

Construction and Verification of Software

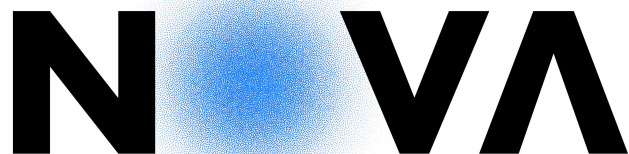
2022 - 2023

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 6 - Abstract Data Types

Bernardo Toninho (htoninho@fct.unl.pt)

based on previous editions by **João Seco** and **Luís Caires**



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Outline

- Framing in Hoare Logic
- Modifying an Array
- ADTs and TypeStates
- Building a Queue with TypeStates

Part I

Framing and Changing State

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.
- This is captured by the derived (constancy or frame) rule

$$\frac{\{A\} P \{B\}}{\{A \wedge C\} P \{B \wedge C\}} \text{ where } M(P) \cap V(C) = \emptyset$$

- **Crucial:** Provided that the variables modified by P (M(P)) are not referred by C (V(C)).

$$\frac{\{x > 0\} y := x \{y > 0 \wedge x = y\}}{\{x > 0 \wedge z < 0\} y := x \{y > 0 \wedge x = y \wedge z < 0\}}$$

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.
- This is captured by the derived (constancy or frame) rule

$$\frac{\{A\} P \{B\}}{\{A \wedge C\} P \{B \wedge C\}} \text{ where } M(P) \cap V(C) = \emptyset$$

- **Crucial:** Provided that the variables modified by P ($M(P)$) are not referred by C ($V(C)$).
- Updates to variables do not allow framing the modified variables.

`method deposit(v:int)`

`modifies this`bal` ← like one assignment

Changing State

- Tracking changes with dynamic memory is not covered by Hoare Logic. Each approach adopts some kind of strategy to make the frame rule sound:

`predicate Valid()`

`reads this`a, this`size, this`s, this.a`
`{ RepInv() && Sound() }`

memory referred by

`method add(x:int)`

`requires Valid()`

`ensures Valid()`

`modifies this`a, this.a, this`size, this`s`

field of

contents of

memory modified by

- Dafny refers to allocated memory areas. Objects and arrays. Modification of fields are modifications to the container object.

Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

$$\frac{\{A\} \ P \ \{B\}}{\{A \wedge C\} \ P \ \{B \wedge C\}}$$

- Information in the interface is important to know modified and referred memory.

```
method Main() {  
    var a:Account := new Account();  
    a.deposit(10);  
    a.withdraw(20); <<<< ????  
    if a.getBalance() > 10  
    { a.withdraw(10); a.deposit(10); }  
}
```

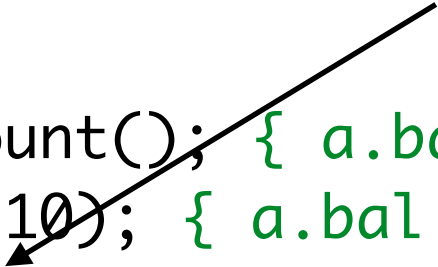
Changing State

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

$$\frac{\{A\} P \{B\}}{\{A \wedge C\} P \{B \wedge C\}}$$

- Information in the interface is important to know modified and referred memory.

```
requires 0 <= v <= getBal()  
  
method Main() {  
  var a:Account := new Account(); { a.bal >= 0 }  
  { a.bal >= 0 } a.deposit(10); { a.bal >= 0 }  
  { a.bal >= 0 } a.withdraw(20);  
  if a.getBalance() > 10  
  { { a.bal > 10 } a.withdraw(10); a.deposit(10); }  
}
```



Part II


Modifying an Array

Modifying an array

- Tracking state changes and invalidating knowledge along the proof is crucial in the verification of imperative programs.

```
method selectionSort(a:array<char>, n:int)
  requires 0 <= n <= a.Length
  modifies a
{
  var i := 0;
  while i < n
    decreases n - i
    invariant 0 <= i <= n
    invariant sorted(a, i)
    invariant partitioned(a, i, n)
  {
    selectSmaller(a, i, n);
    i := i + 1;
  }
}
```

```
method selectSmaller(a:array<char>, i
  requires 0 <= i < n <= a.Length
  requires sorted(a, i)
  requires partitioned(a, i, n)
  modifies a
  ensures sorted(a, i+1)
  ensures partitioned(a, i+1, n)
```



- It is like an assignment to all positions in the array

Modifying an array

- To have control about the positions that are indeed modified extra information is needed.

```
method sortedInsertion(a:array<char>, na:int, e:char)
returns (z:array<char>, nz:int, pos:int)
  requires 0 <= na < a.Length
  requires sorted(a, na)
  modifies this.a
  ensures sorted(a, na+1)
  ensures 0 <= pos < na+1 && a[pos] == e
  ensures forall k :: 0 <= k < pos ==> a[k] == old(a[k])
  ensures forall k :: pos < k < na+1 ==> a[k] == old(a[k-1])
```

- This allows the caller context to maintain (frame) knowledge about the unmodified positions.
- All knowledge about the array must be given by the post conditions **explicitly**.

Another example

- The Set example

```
method add(x:int)
requires Valid() && size() < maxSize()
ensures Valid()
ensures s == old(s) + {x}
modifies this`s, this`size, this.a
{
    var idx := find(x);
    if( idx < 0 ) {
        a[size] := x;
        size := size + 1;
        s := s + {x};
    }
}
```

- The precondition about the maximum size is not nice...

Another example

- A more general implementation is needed to eliminate it

```
method add(x:int)
requires Valid()
ensures Valid()
ensures s == old(s) + {x}
modifies this`s, this`size, this`a, this.a
{
    if( size == a.Length ) { a := Grow(); }

    var idx := find(x);
    if( idx < 0 ) {
        a[size] := x;
        size := size + 1;
        s := s + {x};
        assert forall i :: 0 <= i < size-1 ==> a[i] == old(a[i]);
    }
}
```

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 Valid()  
    ensures size < na.Length  
    ensures fresh(na) ← memory not tracked before  
    ensures forall k::( $0 \leq k < \text{size}$ ) ==> na[k] == a[k];  
  {  
    na := new int[a.Length*2];  
    var i := 0;  
    while (i < size)  
      decreases size-i  
      invariant  $0 \leq i \leq \text{size}$  ;  
      invariant forall k::( $0 \leq k < i$ ) ==> 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)  
    requires Valid()  
    ensures Valid()  
    ensures s == old(s) + {x}  
    modifies this`s, this`size, this`a, this.a  
    {  
      if( size == a.Length ) { a := Grow(); }  
  
      var idx := find(x);  
      if( idx < 0 ) {  
        a[size] := x;  
        size := size + 1;  
        s := s + {x};  
        assert forall i :: 0 <= i < size-1 ==> a[i] == old(a[i]);  
      }  
    }  
}
```

Set ADT (growable)

```
class ASet {
  method del(x:int)
    modifies this, a;
    requires Valid()
    ensures Valid()
    ensures x in old(s) <==> old(s) == s + {x}
{
  var i:int := find(x);
  if i >= 0 {
    var pos := 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;
  }
}
```


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?

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).

Part III

Framing

ADTs and Framing

- The ADT operations pre / post conditions must always preserve the representation invariant
- The ADT pre / post conditions should not refer to the concrete state, to preserve information hiding.
- To do that, you may expose ghost variables.
- ...but what about **reads** / **modifies** clauses?

ADTs and Framing — Counter

```
class Counter
{
    var incs: int;
    var decs: int;

    ghost var Value : int;

    predicate Valid()
        reads this
    {
        Value == incs - decs
    }

    constructor ()
        ensures Valid()
        ensures Value == 0
    {
        incs, decs, Value := 0, 0, 0;
    }
}
```

ADTs and Framing — Counter

```
class Counter
{
    ...
    method getValue() returns (x: int)
        requires Valid()
        ensures x == Value
    {
        x := incs - decs;
    }

    method inc()
        modifies this
        requires Valid()
        ensures Valid()
        ensures Value == old(Value) + 1
    {
        incs, Value := incs + 1, Value + 1;
    }
}
```

ADTs and Framing — Counter

