Chapter 4.1 - Mapping UML Models to Code

Walter F. Tichy

Literature

- This lecture is based on section 10.4.2 from

  B. Bruegge, A.H. Dutoit, **Object-Oriented Software Engineering: Using UML, Patterns and Java,** Pearson Prentice Hall, 2004.

- **Read!**

Chapter 4.1.1 - Mapping Class Diagrams

Walter F. Tichy

Mapping of classes: OO languages

- For OO languages, each UML class is mapped to a class in the programming language (including attributes and methods).

```
class C { /* attributes */
           /* methods */ };
```

Mapping of classes: Non-OO languages (1)

- If no OO language is available, a class is mapped to a compound (record, structure); but this contains only the attributes. For the C language

  ```
  struct C { int a1; /* attributes */ };
  struct C c1, c2, c3; /* Instances */
  ```

- The access to the attribute `a1` of the instance `c3` happens as follows: `c3.a1`

Mapping of classes: Non-OO languages (2)

- Alternatively with type definition in C:

  ```
  typedef struct { int a1; /* attributes */ } C;
  C c1, c2, c3; /* instances */
  ```

- Alternatively, the memory for an instance can also be requested at runtime:

  ```
  C * c4; /* C* denotes a reference to an object of type C */
  c4 = (C*) malloc(sizeof(C));
  ```

- `malloc()` returns `void*` (an untyped pointer), so don't forget the type conversion (*cast*)!

- For references, the attribute access is done with operator `->` :
  `c4->a1`

Mapping of classes: Non-OO languages (3)

- Methods are mapped to subroutines that contain the compound type as an additional reference parameter. Method

  ```
  m(parameter) { ...this.attribute... }
  ```

  becomes freestanding function

  ```
  m(C * object, parameter) { ...object->attribute... }
  ```

- The call `c4.m(parameter)` becomes `m(c4, parameter)`

- Inheritance is simulated by adding the attributes of the superclass(es) to the composite.

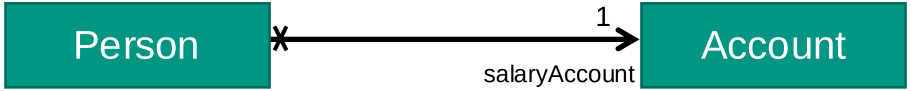
- Polymorphism can be simulated, if necessary, using function pointers and type conversion of the first parameter. But better to use C++ for that.

Association mapping

- Even OO languages do not provide associations, only references. The latter are used to implement the different types of associations (uni-/bi-directional, multiplicities).

Unidirectional one-to-one association

- For navigating from one object ot another, use an instance variable that contains a reference to the other class.

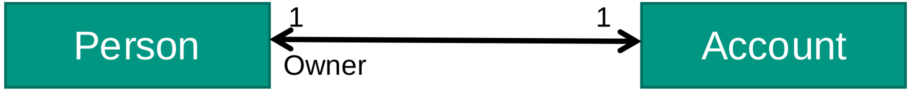


```
class Person {
  private Account salaryAccount;
  public Person() {
    salaryAccount = new Account();
  }
  public Account getSalaryAccount() {
    return salaryAccount;
  }
}
```

Privatization of `salary account` and associated access method prevents accidental changing of association.

Bidirectional one-to-one association

- In both classes, use an instance variable that contains a reference to the other.



- Attention: real 1:1 relationship means mutual dependence!

Bidirectional one-to-one association

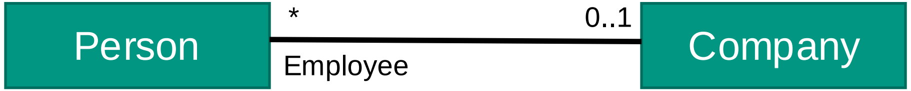
```
class Person {
  private Account salaryAccount;
  public Person() {
    salaryAccount = new Account(this);
  }
  public Account getSalaryAccount() {
    return salaryAccount;
  }
}
```

Here solution for initial consistency: make sure that no `null` values can occur during initialization

```
class Account {
  private Person owner;
  public Account(Person owner) {
    this.owner = owner;
  }
  public Person getOwner() {
    return owner;
  }
}
```

1:N associations

- Multiple references must be expressed by set-valued instance variables.



