Construction and Verification of Software

2017 - 2018

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 5 - Type States

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

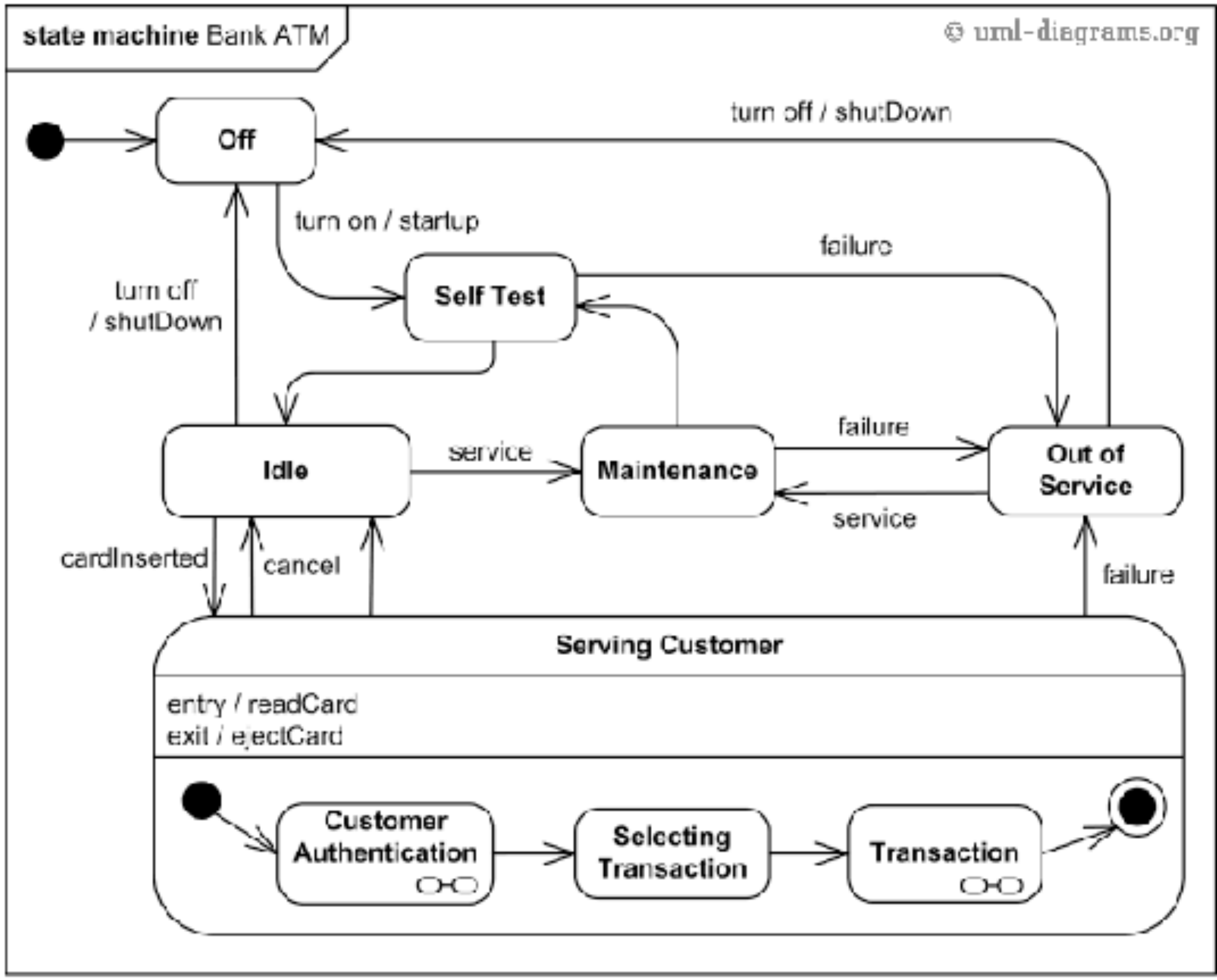
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

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.

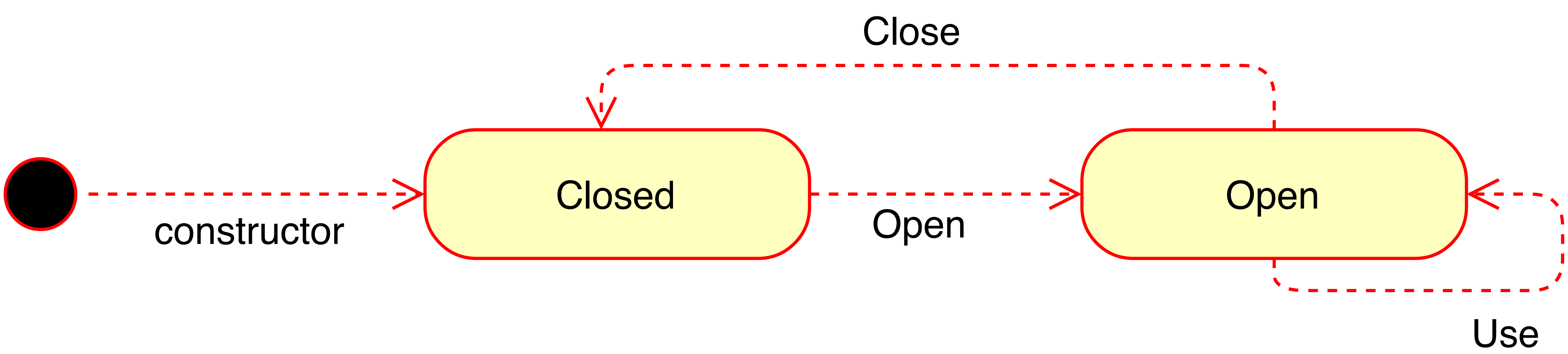
- If a state if formally connected to conditions over the state of an object, the correction of state transitions may be mechanically checked