```
class Counter
{ ...
  method dec()
    modifies this
    requires Valid()
    ensures Valid()
    ensures Value == old(Value) - 1
  {
    decs, Value := decs + 1, Value - 1;
  }
  method Main()
  {
    var c := new Counter();
    c.inc(); c.inc();
    c.dec(); c.inc();
    var x := c.getValue();
    assert x == 2;
    assert c.Valid();
  }
```

ADTs and Framing

- For the Counter class, the **reads/modifies** clauses are abstract (no leaking of internal representation).
- Clients can use the Counter objects in the expected way.
- Even though Counters are dynamically allocated, they use no dynamically allocated memory!
- Lets see what happens in that case...

ADTs and Framing — Counter with Cells

```
class Cell
{
    var data: int;

    constructor (n: int)
        ensures data == n
    { data := n; }
}

class CounterCell
{
    var incs: Cell;
    var decs: Cell;
    ghost var Value : int;

    predicate Valid()
        reads this, incs, decs
    {
        incs != decs && Value == incs.data - decs.data
    }
}
```

ADTs and Framing — Counter with Cells

```
class Cell
{
    var data: int;

    constructor (n: int)
        ensures data == n
    { data := n; }
}
```

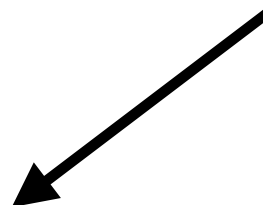
```
class CounterCell
{
```

```
    var incs: Cell;
    var decs: Cell;
    ghost var Value : int;
```

```
    predicate Valid()
        reads this, incs, decs
```

```
    {
        incs != decs && Value == incs.data - decs.data
    }
```

API leaks the representation!



ADTs and Framing — Counter with Cells

```
class CounterCell  
{
```

```
  constructor ()
```

```
    ensures Valid()
```

```
    ensures Value == 0
```

```
    ensures fresh(incs)
```

```
    ensures fresh(decs)
```

```
{
```

```
  incs := new Cell(0);
```

```
  decs := new Cell(0);
```

```
  Value := 0;
```

```
}
```

```
method inc()
```

```
  requires Valid()
```

```
  modifies this
```

```
  ensures Valid() && Value == old(Value) + 1
```

```
  modifies incs
```

```
{
```

```
  incs.data, Value := incs.data + 1, Value + 1;
```

```
}
```

API leaks the representation!



ADTs and Framing — Counter with Cells

```
class CounterCell
{
  method Main()
  {
    var c := new CounterCell();
    c.inc();
    c.inc(); //Error: Call may violate context's modifies clause
    c.dec(); //Error: Call may violate context's modifies clause
    c.inc(); //Error: Call may violate context's modifies clause
    var x := c.getValue();
    assert x == 2;
    assert c.Valid();
  }
}
```

- The issue is that after the first inc(), we lose too much information...

ADTs and Framing — Counter with Cells

```
class CounterCell
```

```
{
```

```
  constructor ()
```

```
    ensures Valid()
```

```
    ensures Value == 0
```

```
    ensures fresh(incs)
```

```
    ensures fresh(decs)
```

```
{
```

```
  incs := new Cell(0);
```

```
  decs := new Cell(0);
```

```
  Value := 0;
```

```
}
```

```
method inc()
```

```
  requires Valid()
```

```
  modifies this
```

```
  ensures Valid() && Value == old(Value) + 1
```

```
  modifies incs
```

```
  ensures incs == old(incs) && decs == old(decs)
```

```
{
```

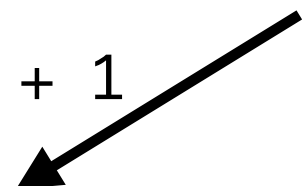
```
  incs.data, Value := incs.data + 1, Value + 1;
```

```
}
```

API leaks the representation!



Effective usage requires
even more leaking



ADTs and Framing

- It seems that all hope of modularity is lost once we use “real” objects...
- Luckily, there is a better way — **Dynamic Frames**
- A **dynamic frame** is an expression denoting the set of objects that is used in **reads** and **modifies** clauses.
- The frame is **dynamic** because the expr. may evaluate to different sets of objects, depending on the program state.

ADTs and Framing — Counter with DF

```
class CounterDynamicFrames  
{
```

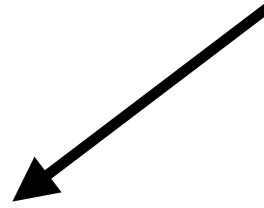
```
    var incs: Cell;
```

```
    var decs: Cell;
```

Set of objects in the representation!

```
    ghost var Value : int;
```

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



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

```
{
```

```
    this in Repr &&
```

```
    incs in Repr && decs in Repr &&
```

```
    incs != decs && Value == incs.data - decs.data
```

```
}
```

ADTs and Framing — Counter with DF

```
class CounterDynamicFrames  
{
```

```
    var incs: Cell;
```

```
    var decs: Cell;
```

```
    ghost var Value : int;
```

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

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

```
{
```

```
    this in Repr &&
```

```
    incs in Repr && decs in Repr &&
```

```
    incs != decs && Value == incs.data - decs.data
```

```
}
```

Validity requires frame
to be accurate



ADTs and Framing — Counter with DF

```
constructor ()  
  ensures Valid() && fresh(Repr - {this})  
  ensures Value == 0  
{  
  incs := new Cell(0);  
  decs := new Cell(0);  
  Value := 0;  
  
  Repr := {this};  
  Repr := Repr + {incs, decs};  
}
```

Constructor initializes Repr
with desired objects



```
method getValue() returns (x: int)  
  requires Valid()  
  ensures x == Value  
{  
  x := incs.data - decs.data;  
}
```

ADTs and Framing — Counter with DF

```
constructor ()  
  ensures Valid() && fresh(Repr - {this})  
  ensures Value == 0  
{  
  incs := new Cell(0);  
  decs := new Cell(0);  
  Value := 0;  
  
  Repr := {this};  
  Repr := Repr + {incs, decs};  
}
```

↑
All objects other than **this**
allocated after constructor invocation

```
method getValue() returns (x: int)  
  requires Valid()  
  ensures x == Value  
{  
  x := incs.data - decs.data;  
}
```

ADTs and Framing — Counter with DF

```
method inc()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
  ensures Value == old(Value) + 1
{
  incs.data, Value := incs.data + 1, Value + 1;
}
method dec()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
  ensures Value == old(Value) - 1
{
  decs.data, Value := decs.data + 1, Value - 1;
}
```

- Any extra objects in Repr are fresh

Dynamic Frames

- Constructor specifies freshness of Repr - `{this}`.
- Any extra objects in Repr are fresh.
- Otherwise (this may be fine):

