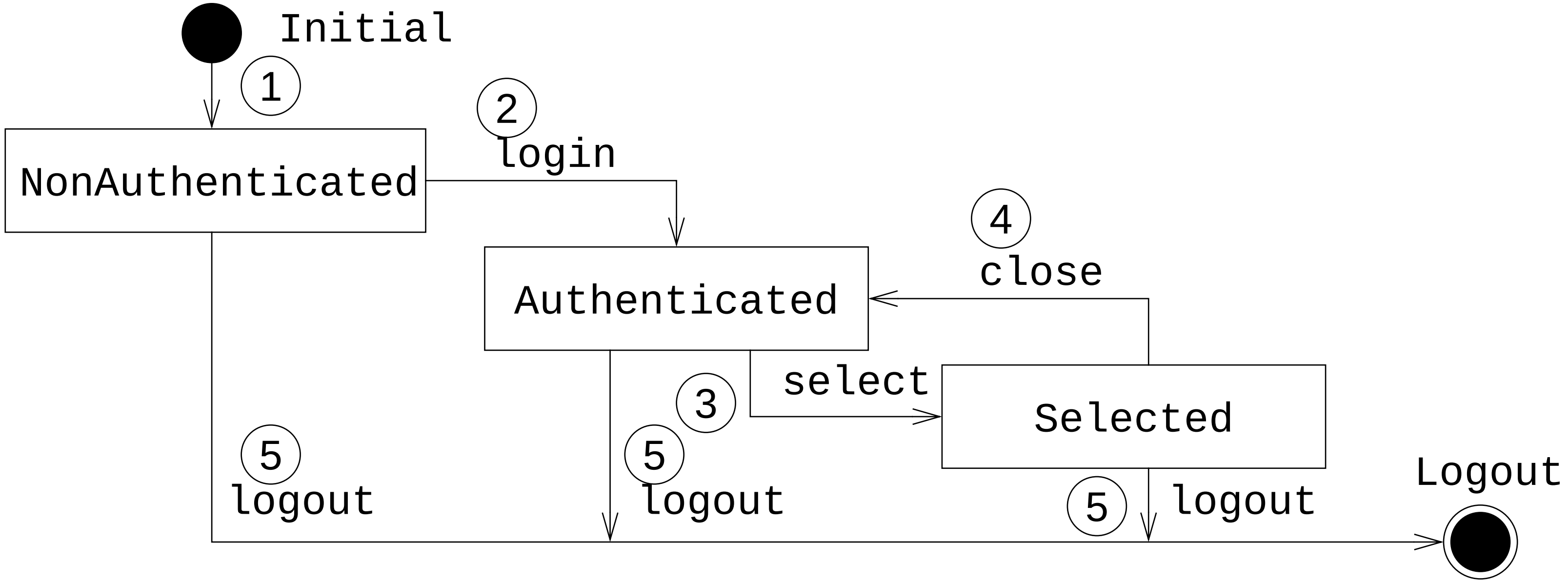
State

Intent

Implement behavior as a state machine. An object’s behavior will appear to as if it would change its class.

Motivation

Consider a simplified view of an IMAP session in an IMAP server:



The states:

- `Initial`: no session has been initialized.
- `NonAuthenticated`: session has been started, client not authenticated.
- `Authenticated`: client is authenticated.
- `Selected`: a mailbox has been chosen.
- `Logout`: the session has been terminated.

The transitions:

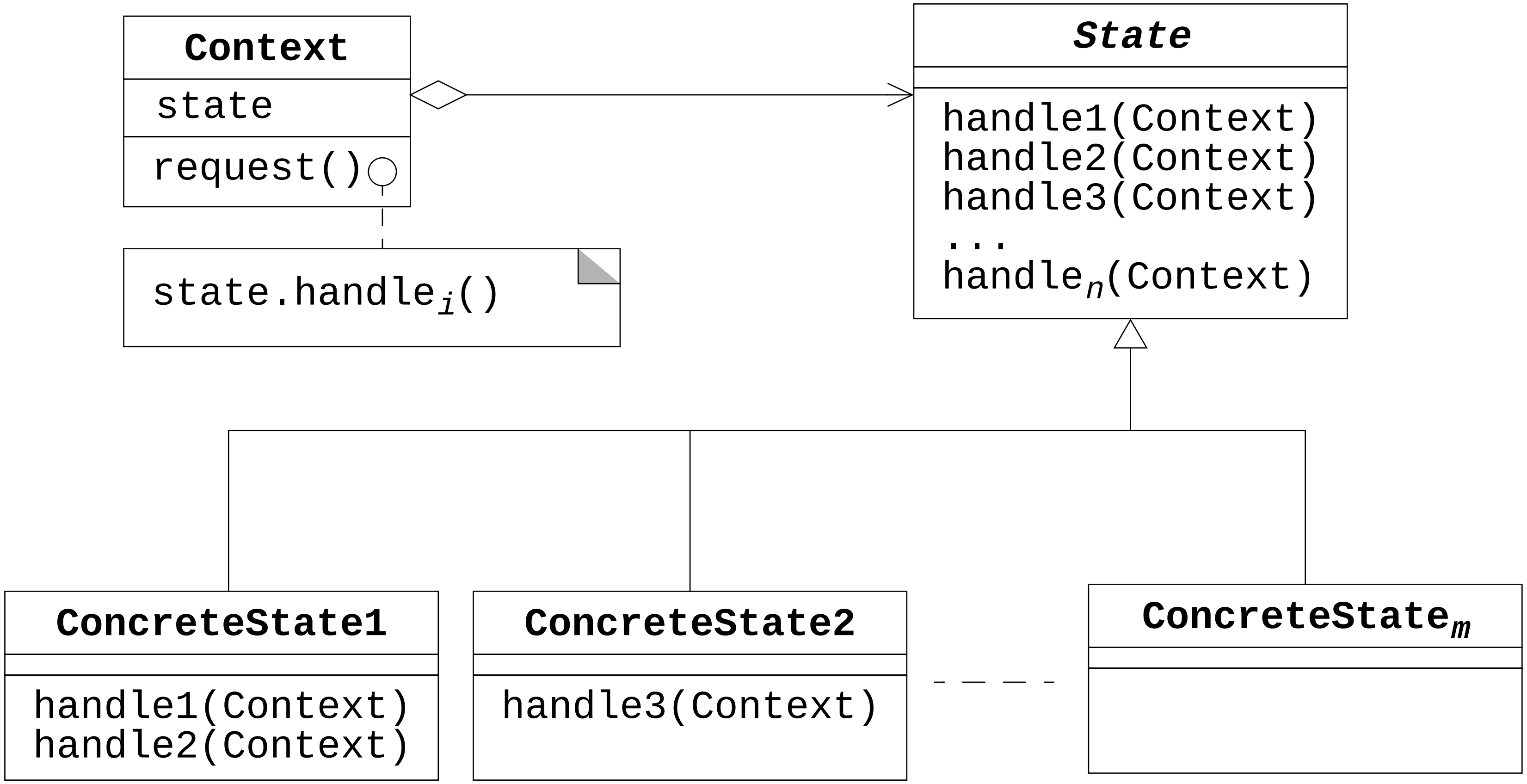
1. Session is established.
2. Successful login.
3. Successful selection of a mailbox.
4. Closing of a mailbox.
5. Logout, i.e., termination of the session.

Applicability

Use the State pattern:

- An object’s behavior must be changed as if it would belong to another class.
- The object’s methods have large, multi-part conditional statements that depend on the object’s state.

Structure


Context
state
request()
state.handle$_i$()
State
handle1(Context)
handle2(Context)
handle3(Context)
...
handle$_n$(Context)
ConcreteState1
handle1(Context)
handle2(Context)
ConcreteState2
handle3(Context)
ConcreteState$_m$

Participants

- **`Context`**:
  - defines the interface of interest to clients
  - maintains an instance of `ConcreteState`
  - remembers additional information regarding the current state such as input and/or output streams, etc.
- **`State`**:
  - defines the interface common to all `ConcreteState`$_i$ classes
  - provides default implementations for all `handle`$_i$ methods.
- **`ConceteState`**$_i$:
  - implements the behavior associated to this state by overwriting a subset of the `handle`$_i$ methods.

Collaborations

- Clients configure a `Context` object with a `State` object, which typically denotes the initial state.
- A `Context` object delegates state-specific requests to the current `State` object.
- Either the `Context` object or the `State` object can decide which state succeeds another.

Consequences

- State-specific behavior is localized in one class.
- The logic of state transitions is partitioned between the `State` subclasses.
- If `State` objects contain no instance variables then they can be shared.