1:N associations

```
class Person {
  private Company firm;
  public Person() {}
  public void setCompany(Company f) {
    if (firm == f) return;
    firm = f;
    firm.hire(this);
  }
  public Company getCompany() { return firm; }
}

class Company {
  private Collection<Person> employees = new ArrayList<Person>();
  public Company() {}
  public void hire(Person p) {
    if (employees.contains(p)) return;
    employees.add(p);
    p.setCompany(this);
  }
}
```

Prevent infinite loop

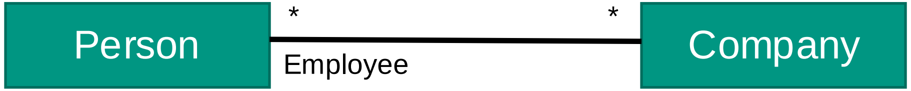
Add reference

This code has an error! Which one?

The curse of the dangling reference!

M:N association

- For N:M associations, two set-valued attributes must be used.



```
class Person {
  private Collection<company> employers;
  ...
}
class Company {
  private Collection<person> employees;
  ...
}
```

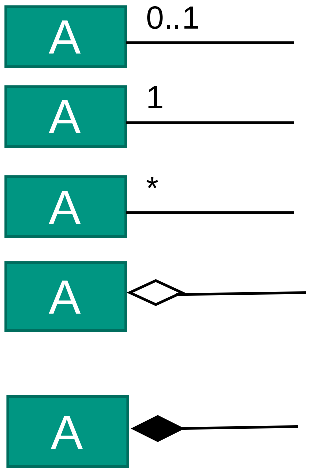
M:N association

```
class Person {
  ...
  public void addEmployer(Company f) {
    if (employers.contains(f)) return;
    employers.add(f);
    f.hire(this);
  }
  ...
}

class Company {
  ...
  public void hire(Person p) {
    if (employees.contains(p)) return;
    employees.add(p);
    p.addEmployer(this);
  }
  ...
}
```

Prevent infinite loop

Summary



Instance variable of **type A** on the opposite side

Instance variable of **type A** on the opposite side
+ code that ensures that the reference is never `null`

instance variable of **type multiple<A>** on the opposite side

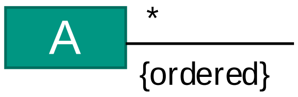
Instance variable of **type multiple<B>** in **A**, or, for fixed output degree of class A, fixed number of instance variables of **type B** in **A.**

Like ◇

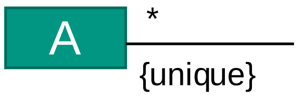
Basically, for association manipulating code:

- View associations as a "whole”
- Always adjust both sides
- "Think transactional“
- Don´t forget synchronization in concurrent programs (omitted here!)

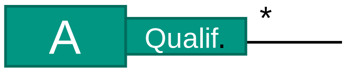
Special cases



Instance variable of **type List<A>** (with order) at the opposite side, e.g., ArrayList<A> or Vector<A>.

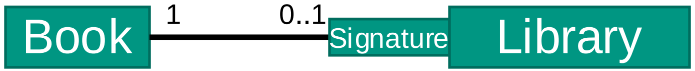


Instance variable of **type Set<A>** at the opposite side, e.g. HashSet<A> or TreeSet<A>.



Instance variable of **type Map<Qualifier, B>** in **A,** e.g., HashMap<Qualifier, B>.
+ Methods for access via the qualifier