```
method ShareMe(cnt: CounterDynamicFrames)
  requires Valid() && cnt.Valid()
  modifies Repr
  ensures Valid() && Value == old(Value)
{
  if (incs.data == cnt.incs.data && decs != cnt.incs) {
    incs := cnt.incs;
    Repr := Repr + {incs};
  }
}
```

ADTs and Framing — Dynamic Frames

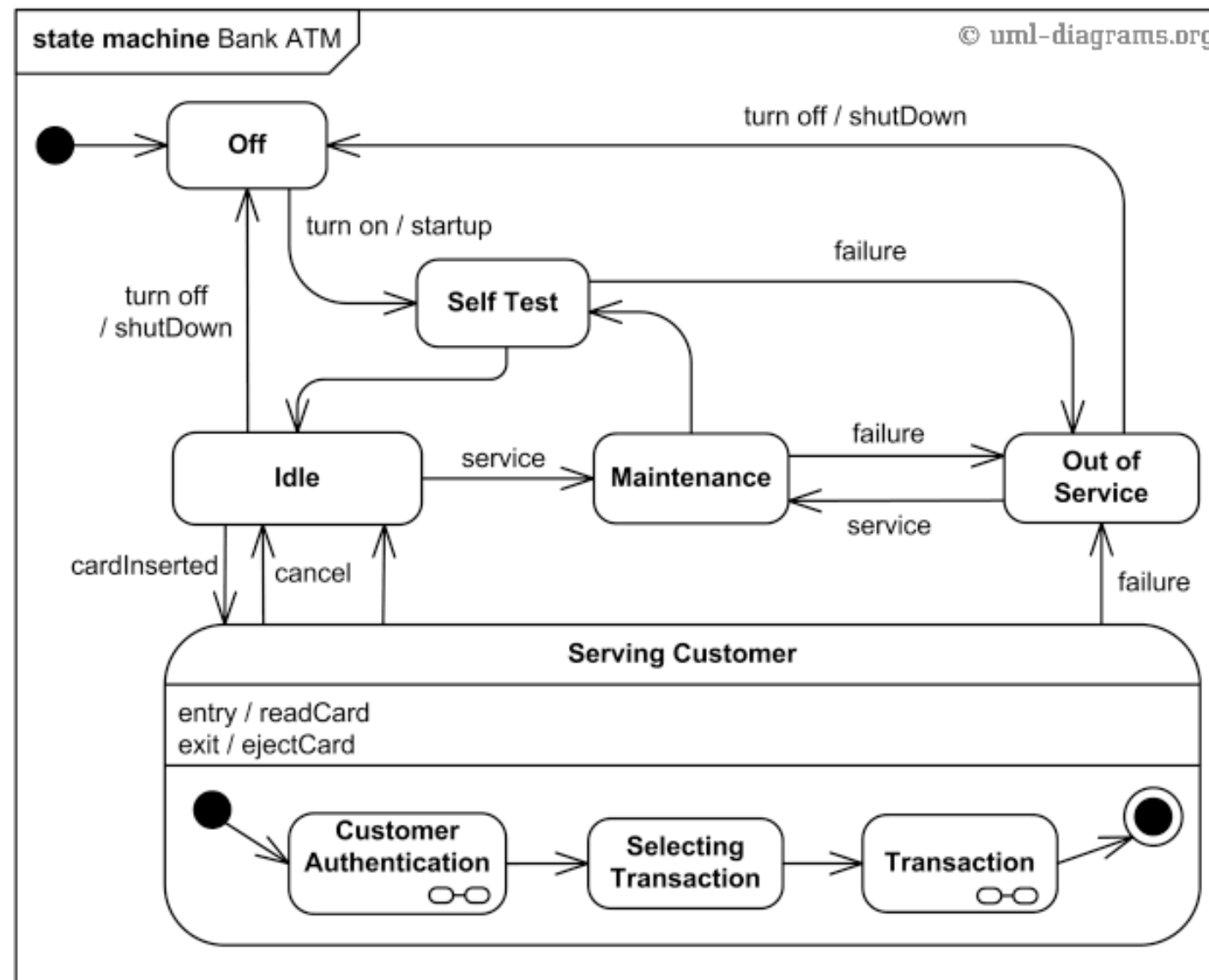
- Each object carries with it a (ghost) set of the objects in its memory representation (Repr).
- Invariants enforce validity of Repr.
- Constructor specifies freshness of Repr - `{this}`.
- Methods now **read** / **modify** Repr instead of exposing internal state (generally, `fresh(Repr - old(Repr))`).
- Idiom supports arbitrary nesting of objects (can refer to Repr of fields and so on).

Part IV

ADTs and Typestates

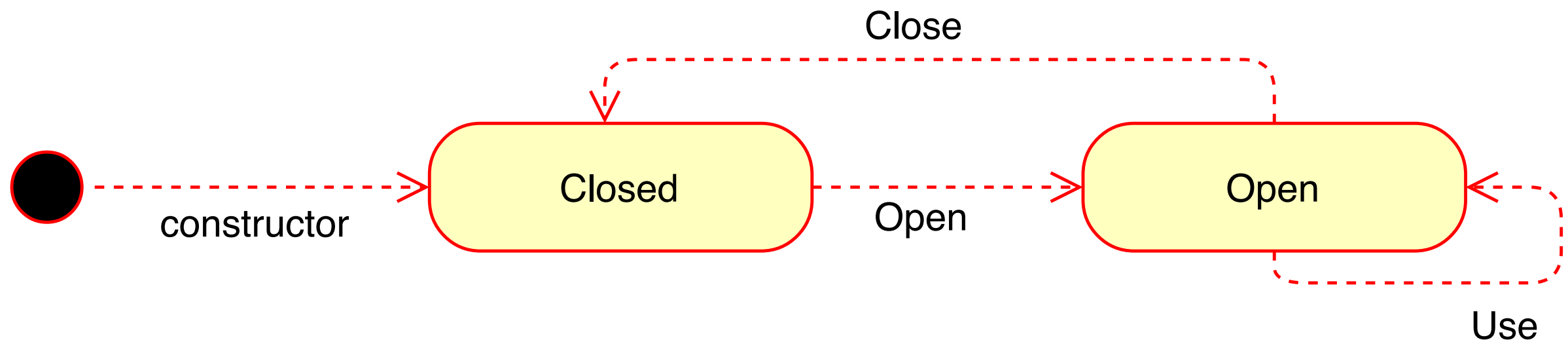
State Transition Diagrams

- 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



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;  
  
    predicate OpenState()  
    reads this  
    { ... }  
  
    predicate ClosedState()  
    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;
```

```
  predicate OpenState()
```

```
  reads this
```

```
  { ... }
```

```
  predicate ClosedState()
```

```
  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;
```

```
  predicate OpenState()
```

```
  reads this
```

```
  { ... }
```

```
  predicate Closed() method UsingTheResource()
```

```
  reads this
```

```
  { ... }
```

```
  constructor (
```

```
  ensures Closed
```

```
  { ... }
```

```
  ...
```

```
}
```

```
{
```

```
  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;  
    ...  
    predicate OpenState()  
        requires Valid()  
        reads Repr  
    { h != null && size == h.Length }  
  
    predicate ClosedState()  
        requires Valid()  
        reads Repr  
    { size == 0 }  
  
    constructor ()  
    ensures Valid() && fresh(Repr-{this}) && ClosedState()  
    { size, h := 0 , null; Repr := this; }
```