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 ClosedState():bool
    reads this
    { …  }

    constructo
    ensures  C
    { … }

  …

}
```

```
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 C
    reads this
    { …  }

    constructo
    ensures  C
    { … }

  …

}
```

```
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

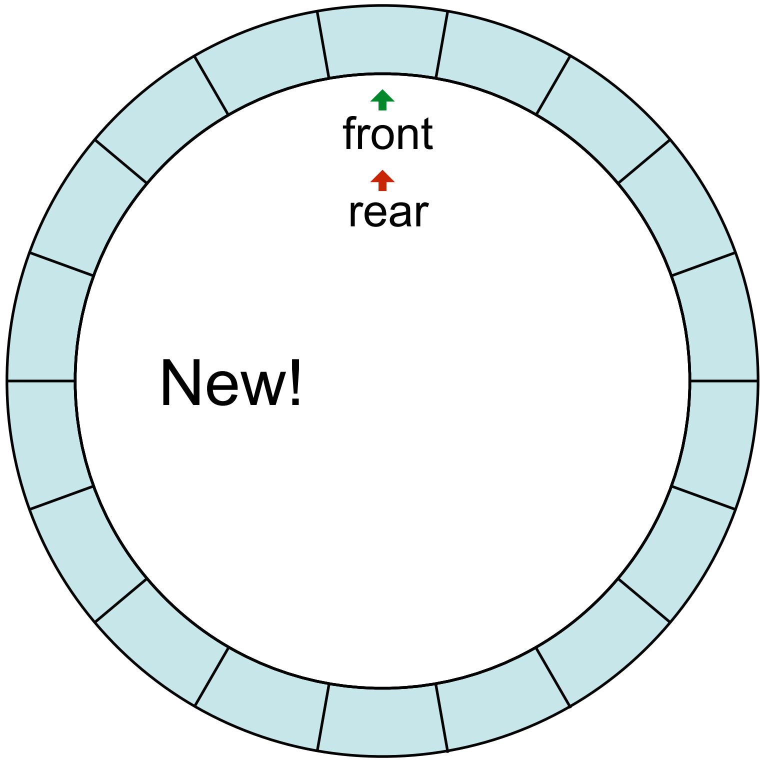
- In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states
- It is often enough to expose TypeState assertions to ensure ADT soundness and no runtime errors
- In general, full functional specifications in terms the abstract state 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 Rep Inv
  - Make sure all operations preserve the Rep Inv
    - they may assume the Rep Inv
    - 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

