# Software Engineering I (02161)

## Week 8

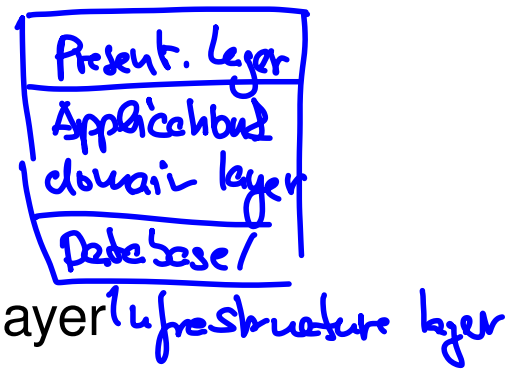
Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2017

# Recap

- Last week
  - Sequence diagrams
  - Centralized vs. Decentralized Control
  - How to implement associations from class diagrams
  - Layered architecture: basics
- This week
  - Version control
  - State machines
  - Layered architecture: Presentation layer
- Next week
  - Layered architecture: Persistency layer
  - Software development processes



# Contents

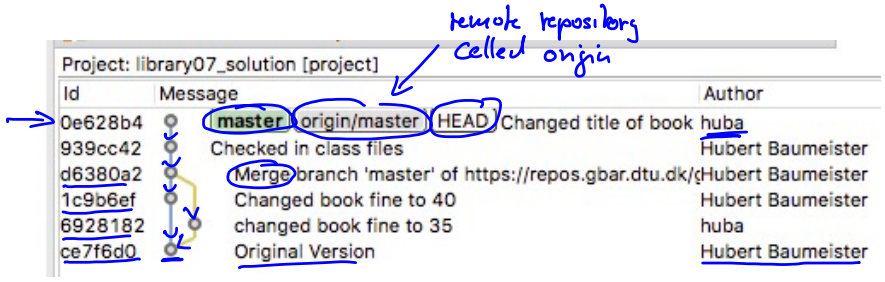
# What is version control?

Version Control

- Stores and manages snapshots of project files (e.g. .java files)
- Manages concurrent work on project files
- Various systems: Git, Concurrent Versions System (CVS), Subversion (SVN), Team Foundation Server (TFS) ...

# Git

- Developed by Linus Torvalds for use with Linux
- Set of ommand line tools, IDE support
- Set of files are collected into ”snapshots” called **commits**
- Commits have one or more parents and describe the difference to their parents
- Names of commits are SHA1 hashes of their contents usually identified by the first 7 characters in hex representation 63d281344071f3ae1054bca63f1117f76a3d5751 and 63d2813




- Branches: Two commits for the same parent
- Merging: Merging the changes of two commits into one