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 Repr
```

```
    requires Valid() && ClosedState() && N > 0
```

```
    ensures Valid() && OpenState() && fresh(Repr-old(Repr))
```

```
    { h, size := new int[N], N;
```

```
      Repr := Repr + {h}; }
```

```
    method Close()
```

```
    modifies Repr
```

```
    requires Valid() && OpenState()
```

```
    ensures Valid() && ClosedState() && fresh(Repr-old(Repr))
```

```
    { h, size := null, 0;
```

```
      Repr := Repr - {h}; }
```

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 can be caused by misusing the representation type. Notice that states are also used as invariants, essential to execute different methods.

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 ReplInv (no variants/less specific).

Key Points

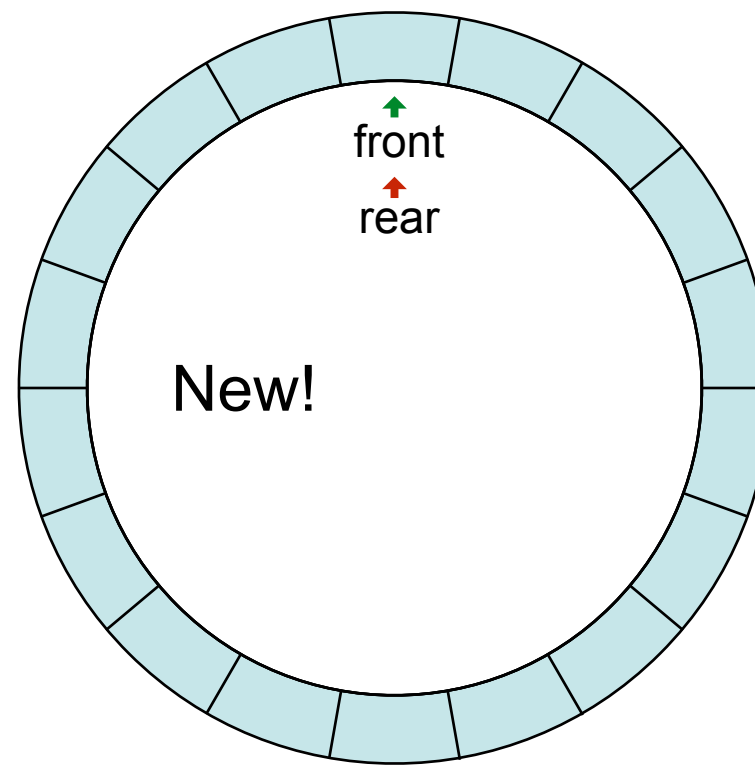
- Software Design Time
 - Abstract Data Type
 - What are the Abstract States / Concrete States?
 - What is the Representation Invariant?
 - What is the Abstraction Mapping?
- Software Construction Time
 - Make sure constructor establishes the Replnv
 - Make sure all operations preserve the Replnv
 - **they may assume the Replnv**
 - **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

Part V