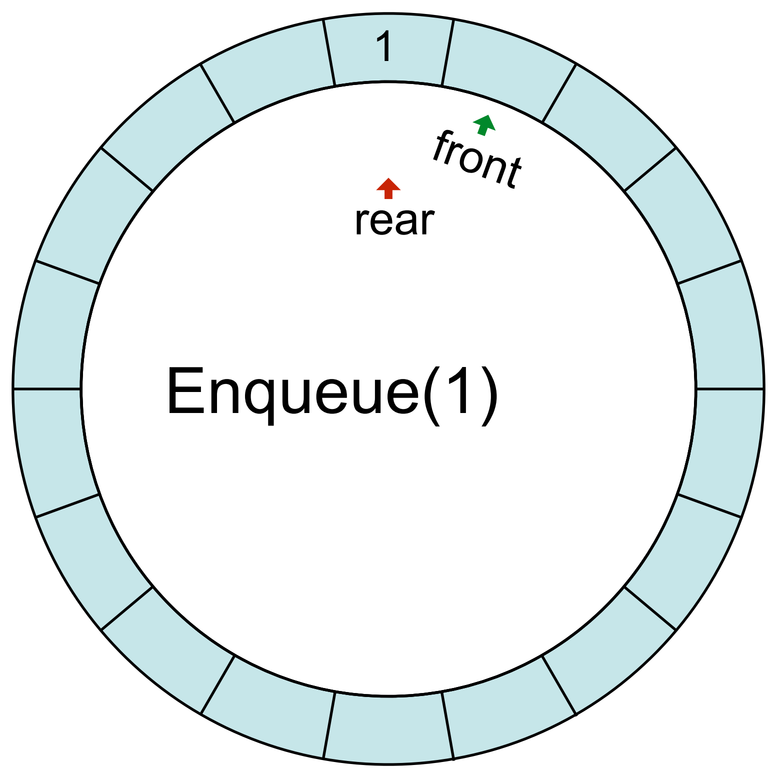
TypeStates - Queue

- An implementation using a circular buffer…



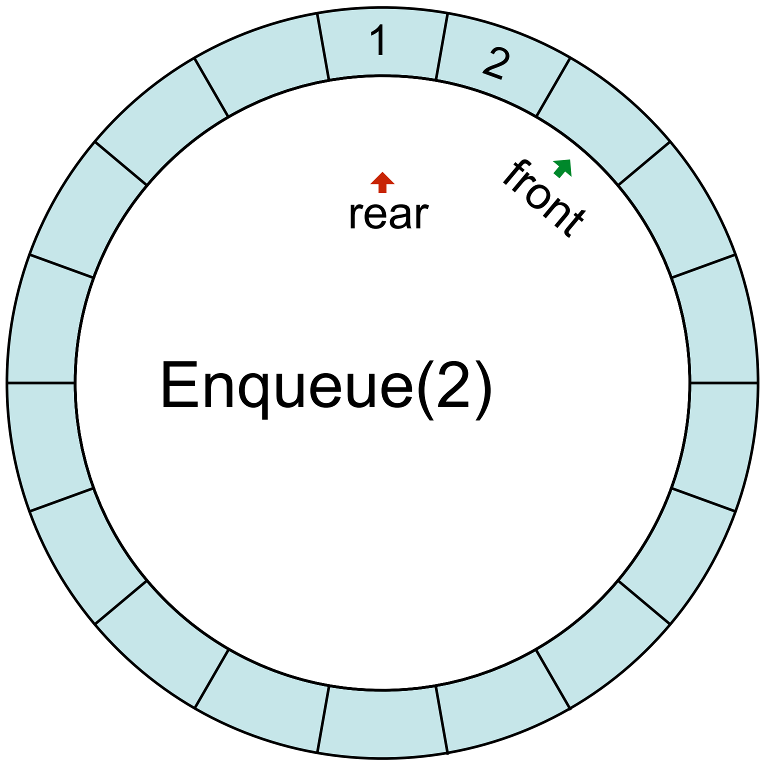
TypeStates - Queue

- An implementation using a circular buffer…



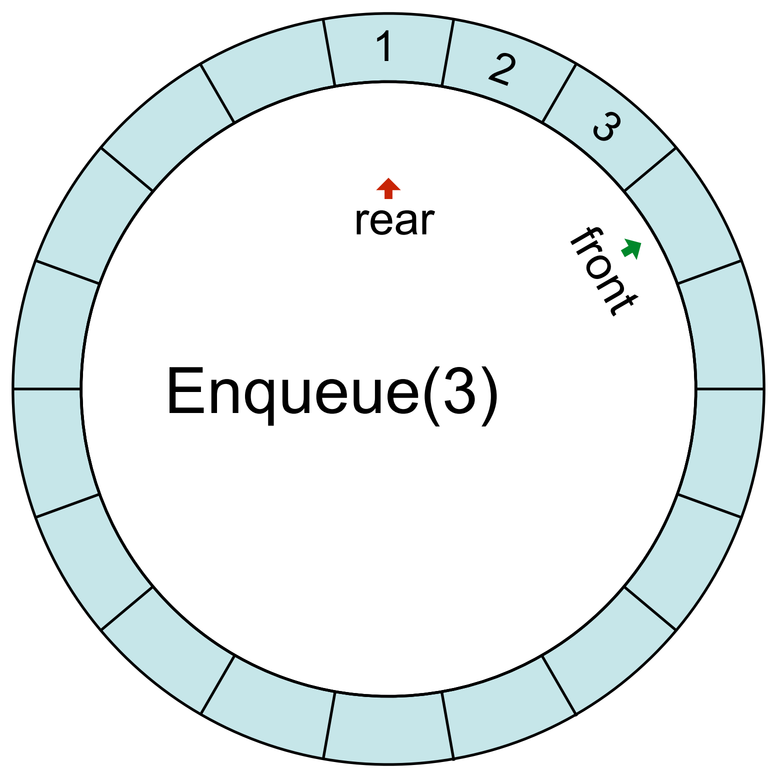
TypeStates - Queue

- An implementation using a circular buffer…



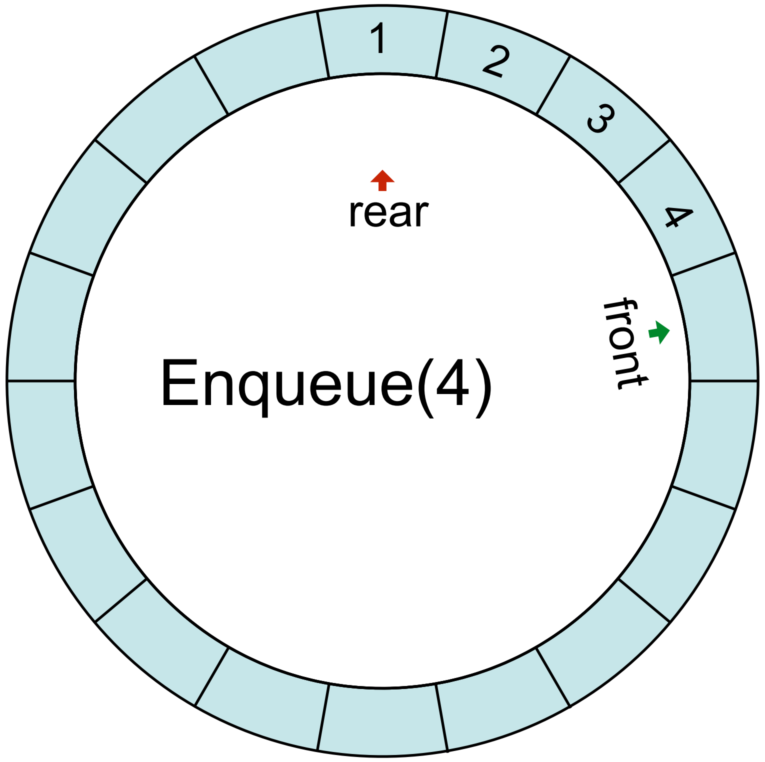
TypeStates - Queue