Qualified association

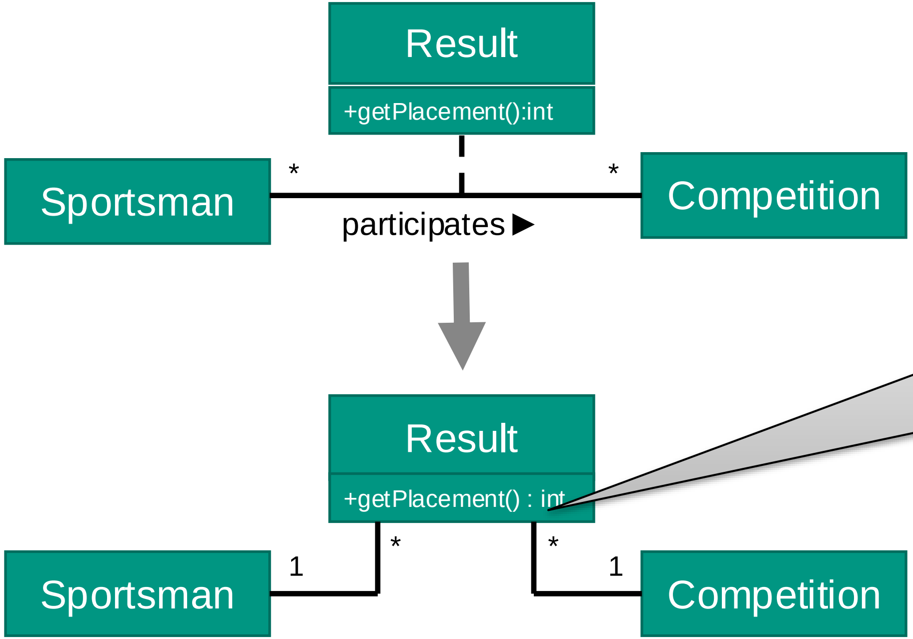


```
class Library {
  private Map<String, Book> books;
  public void addBook(String signature, Book b) {
    if (! books.containsKey(signature)) {
      books.put(signature,b);
      b.setLibrary(signature,this);
    }
  }
  public book getBook(String signature) {
    return books.get(signature);
  }
  ...
}
```

Access via qualifier

Association classes

- Realization by model transformation:



Code must ensure that the realization of the *two associations* behaves as if it were an *association* with additional attributes

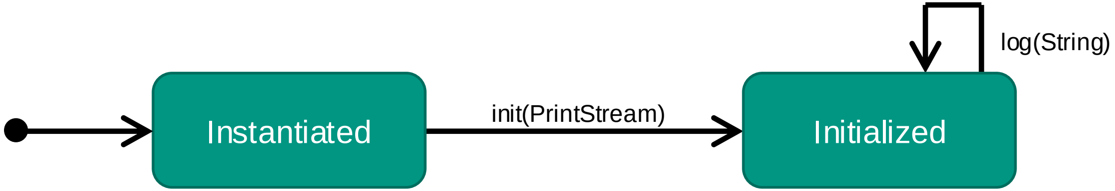
Chapter 4.1.2 - Mapping and implementing state machines

Walter F. Tichy

Storage of the state of an object

- **Implicit** storage
  - The state of the object can be "calculated" from the attribute values of an instance
  - No dedicated instance variables needed, but the state must be recalculated each time
  - State transition function is implicit
- **Explicit** storage
  - The state of an object is stored in dedicated instance variables and can therefore be easily read and reset
  - The state transition function must also be explicitly specified

Example of implicit storage of the state

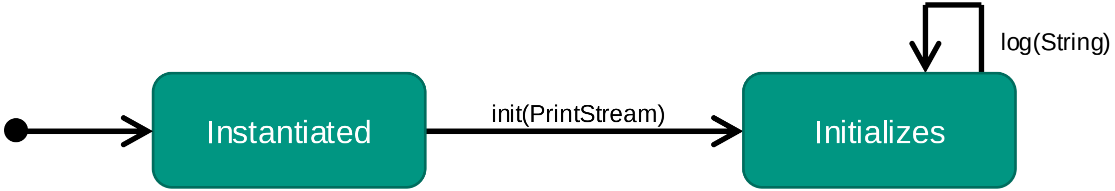


```
public class Logger {
   private PrintStream log;
   public void init(PrintStream dst) {
      log = dst;
   }
   public void log(String msg) {
      if (log == null) throw new IllegalStateException();
      log.append(msg);
   }
}
```

"Calculation" of the current state from the attribute values.

from `java.lang`

Example of explicit storage of the state



```
public class Logger {
   private enum state { instantiated, initialized };
   private state state = state.instantiated;
   private PrintStream log;
   public void init(PrintStream dst) {
      log = dst;
      state = state.initialized;
   }
   public void log(String msg) {
      if (state != state.initialized)
         throw new IllegalStateException();
      log.append(msg);
   }
}
```

Explicit storage of the state

State can be read out directly

Comparison implicit/explicit storage of the state

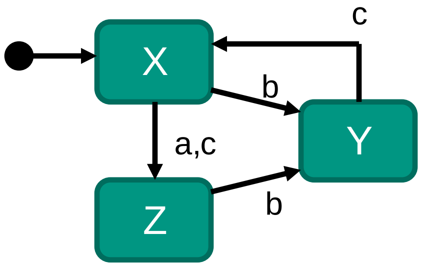
- Implicit storage saves memory, explicit (potentially) saves computation time.
- Implicit storage is potentially more complicated (trickier), explicit storage needs more space.
- Explicit storage is always obvious and therefore more likely to be considered when changes are made.
- Implicit storage is not always possible, but explicit storage is.