Typestates (Example)

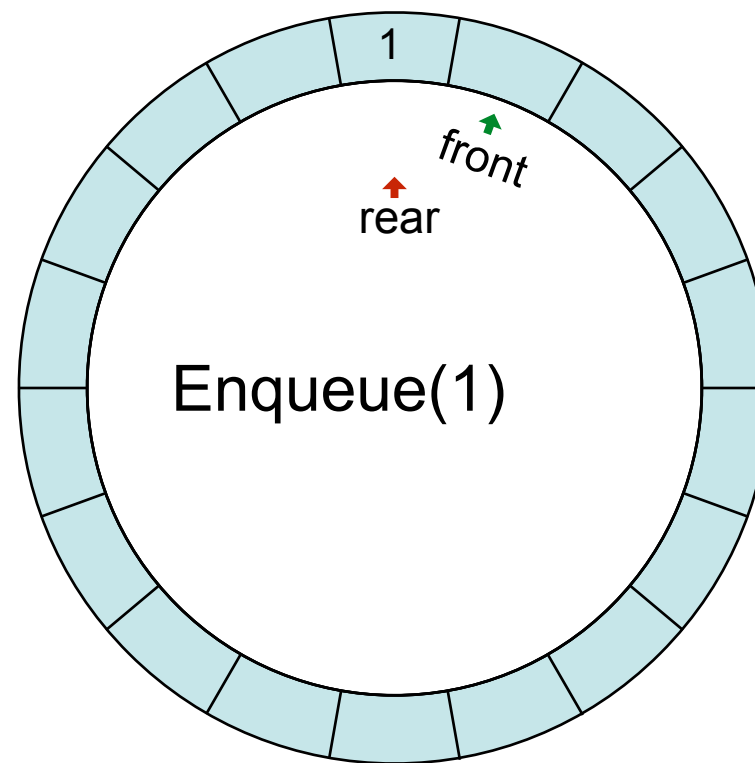
TypeStates - Queue

- An implementation using a circular buffer...



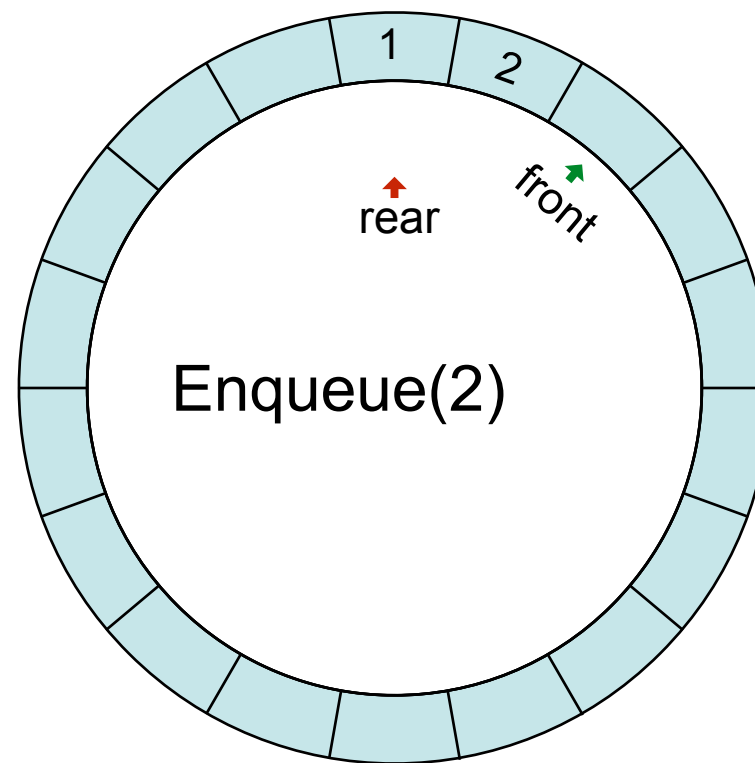
TypeStates - Queue

- An implementation using a circular buffer...



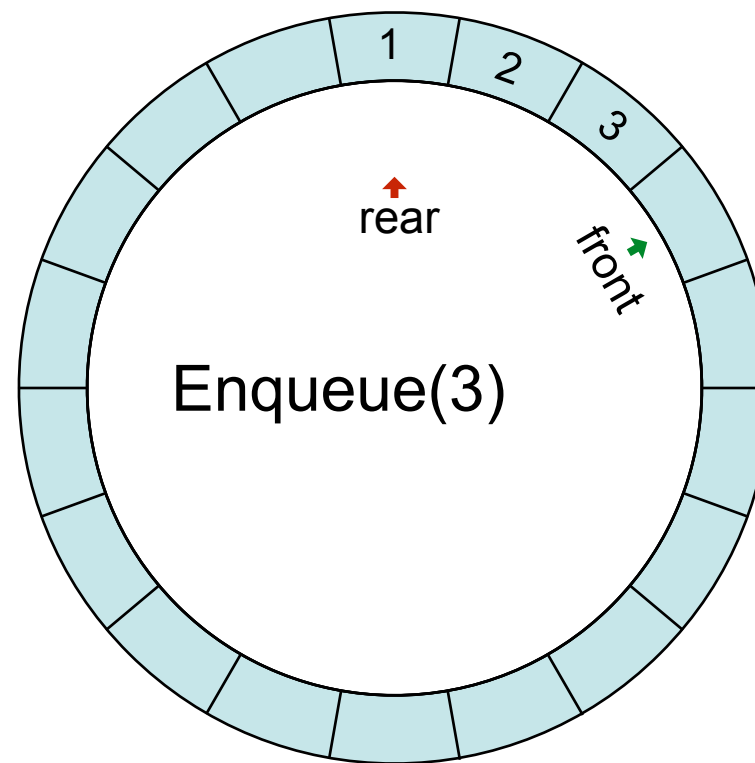
TypeStates - Queue

- An implementation using a circular buffer...



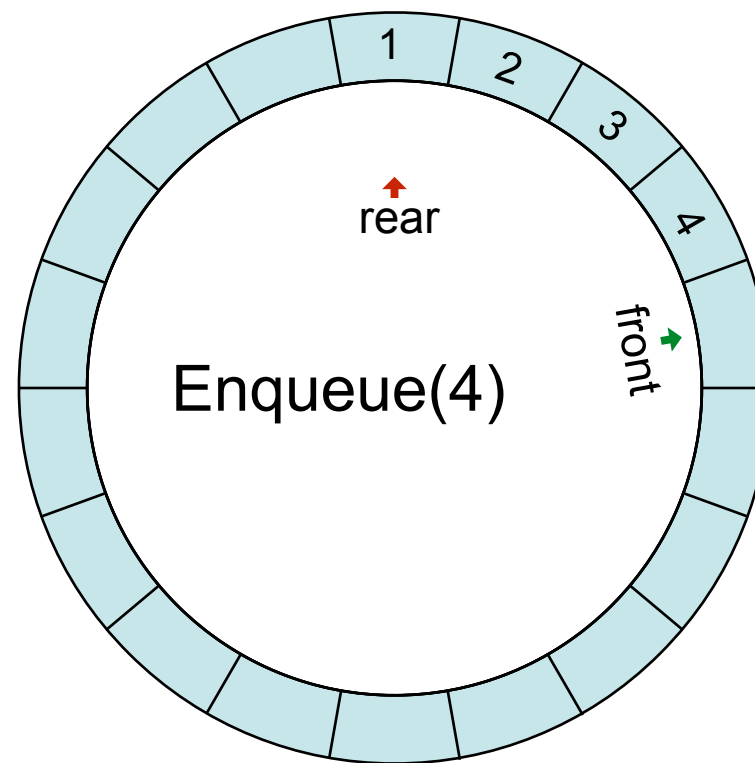
TypeStates - Queue

- An implementation using a circular buffer...



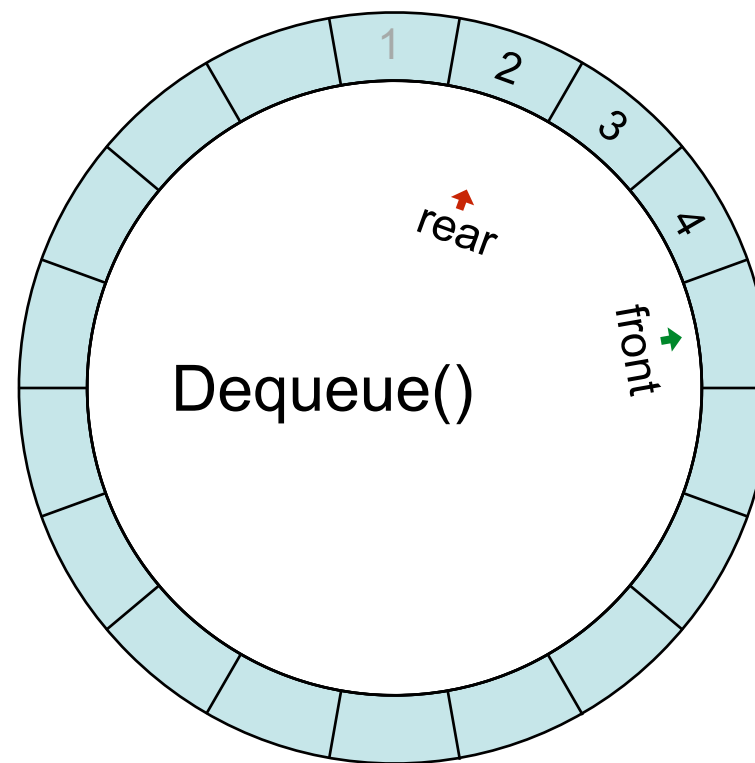
TypeStates - Queue

- An implementation using a circular buffer...



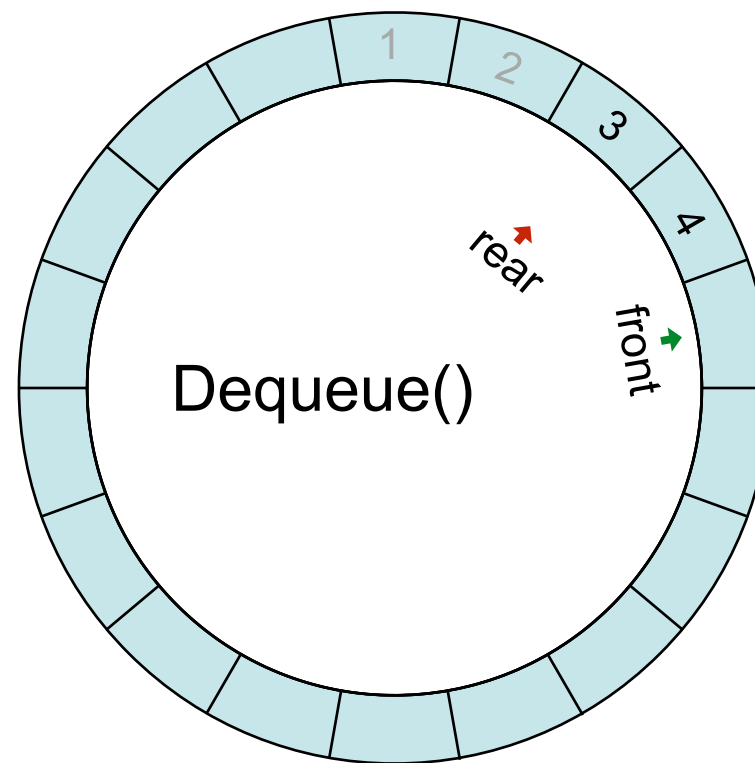
TypeStates - Queue

- An implementation using a circular buffer...



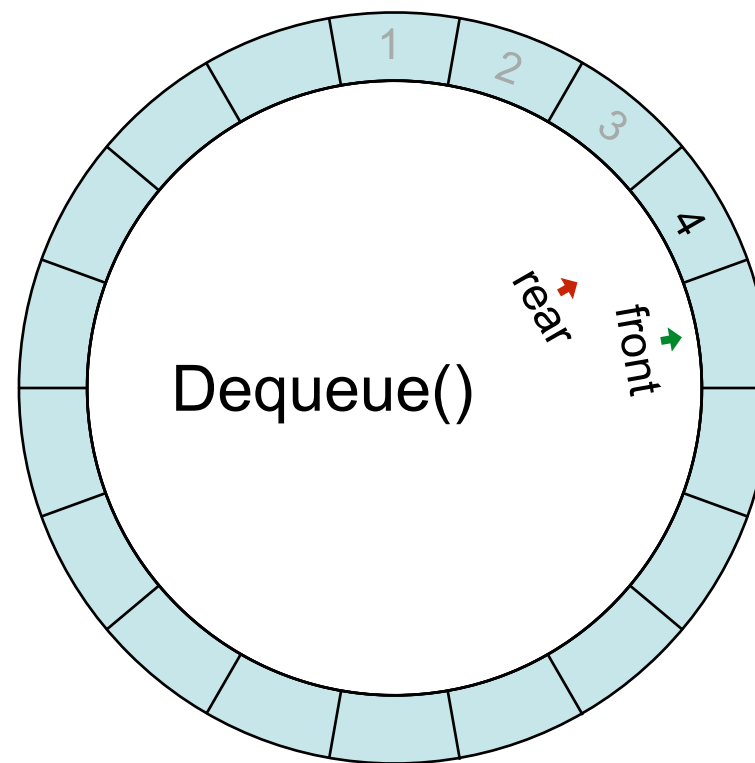
TypeStates - Queue

- An implementation using a circular buffer...



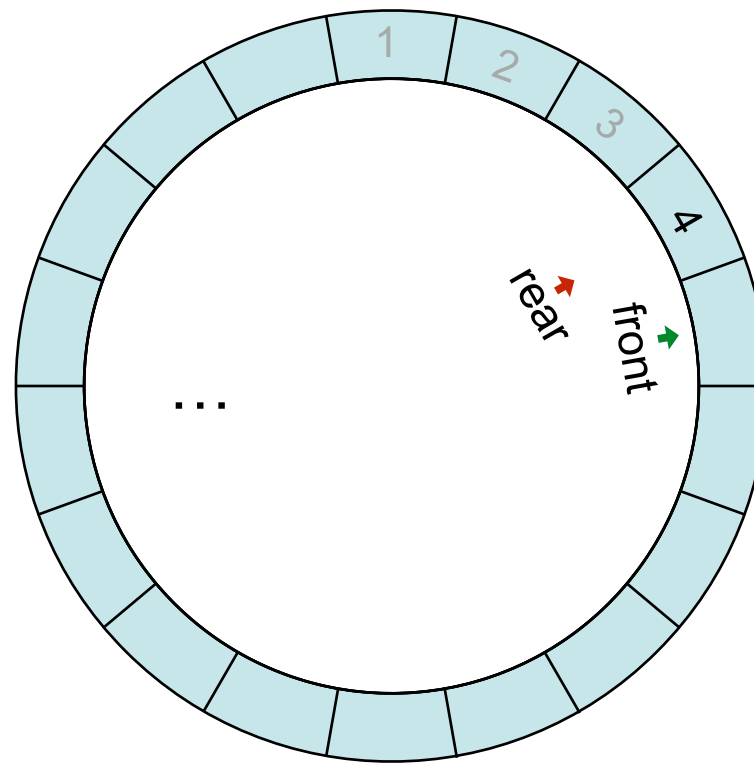
TypeStates - Queue

- An implementation using a circular buffer...



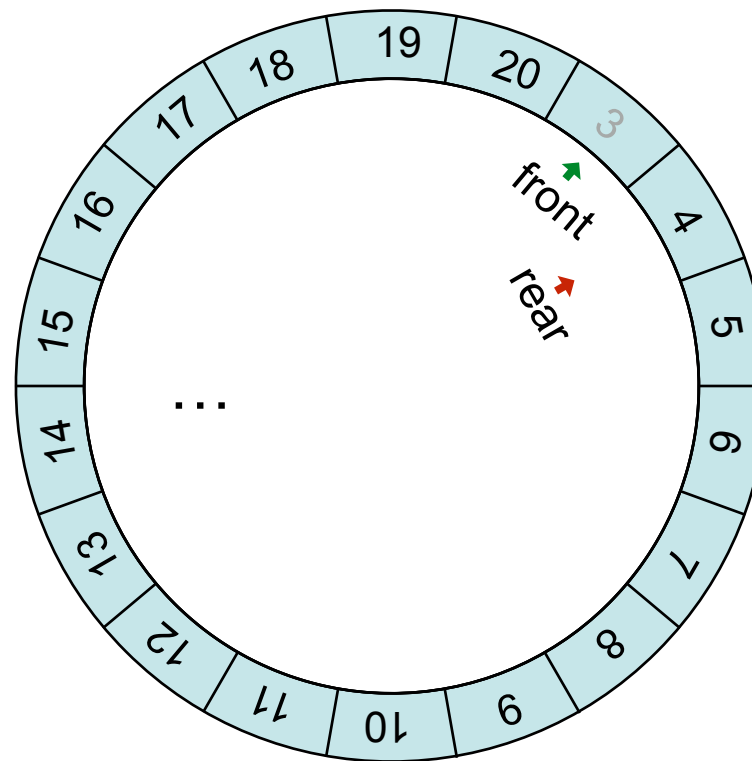
TypeStates - Queue

- An implementation using a circular buffer...



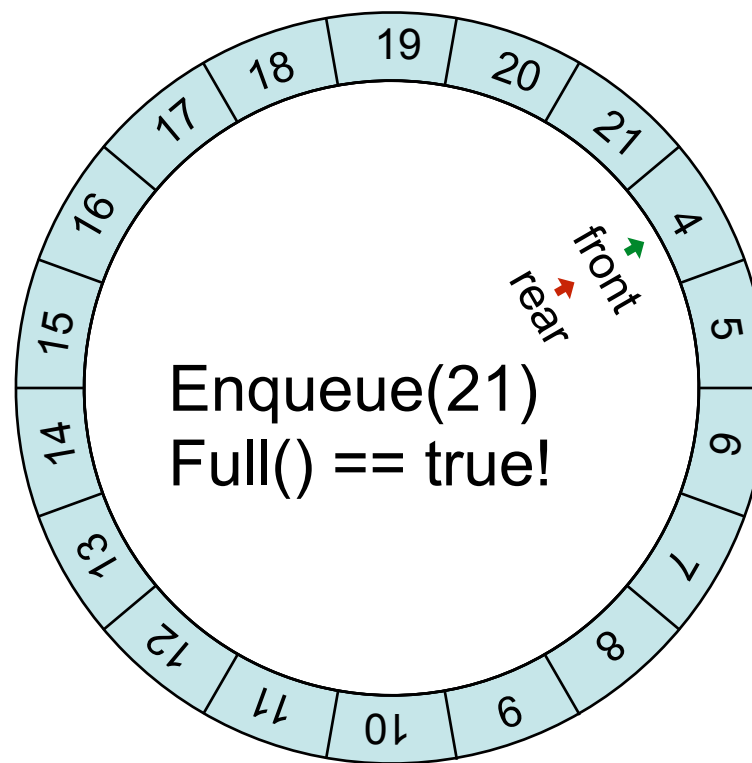
TypeStates - Queue

- An implementation using a circular buffer...



TypeStates - Queue

- An implementation using a circular buffer...



TypeStates - Queue

- An implementation using a circular buffer...