- An implementation using a circular buffer…



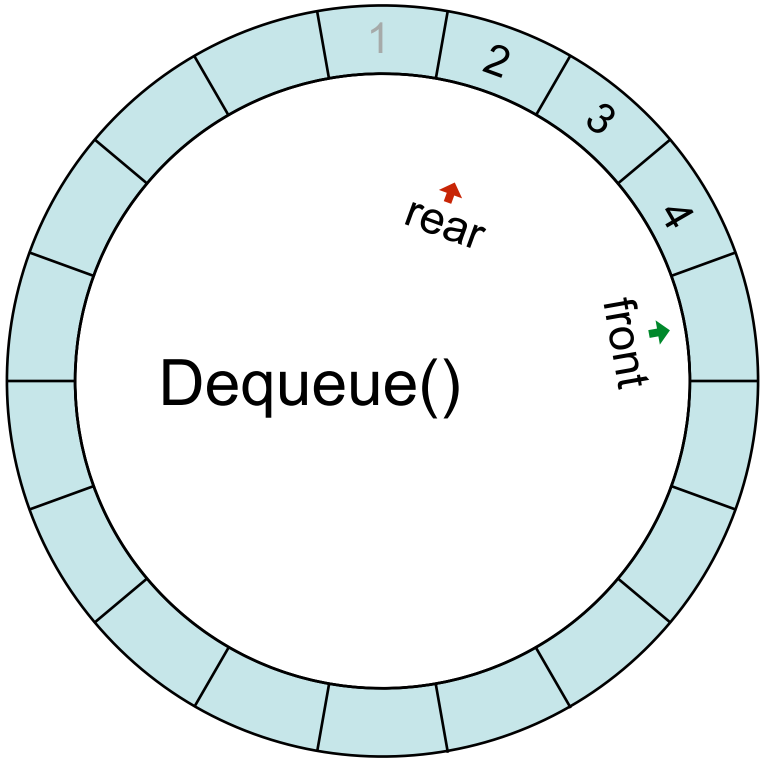
TypeStates - Queue

- An implementation using a circular buffer…



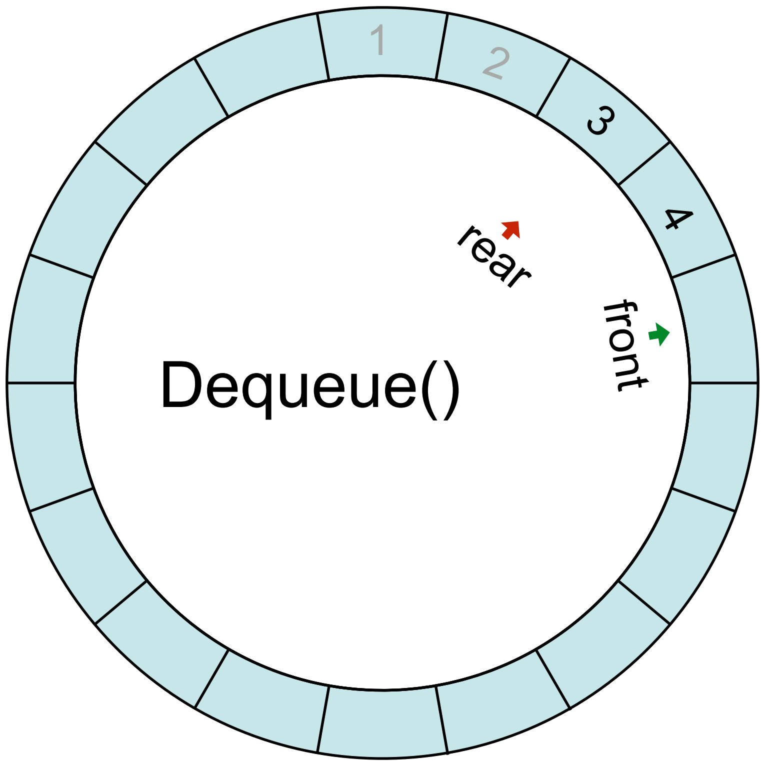
TypeStates - Queue

- An implementation using a circular buffer…



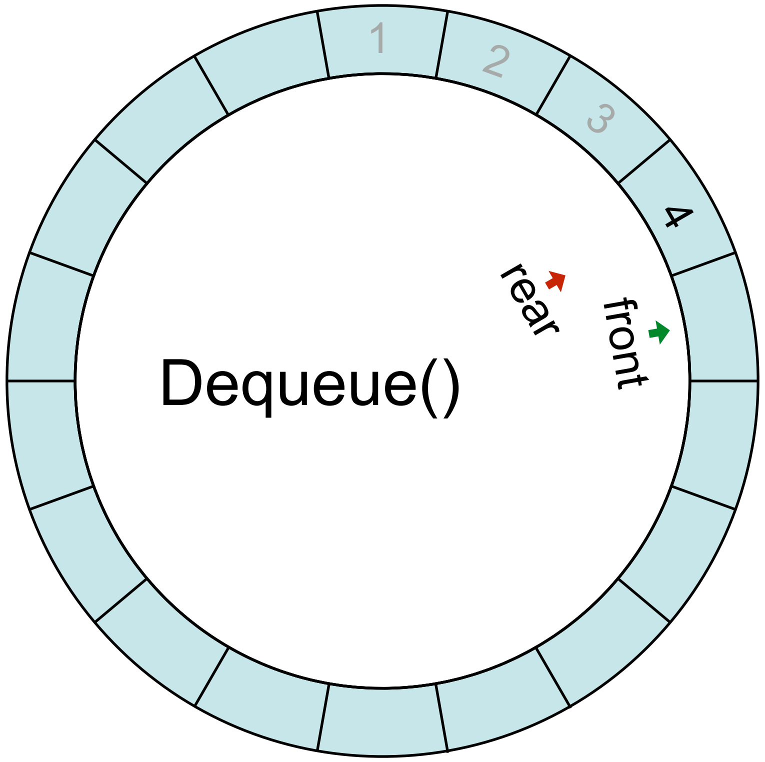
TypeStates - Queue

- An implementation using a circular buffer…



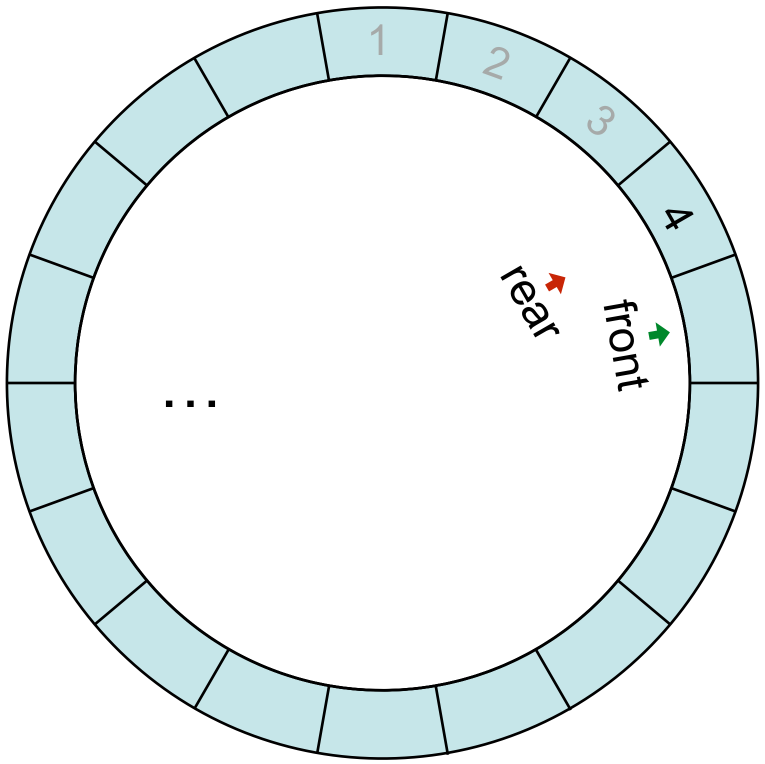
TypeStates - Queue

