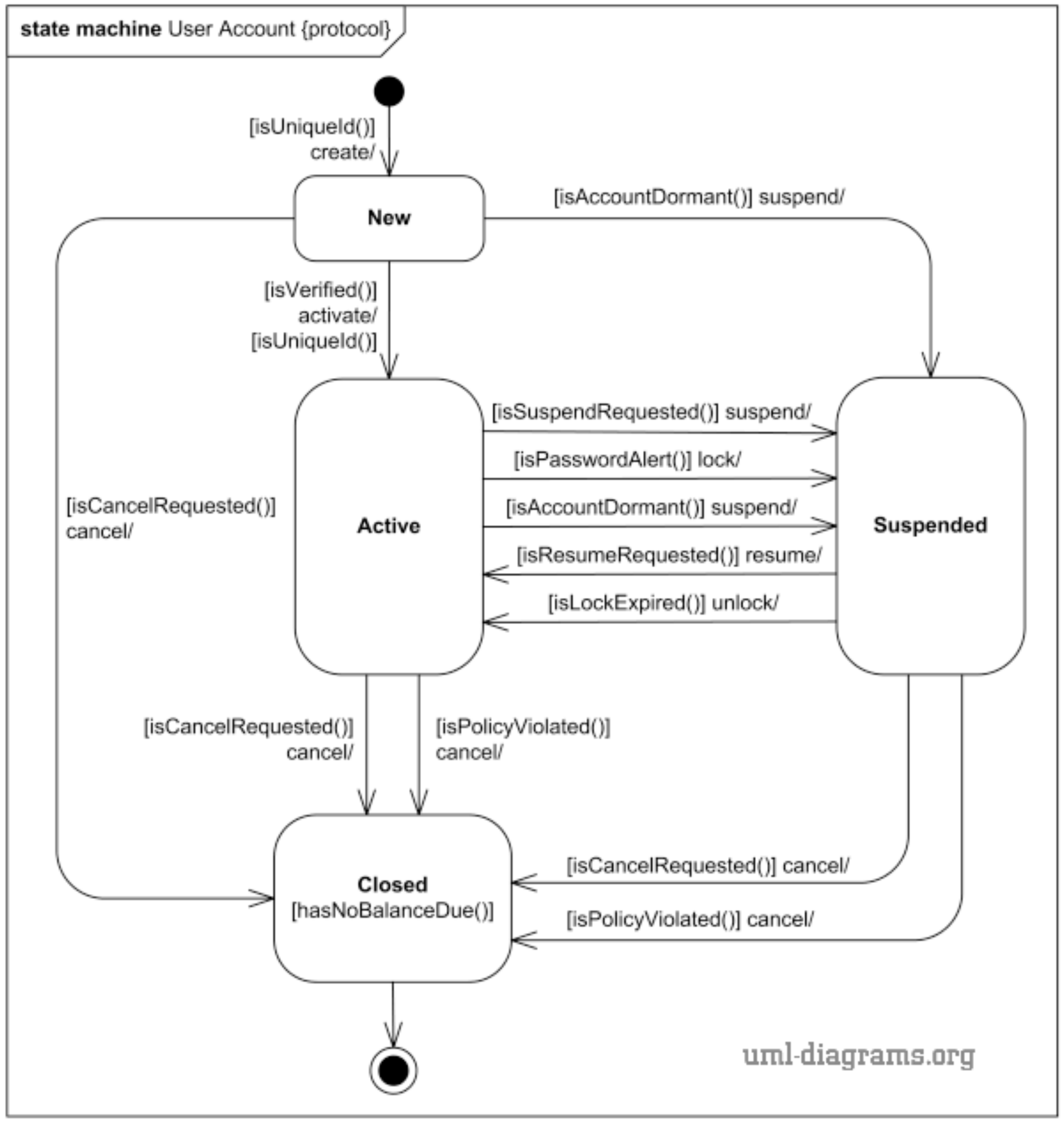
TypeStates - Queue

- Dynamic Tests ensure the proper state for a given operation…

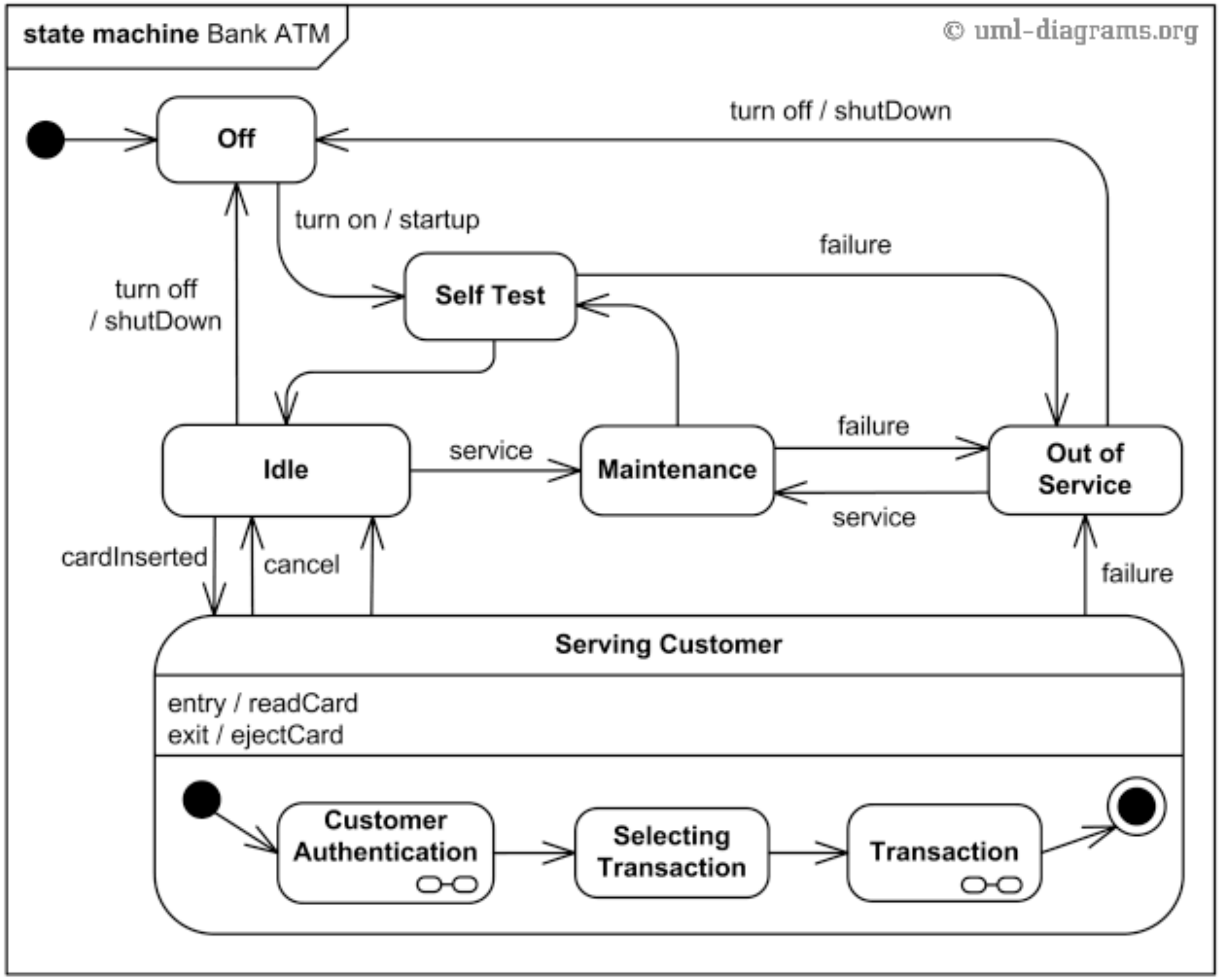
```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    q.Enqueue(1);
    r := q.Dequeue();
    if !q.Empty()
    { r := q.Dequeue(); q.Enqueue(2); }
    if !q.Full() { q.Enqueue(3); }
    if !q.Full() { q.Enqueue(4); r := q.Dequeue(); }
    if !q.Full() { q.Enqueue(5); }
}
```

TypeStates - UserAccount in a store


state machine User Account {protocol}
[isUniqueId()] create/
New
[isAccountDormant()] suspend/
[isVerified()] activate/ [isUniqueId()]
Active
[isSuspendRequested()] suspend/
[isPasswordAlert()] lock/
[isAccountDormant()] suspend/
[isResumeRequested()] resume/
[isLockExpired()] unlock/
Suspended
[isCancelRequested()] cancel/
[isCancelRequested()] cancel/
[isPolicyViolated()] cancel/
Closed [hasNoBalanceDue()]
[isCancelRequested()] cancel/
[isPolicyViolated()] cancel/
uml-diagrams.org

TypeStates - ATM



Further Reading

- **Program Development in Java**, ***Barbara Liskov and John Guttag***, Addison Wesley, 2003, Chapter 5 “Data Abstraction” (other book chapters are also interesting).
- **Programming with abstract data types**, ***Barbara Liskov and Stephen Zilles***, ACM SIGPLAN symposium on Very high level languages, 1974 (read the introductory parts, the rest is already outdated, but the intro is a brilliant motivation to the idea of ADTs). You can access this here: http://dl.acm.org/citation.cfm?id=807045.