```
class {autocontracts} Queue {  
  // Representation type  
  var a:array<int>;  
  var front: int;  
  var rear: int;  
  var numberOfElements: int;  
  ...  
  
  constructor(N:int)  
    requires 0 < N  
  {  
    a := new int[N];  
    front, rear := 0 , 0;  
    Repr := {this};  
    numberOfElements := 0;  
  }  
  ...  
}
```

TypeStates - Queue

- What's wrong with it? a ReplInv is necessary to maintain front and rear within bounds...

```
class {:autocontracts} Queue {  
  
  ...  
  method Enqueue(V:int)  
  {  
    a[front] := V;  
    front := (front + 1)%a.Length;  
    numberOfElements := numberOfElements + 1;  
  }  
  
  method Dequeue() returns (V:int)  
  {  
    V := a[rear];  
    rear := (rear + 1)%a.Length;  
    numberOfElements := numberOfElements - 1;  
  }  
}
```

TypeStates - Queue

```
class {:autocontracts} Queue {  
  // Representation type  
  var a:array<int>;  
  var front: int;  
  var rear: int;  
  var numberOfElements: int;  
  
  // Representation invariant  
  predicate RepInv()  
    reads Repr  
  { 0 <= front < a.Length && 0 <= rear < a.Length }  
  
  constructor(N:int)  
    requires 0 < N  
    ensures RepInv()  
  {  
    a := new int[N];  
    front, rear := 0 , 0;  
    Repr := {this}  
    numberOfElements := 0;  
  }  
  ...  
}
```


TypeStates - Queue

```
class {:autocontracts} Queue {  
  ...  
  method Enqueue(V:int)  
    requires RepInv()  
    ensures RepInv()  
  {  
    a[front] := V;  
    front := (front + 1)%a.Length;  
    numberOfElements := numberOfElements + 1;  
  }  
  
  method Dequeue() returns (V:int)  
    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

```
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);  
}
```

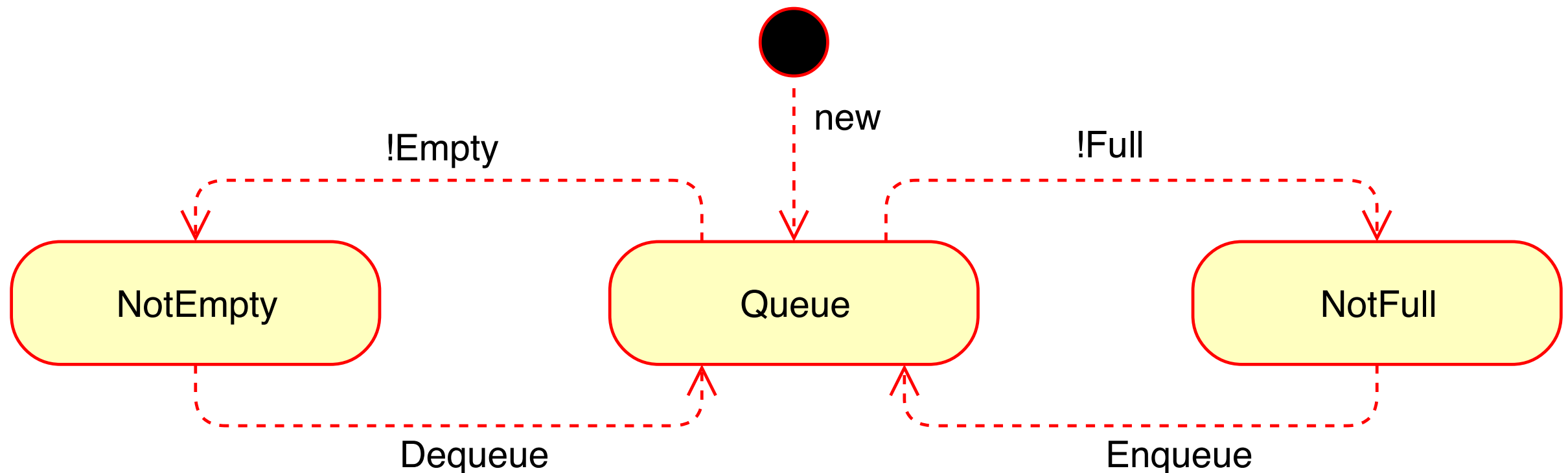
TypeStates - Queue

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

```
predicate RepInv()  
  reads this, Repr  
{  
  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  
}
```