- An implementation using a circular buffer…



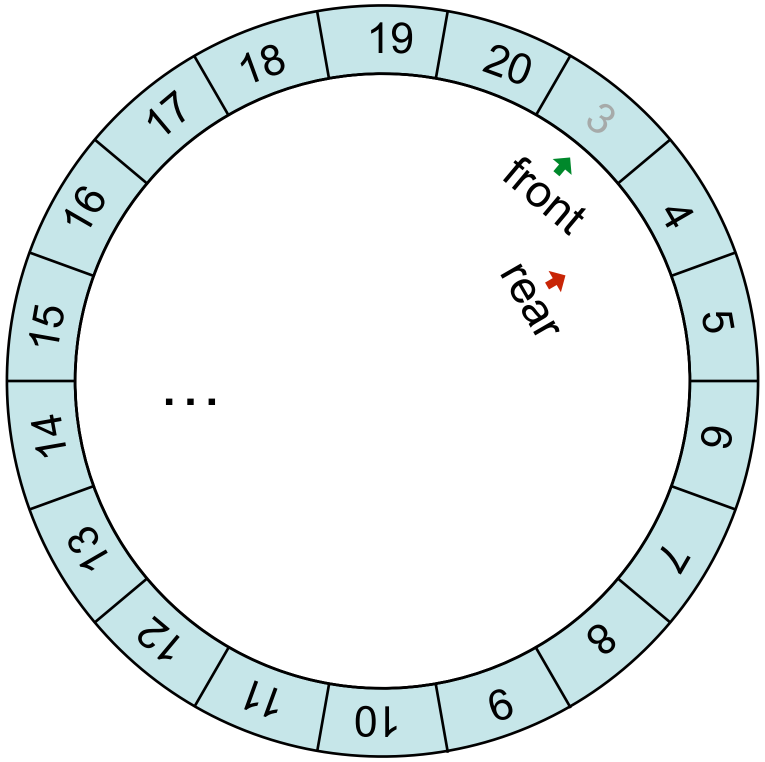
TypeStates - Queue

- An implementation using a circular buffer…



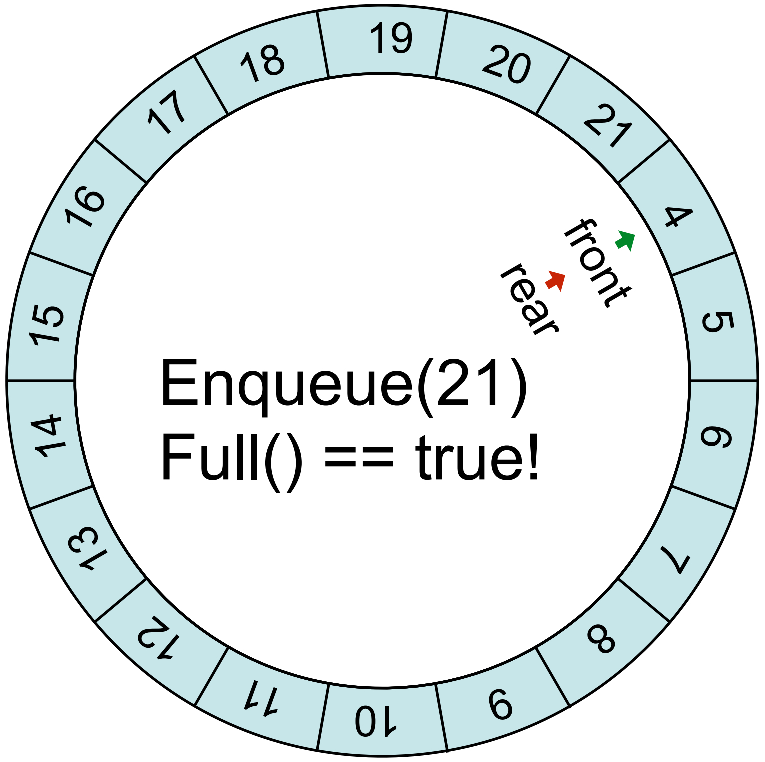
TypeStates - Queue

- 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);
}
```

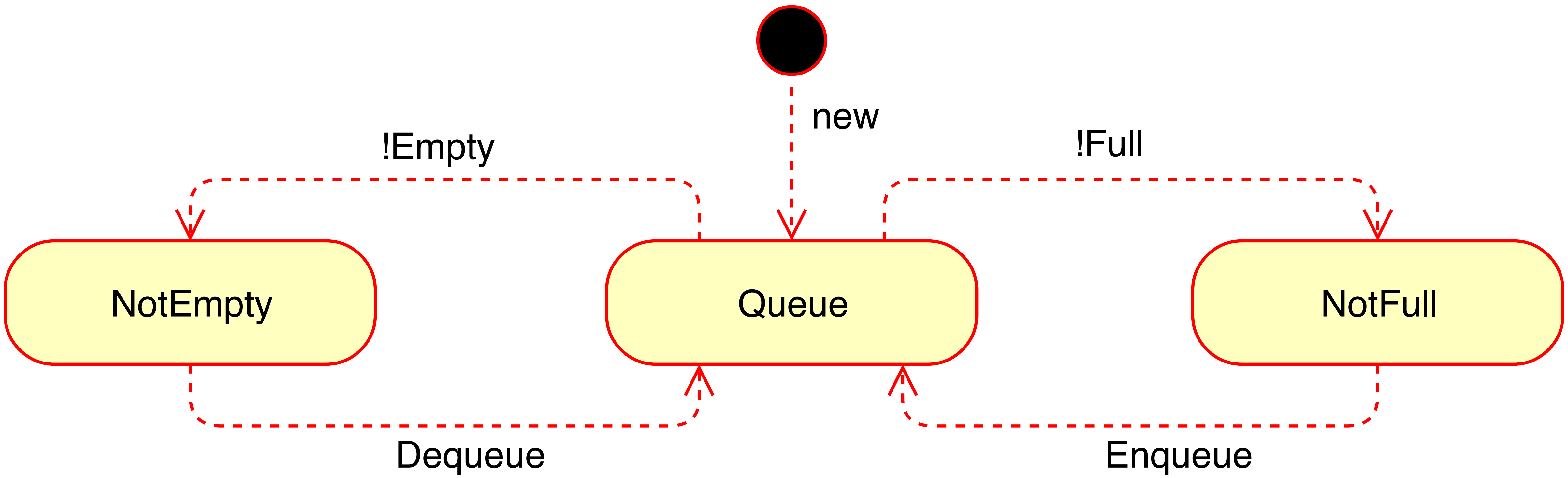
TypeStates - Queue

- 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
 }
```