Implementation

- State transitions: Should the decision made in the `Sate` objects or in `Context`? If the Sate objects determine the new state then `Context` must provide the appropriate interface. (Or, see Example section, this is can be done by `State` objects by returning the new `State` instance.)
- Alternative implementations exist: Table-based, sparse matrix-based.
- Life-time of `State` objects: Creation of `State` objects on demand? When should they be destroyed?
- Dynamic inheritance: The State pattern implements some kind of *dynamic inheritance*.[1]

[1] Which is absent in most common programming languages, except for example Self.

Sample Code

Let’s have a look to the state management of a IMAP server. Upon establishing a session, a `SessionContext` object is created and properly initialized:

```
public class SessionContext {
   private Reader r; private Writer w;
   private State state = new NonAuthenticated();
   public SessionContext(Reader r, Writer w) { ... }
   public void handleCommand() {
      IMAPCommand cmd = new Parser(r).parse();
      if (cmd instanceof IMAPLogin) {
         state = state.login(this, cmd);
      } else if (cmd instanceof IMAPSelect) {
         state = state.select(this, cmd);
      } else if (cmd instanceof IMAPClose) {
         state = state.close(this, cmd);
      } else if (cmd instanceof IMAPLogout) {
         state = state.logout(this, cmd);
      } else {
         state = state.illegalCommand(this, cmd);
}  }  }
```

Notes:

- Given some `Reader`, the input is parsed, and a corresponding `IMAPCommand` object is returned.
- Which method to apply on a `State` object is determined dynamically using Java's `instanceof` operator. Use other means, or let the client decide which method to perform, if your programming language does not support a kind of `instanceof` operator.
- The new kind of `State` object is determined by the method applied to the current `State` object.
- Auxiliary methods are not shown.

The abstract `State` class looks like:

```
public abstract class State {
   public State login(SessionContext ctx, IMAPCommand cmd) {
      return this;
   }
   public State select(SessionContext ctx, IMAPCommand cmd) {
      return this;
   }
   public State close(SessionContext ctx, IMAPCommand cmd) {
      return this;
   }
   public State logout(SessionContext ctx, IMAPCommand cmd) {[1]
      return this;
   }

   public State illegalCommand(Context, IMAPCommand) {[2]
      // Perform some useful error handling here...
   }
```

[1] Could possibly handle all cases here, i.e., no overwriting necessary in subclasses.

[2] Perhaps handled already in class `SeesionContext`.

Each concrete subclass of `State` must overwrite some of the methods `login`, `select`, `close`, or `logout`. For example, class `NonAuthenticated` implements methods `login` and `logout`:

```
public class NonAuthenticated extends State {

   public State login(Context ctx, IMAPCommand cmd) {
      State s = this;
      if (verifyLogin(cmd))
         s = new Authenticated();
      return s;
   }

   public State logout(Context ctx, IMAPCommand cmd) {
      // perform cleaning ..
      return null;
   }
   protected boolean verifyLogin(IMAPCommand cmd) { ... }
}
```

Class Authenticated implements methods select and logout:

```
public class Authenticated extends State {

   public State select(Context ctx, IMAPCommand cmd) {
      State s = this;
      Mailbox mb = selectMailbox(cmd);
      if (mb != null) {
         ctx.setMailbox(mb);
         s = new Selected();
      }
      return s;
   }
   public State logout(Context ctx, IMAPCommand cmd) {
      // perform cleaning ..
      return null;
   }
   protected Mailbox selectMailbox(IMAPCommand cmd) { ... }
}
```

Class Selected implements methods select, close, and logout:

```
public class Selected extends State {
   public State select(Context ctx, IMAPCommand cmd) {
      State s = this;
      Mailbox mb = selectMailbox(cmd);
      if (mb != null) {
         ctx.setMailbox(mb);
         s = new Selected();
      }
      return s;
   }
   public State close(Context ctx, IMAPCommand cmd) {
      // Close selected mailbox, then:
      ctx.setMailbox(null);
      return new Authenticated();
   }
   public State logout(Context ctx, IMAPCommand cmd) {
      // perform cleaning ..
      return null;
   }
   protected Mailbox selectMailbox(IMAPCommand cmd) { ... }
}
```

Related Patterns

- If commands have to be parsed prior to the determination of the operation to be performed on the `State` object, use a parser generator-generated parser or the Parser pattern.
- Use the Flyweight pattern to share the `State` objects.
- If `State` objects are stateless then you may use the Singleton pattern.