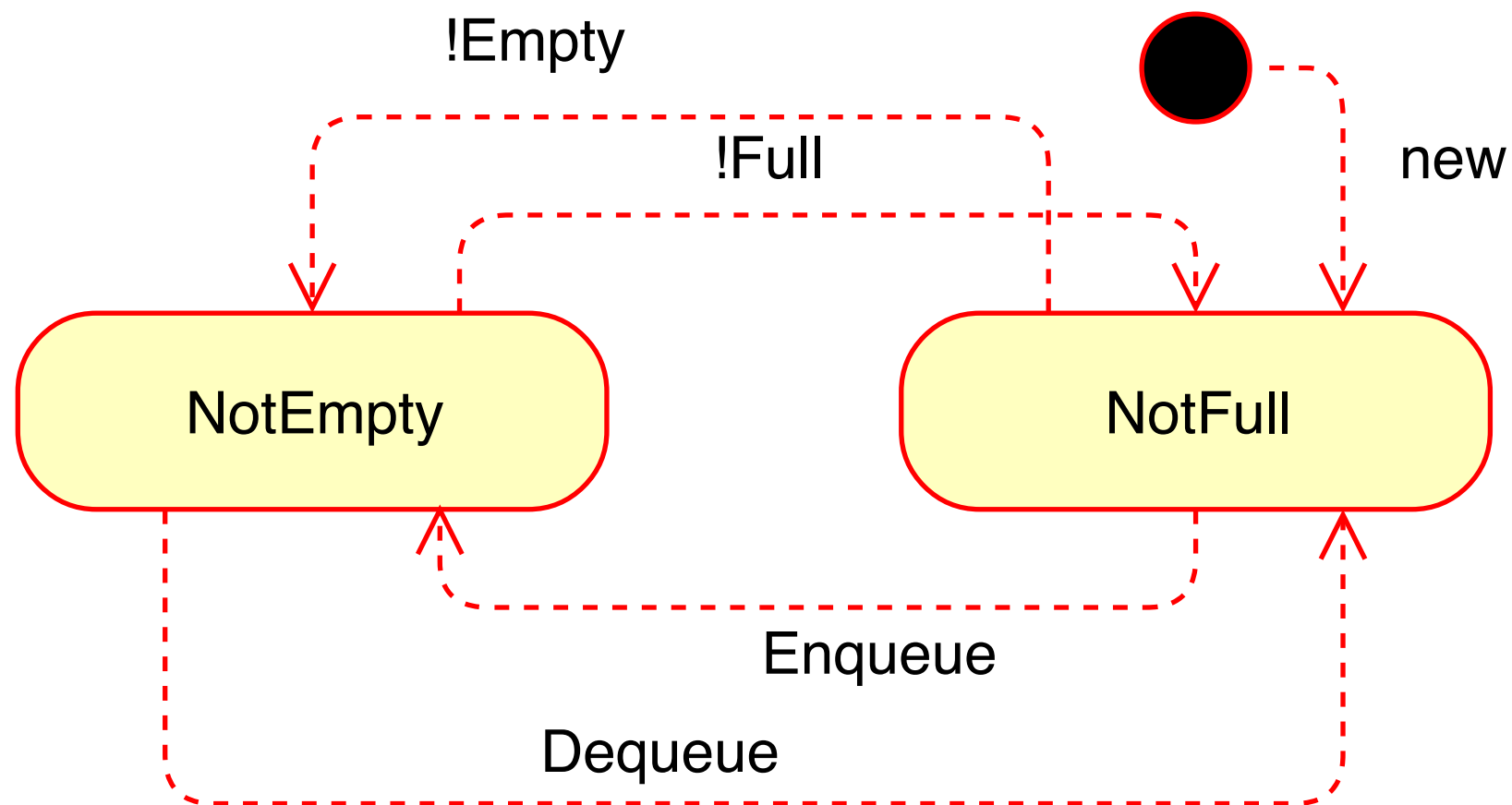
TypeStates - Queue

- 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 {:autocontracts} Queue {  
  ...  
  predicate NotFull()  
    reads this  
  { RepInv() && numberOfElements < a.Length }  
  
  predicate NotEmpty()  
    reads this  
  { RepInv() && numberOfElements > 0 }  
  
  constructor(N:int)  
    requires 0 < N  
    ensures NotFull()  
  { ... }  
  
  method Enqueue(V:int)  
    requires NotFull()  
    ensures NotEmpty()  
  { ... }  
  
  method Dequeue() returns (V:int)  
    requires NotEmpty()  
    ensures NotFull()  
  { ... }
```

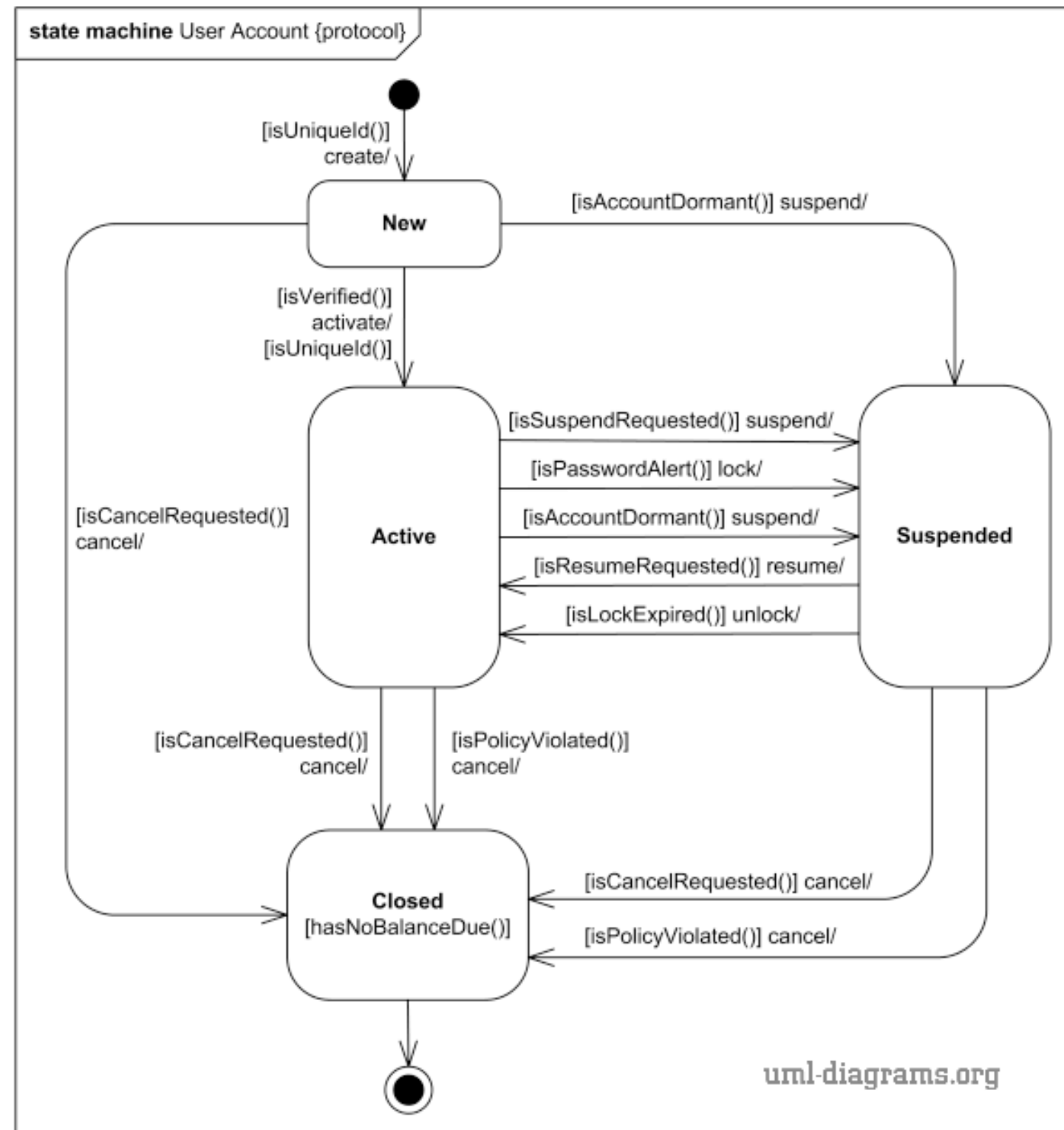
```
method Main()  
{  
  var q:Queue := new Queue(4);  
  var r:int;  
  