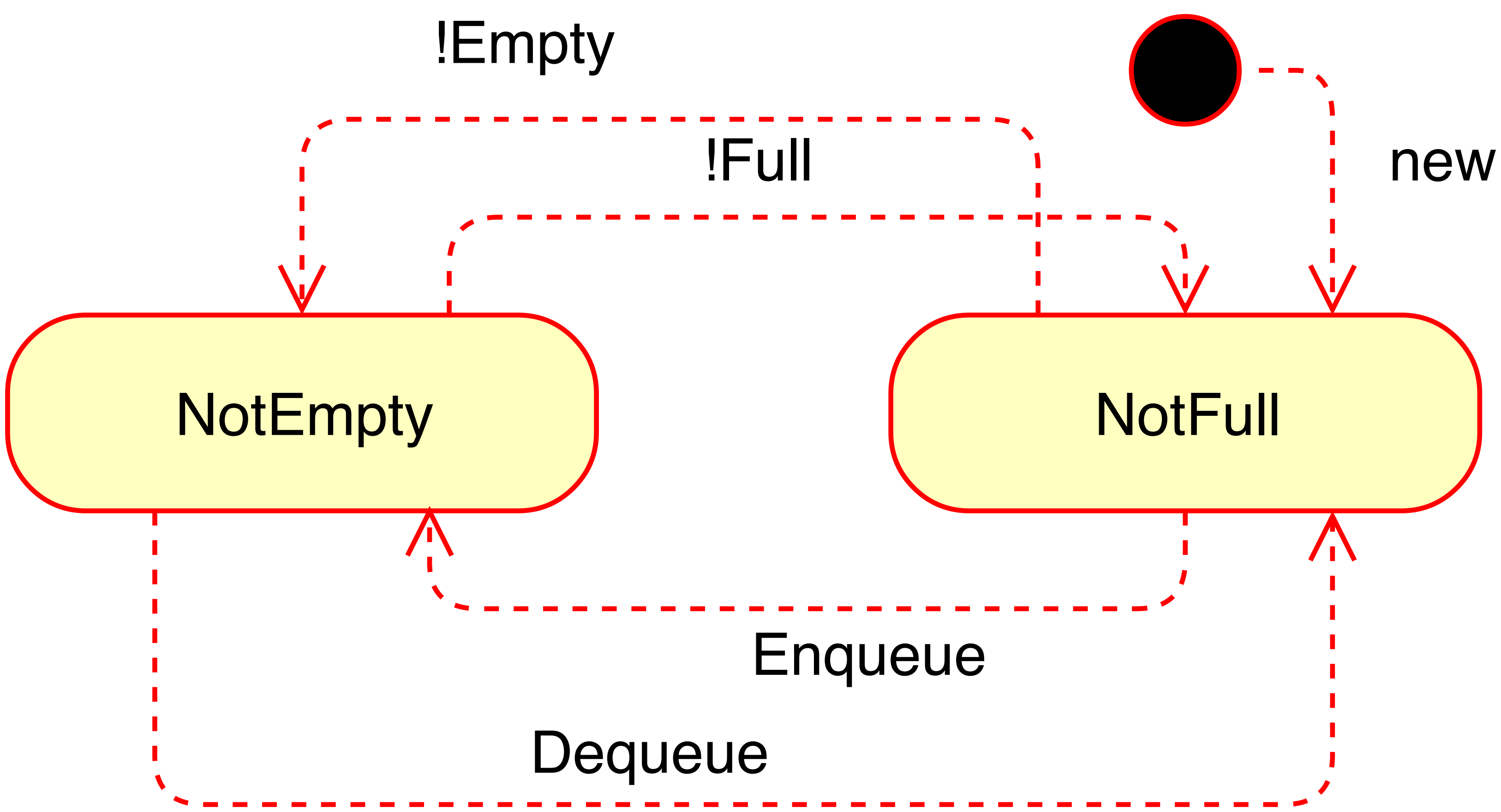
TypeStates - Queue

- 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);
}
```

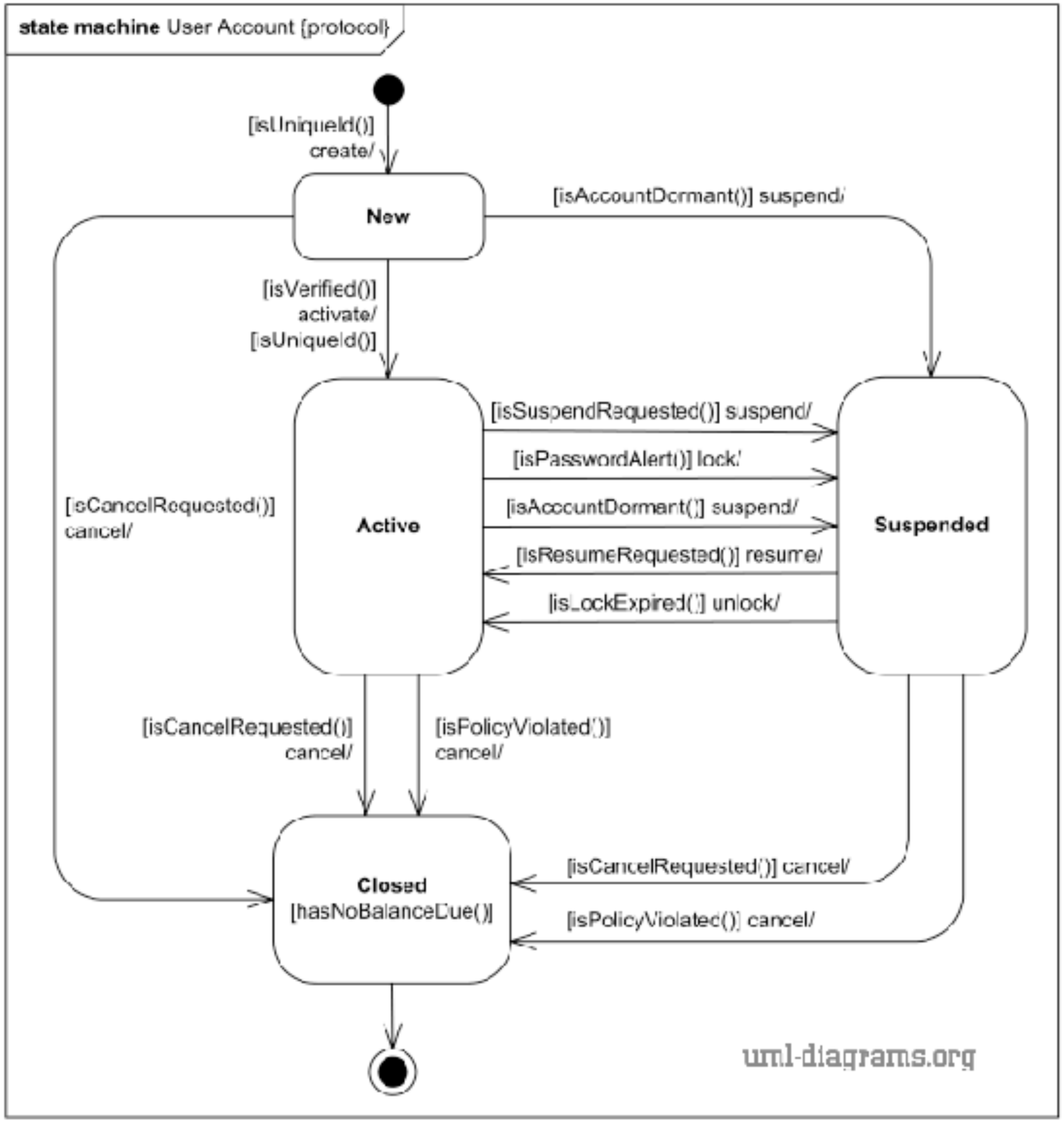