Alternatives of the implementation of explicit storage of the states

- Embedded
  - Each method "knows" the complete machine
    - It performs its task context-sensitively (= according to the current state) and
    - Performs the state transitions itself
    - Implemented as one large case distinction per method (switch- or if-statements).
  - Advantage: more compact, faster
- Outsourced
  - The method runs in the current state, which is an object.
  - The code for reacting and changing the states is located in dedicated (possibly automatically generated) classes
  - The branches of case discrimination are distributed among methods with the same name (strategy pattern)
  - Advantage: more flexible, clearer for complex machines

Example of embedded explicit storage

```
public class XYZ {
    private enum State{ X, Y, Z };
    private State state = State.X;
    public void a() {
        if (state == State.Y)
            throw new IllegalStateException();
        if (state == State.Z)
            throw new IllegalStateException();
        // ... do something ...
        state = State.Z;
    }
    public void b() {
        if (state == State.Y) throw new IllegalStateException();
        // ... do something ...
        state = State.Y;
    }
    public void c() {
        if (state == state.Z) throw new IllegalStateException();
        // ... do something ...
        if (state == state.X) state = State.Z;
        if (state == state.Y) state = State.X;
    }
}
```



The method a() must not be called in state Y or Z

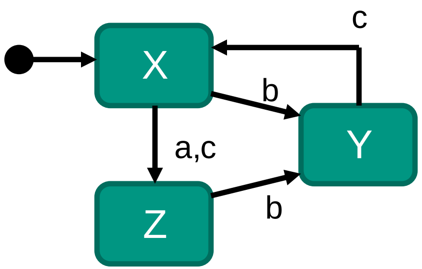
Next state: Z

What if something is supposed to happen here that is dependent on the current state?

The next state depends on the previous one

So, the "do s.t." part has to go into the `if` blocks too!

Example for embedded explicit storage (this time neat) (alternatively also with switch)



```
public void a() {
  if (state==State.X) {
    // ... do something ...
    state = State.Z; return;
  }
  if (state==State.Y)
    throw new IllegalStateException();
  if (state==State.Z)
    throw new IllegalStateException();
}
public void b() {
  if (state==State.X) {
    // ... do something ...
    state = State.Y; return;
  }
  if (state==State.Y)
    throw new IllegalStateException();
  if (state==State.Z) {
    // ... do something else ...
    state = State.Y; return;
  }
}
```

```
public void c() {
    if (state==State.X) {
      // ... do something ...
      State = State.Z; return;
    }
    if (state==State.Y) {
      // ... do something else ...
      state = State.X; return;
    }
    if (state==State.Z)
      throw new IllegalStateException();
}
```

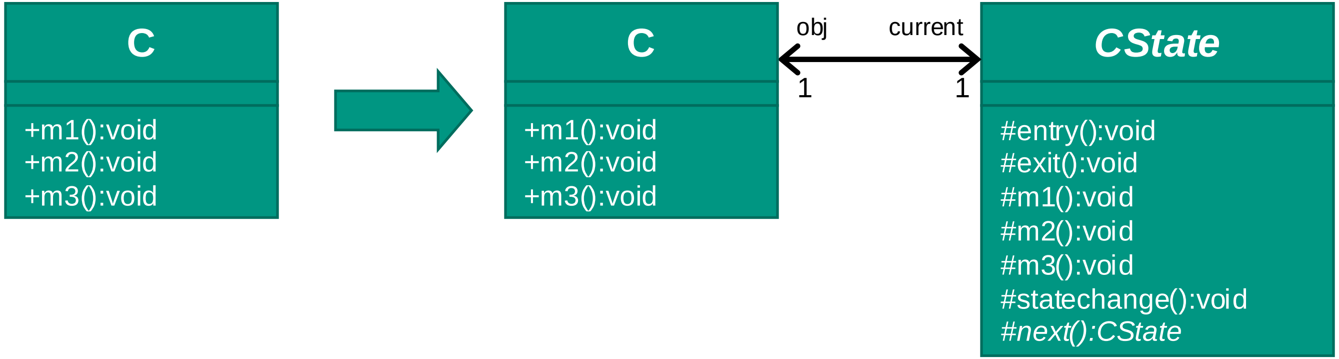
... and the `entry` and `exit actions` actually have to go in there everywhere too!