  q.Enqueue(1);  
  r := q.Dequeue();  
  r := q.Dequeue();  
  q.Enqueue(2);  
  q.Enqueue(3);  
  q.Enqueue(4);  
  q.Enqueue(5);  
}
```

TypeStates - Queue

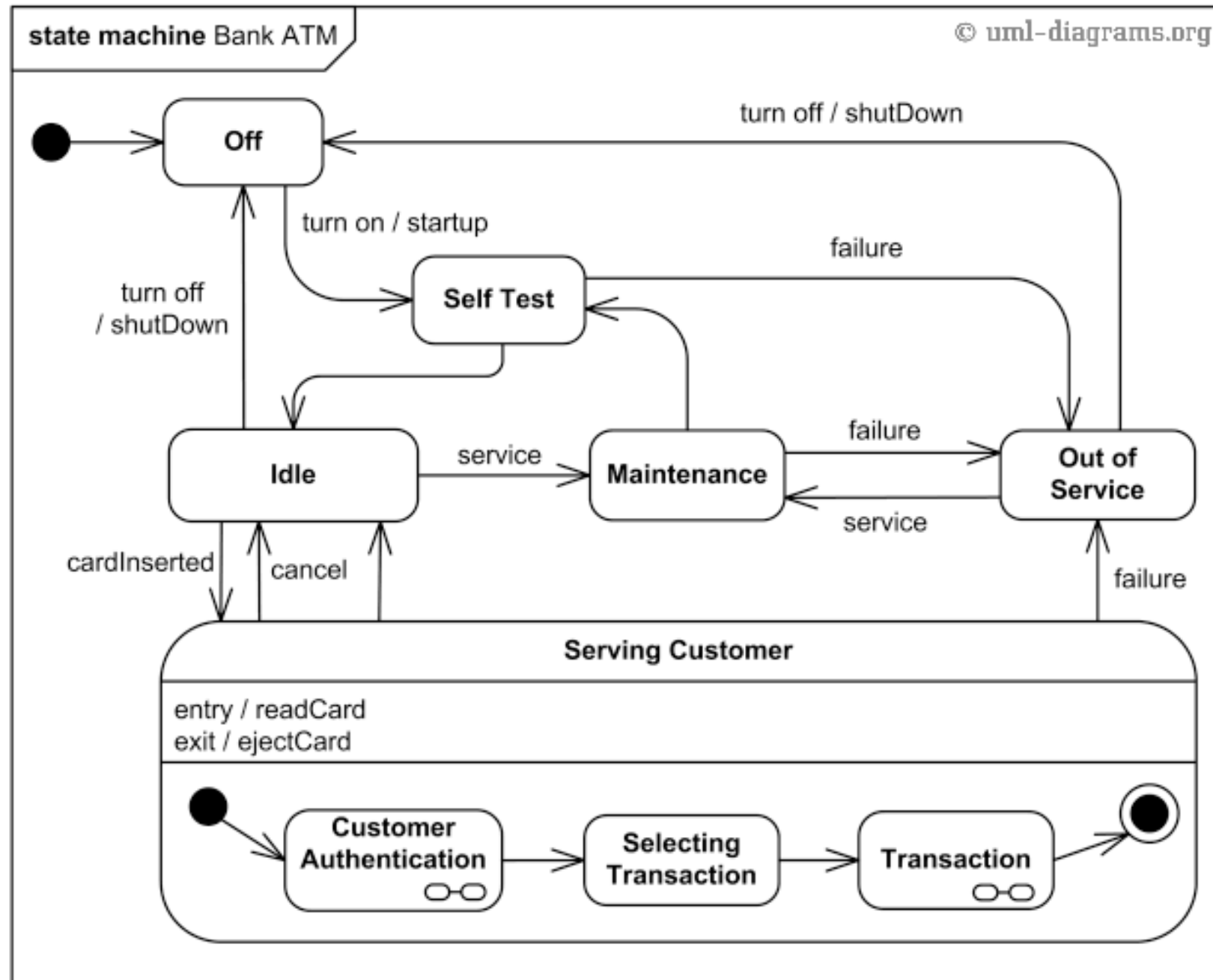
- Dynamic Tests ensure the proper state for a given operation...

```
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



TypeStates - ATM



Exercise 14

- ADT PSet

```
// Use the set implementation ASet of Lecture 3 and  
// add to the representation invariants the property about  
// all values being positive. Make the post-conditions  
// stronger using that property
```

```
// Design some client methods and write assertions  
// that are a consequence of the abstract invariant.
```

Exercise 15

- Extended Bank Account that stores operations

```
// Implement a bank account whose internal
// representation is an array of bank movements
// (debit, credit).

// Make the adequate abstract representation for
// the balance and define the soundness mapping.
```

Exercise 16 (Key-Value Store)

- This exercise focuses on the development of a small but rigorously 100% bug free dictionary abstract data type (ADT). Consider that the type of keys is the K and the type of values V .
 - The ADT must provide the following operations
 - method `assoc(k:K,v:V)`**
 - // associates val v to key k in the dictionary
 - method `find(k:K)` returns $(r:RES)$**
 - // returns `NONE` if key k is not defined in the dict, or `SOME(v)` if the dictionary
 - method `delete(k:K)`**
 - // removes any existing association of key k in the dictionary
- Every dictionary entry should be represented by a record of type `ENTRY`
- datatype `ENTRY = PACK(key: K, val: V)`**
- The result of function `find` should be represented with type
- datatype `RES = NONE | SOME(V)`**
- The representation type of your ADT should be a mutable data structure (advice: start by something simple - an array, an ordered array, or a closed hashtable).
 - Express the representation invariant using an auxiliary boolean function **`RepInv()`**

Exercise (2nd Handout 17/18)

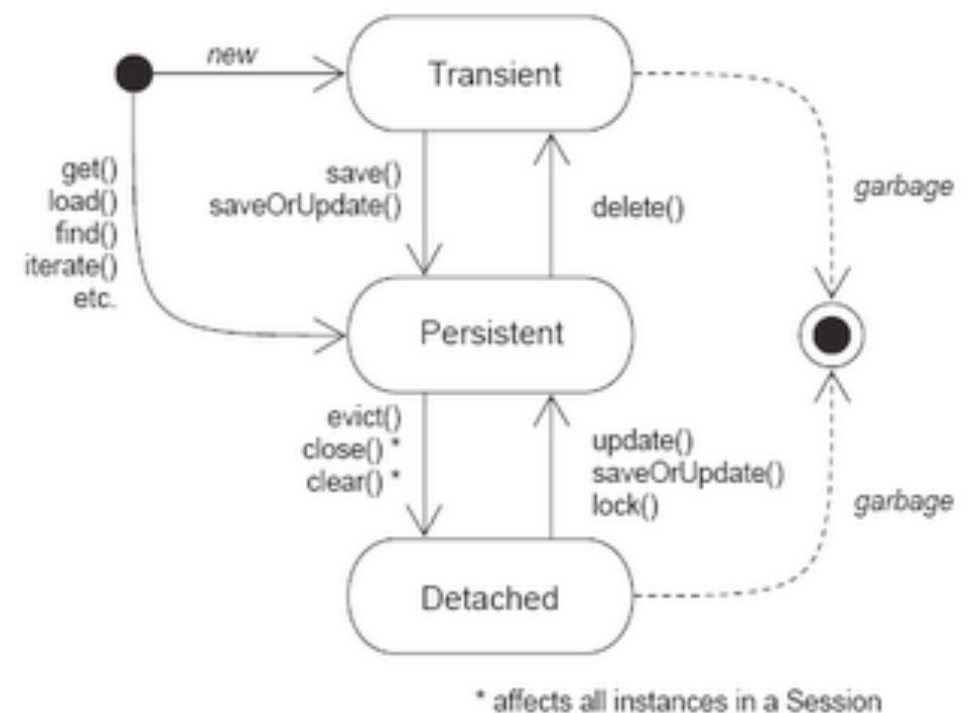
Implement an ADT representing a Persistent Entity of the JPA library such that each item has the following states:

You must research about JPA states and implement two example classes that simulate such behaviour.

Persistency can be achieved by storing items in a collection/store/database represented by an array of objects.

If an item is not persistent, it does not have a valid identifier, when stored or updated the data is copied to the collection (no references).

Consider as an example a Person entity (name, age).



Exercise (2nd Handout 17/18)

To make it easier, consider the simplified state diagram.

Operation find is a method of the store class that instantiates an object (copy from the DB) with a valid id and a valid store connection.

Implement a class Person that accepts the store as parameter in methods save(store) and update(store).

When a Person is transient state, it has no valid id.

When a Person is in detached state, It has no valid store connection (offline from the database).