TypeStates - Queue

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

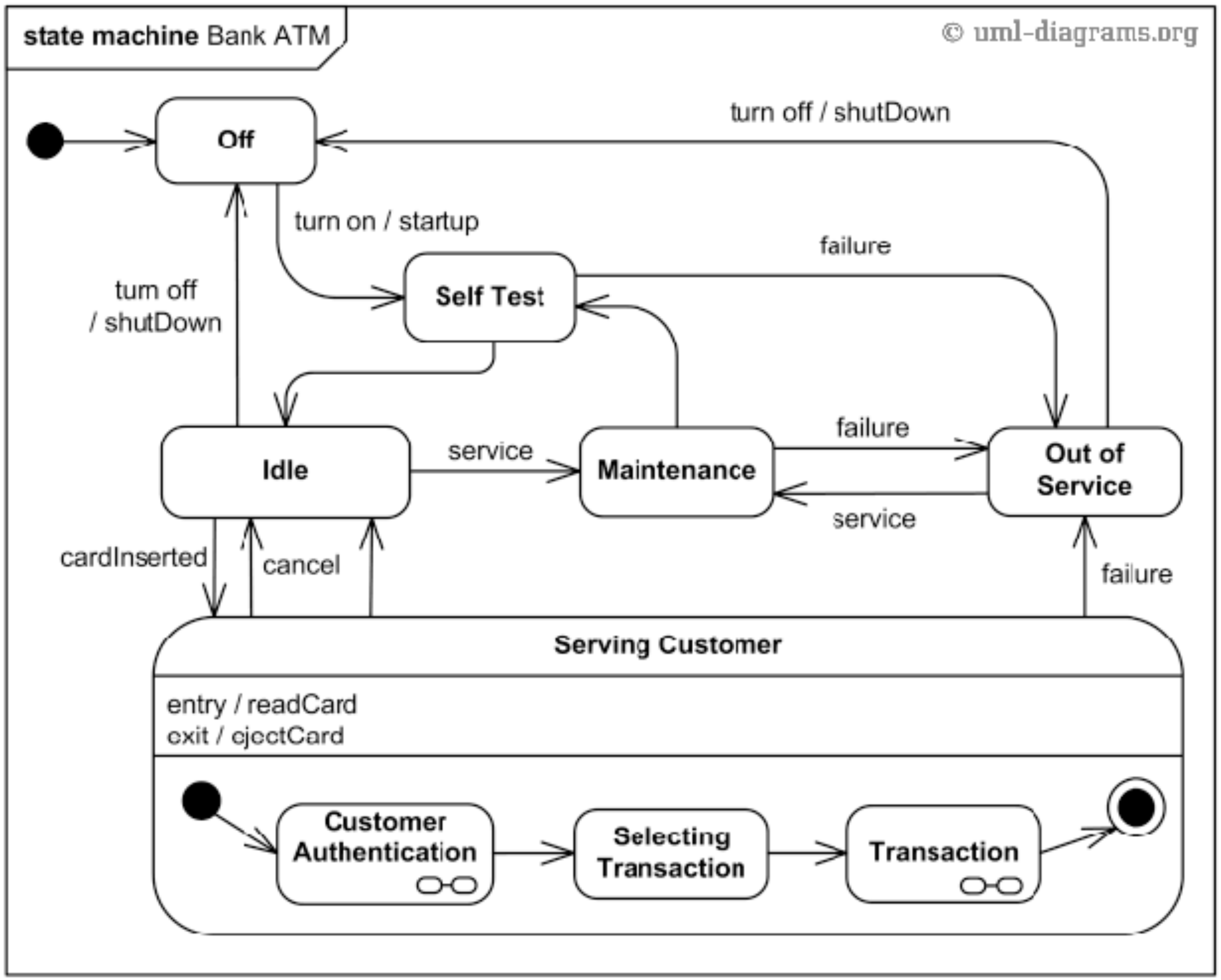
```
method Main()
{
    var q:Queue := new Queue(4);
    var r:int;