Outsourced explicit storage (*State Pattern*)

- Idea
  - The actual object does not know what exactly to do in which state.
  - It knows only its state
  - And delegates what to do when a message arrives to the state in question.
- Advantage:
  - The context sensitivity (= state dependency) of the methods no longer needs to be explicitly managed, instead dynamic polymorphism is used
  - The implementation work is divided among different classes (= different files in Java)
    - Better parallelization of the implementation work
    - "separation of concerns"

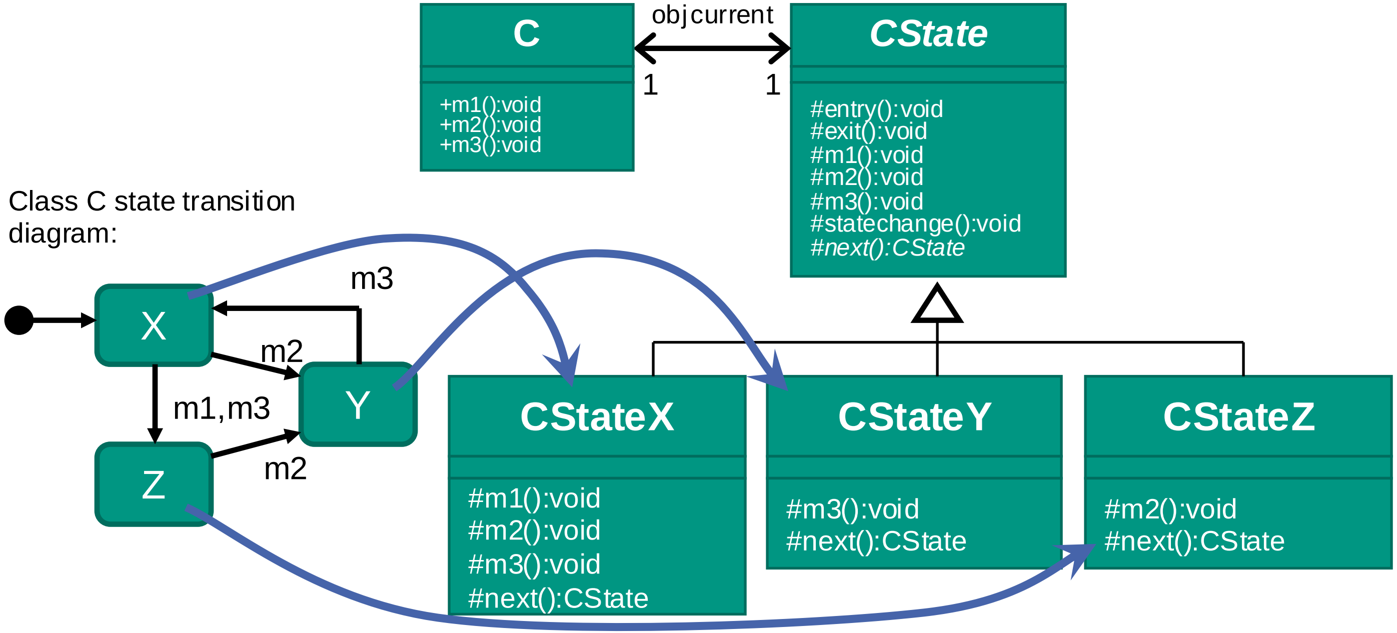
Procedure

- For each class `C` for whose behavior a finite automaton is to be implemented, create an abstract superclass `CState` which stands for all states of `C`:



Procedure

- Each state that the class C can assume is represented by its own type



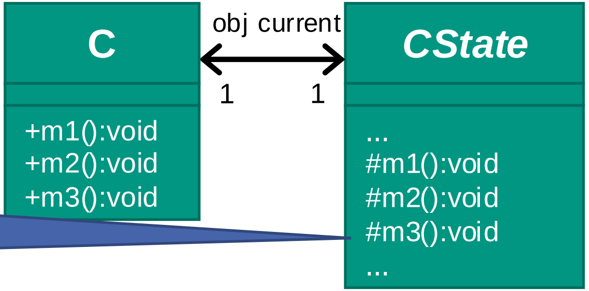
Procedure: Delegation

- The implementation of the methods `m1`, `m2` and `m3` in `C` is delegated:

```
class C {
   ...
   protected CState current;
   ...
   public void m1() {
      current.m1(); // delegate
      // Change of state
      // still missing
   }
   ...
}
```

Call only works if protection is also package-private (e.g. as in Java) and C is in the same package as CState

Alternative: States are inner classes of C



In UML, protected elements are accessible only from the class itself and subclasses

Implementation of the methods of the abstract class `CState`

- Default behavior: "Do nothing on `entry` and `exit event".`
  → a concrete state needs to override implementations only where it wants to.

```
protected void entry() { //empty }
protected void exit()  { //empty }
```

(Question: is this like `null-object ?`)

Implementation of the methods of the abstract class `CState`

- Default behavior: "Calling this method is not allowed in this state".
  → a concrete state needs to specify an implementation only when a call to the method is allowed in the current state (override the default methods in this case)

```
protected void m1() {
    throw new IllegalStateException();
}
protected void m2() {
    throw new IllegalStateException();
}
protected void m3() {
    throw new IllegalStateException();
}
```

Implementation of the methods of the abstract class `CState`

- The abstract method `next()` is an *insertion method* or *hook* that specifies which state would be the next state. It does not make a state change and as an insertion method it is not called from outside!
- The method is abstract, so each state must specify its own implementation.

```
protected abstract CState next();
```

Implementation of the methods of the abstract class CState

- The method changeState() is a template method that performs a state change if necessary. It must be called after each invocation of a method of C:

```
protected final void changeState() {
    CState newState = next();
    this.exit();
    newState.entry();
    obj.current = newState;
}
```

Calling the hook method

Procedure

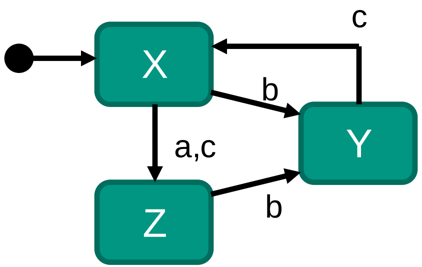
- The implementation of the methods `m1`, `m2` and `m3` in `C` looks like this:

```
class C {
   ...
   protected CState current;
   ...
   public void m1() {
      current.m1();
      current.changeState ();
   }
   ...
}
```

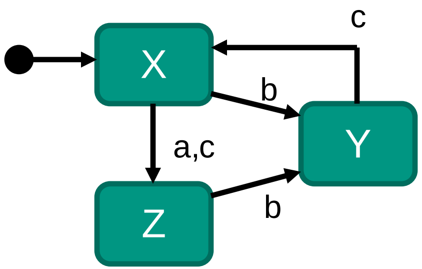
Example

- Let the automaton below be defined for the class `C`.
- Then

```
class C {
  protected CState current = new CStateX(this);
  public void a() {
    current.a();
    current.stateChange();
  }
  public void b() {
    current.b();
    current.stateChange();
  }
  public void c() {
    current.c();
    current.stateChange();
  }
}
```



Example



```
class CState {
  protected C obj; // get back to C
  protected CState(C c) { obj = c; }
  ... // as described
}
class CStateX extends CState {
  private CState fs; //future state
  protected CStateX(C c) { super(c); }
  protected void a() {
     ... // do something
     fs = new CStateZ(obj);
  }
  protected void b() {
     ... // do something
     fs = new CStateY(obj);
  }
  protected void c() {
     ... // do something
     fs = new CStateZ(obj);
  }
  protected CState next() {
     return fs;
  }
}
```

```
class CStateY extends CState {
   prot. CStateY(C c) { super(c); }
   protected void c() {
      ... // do something
   }
   protected CState next() {
      return new CStateX(obj);
   }
}
class CStateZ extends CState {
   prot. CStateZ(C c) { super(c); }
   protected void b() {
      ... // do something
   }
   protected CState next() {
      return new CStateY(obj);
   }
}
```

Here with **new;** instead, singletons or prototypes should be used.