    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



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.

Construction and Verification of Software

2017 - 2018

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lab Assignment 4 - ADTs & TypeStates

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

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)

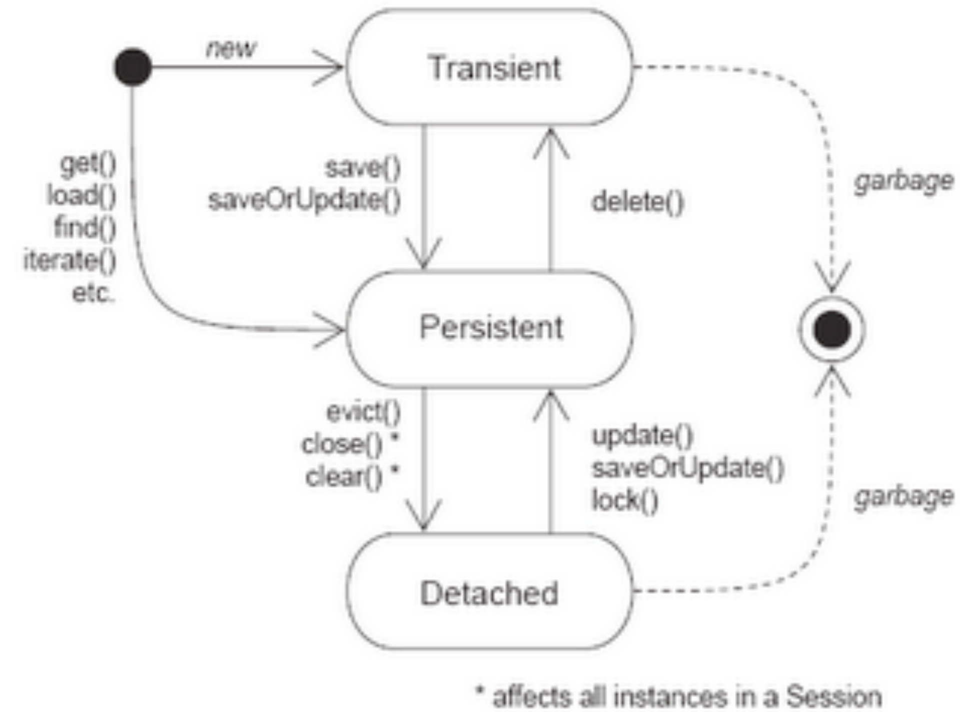
  The result of function find should be represented with type

  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:



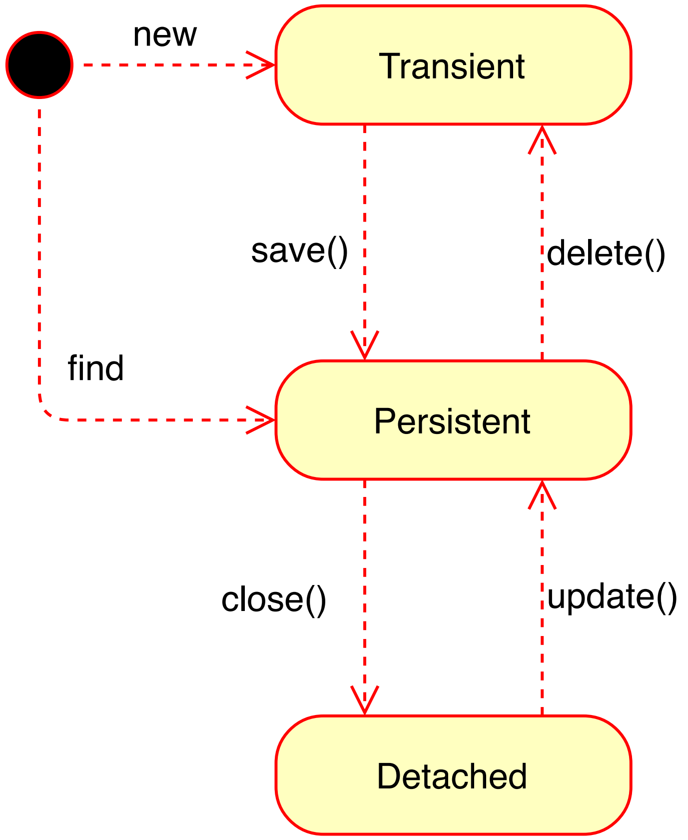
You must research about JPA states and implement two example classes that simulate such behaviour.

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 d
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).