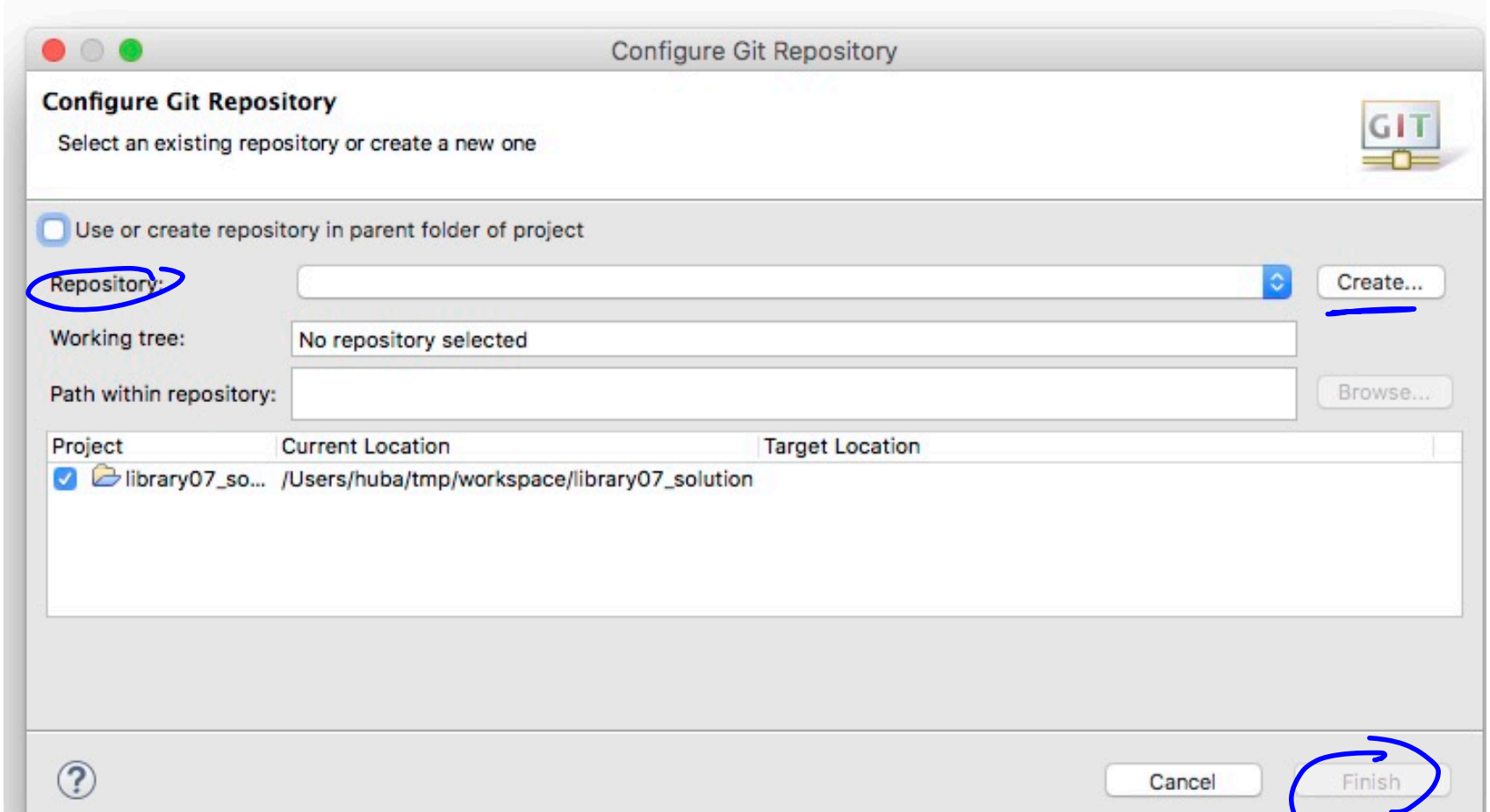
# Git

- Distributed repository
  - Commits stored in the local **repository**
  - Local repository can be synchronized with one or may remote repositories

  $\rightarrow$ **Push** (local $\rightarrow$ remote) and **Pull** (remote $\rightarrow$ local)

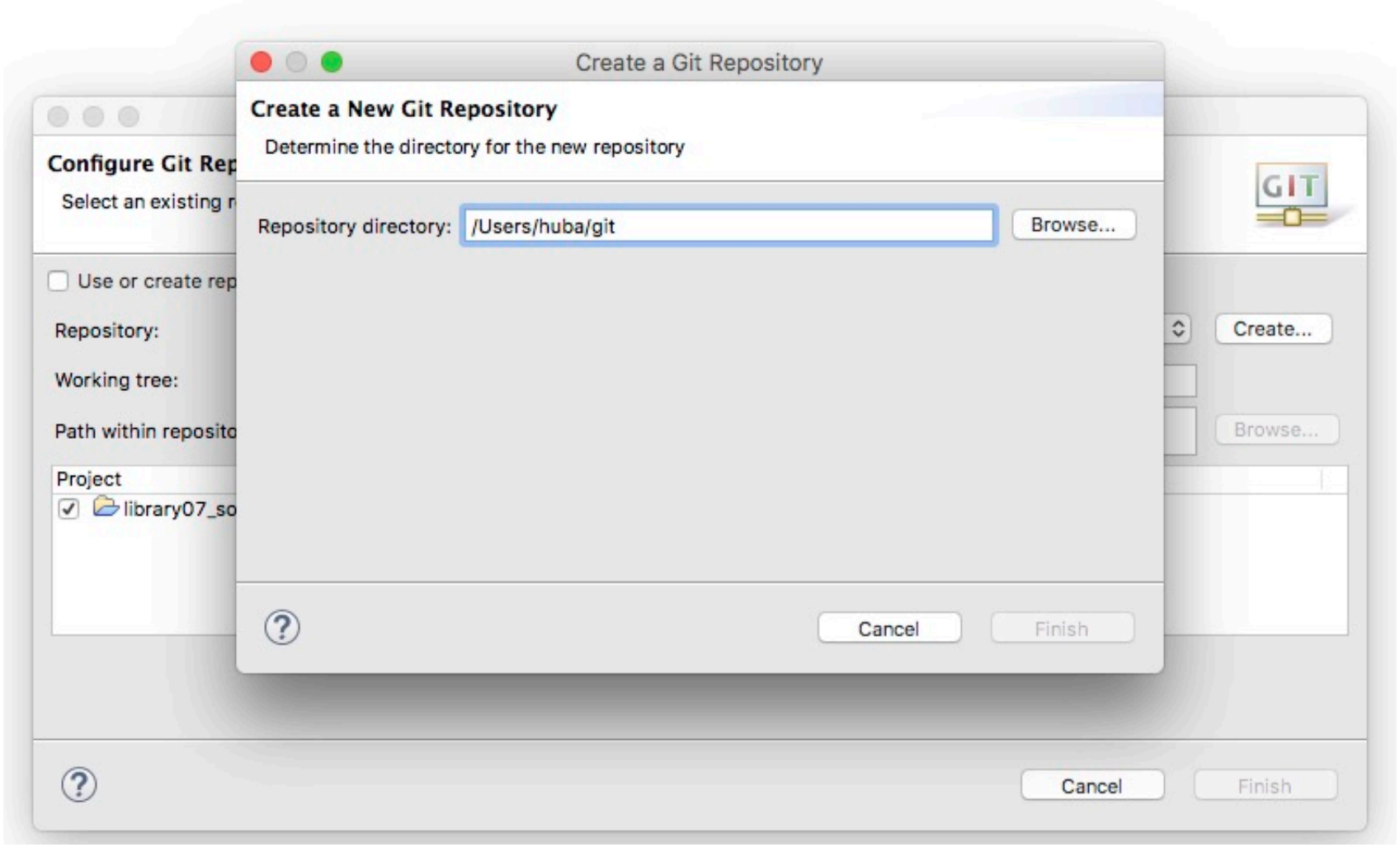
# Starting with a project

1. Create a central repository:
   `http://repos.gbar.dtu.dk`
2. Create an initial project with one of the team members in Eclipse
3. Create a local repository for the project: Use Team::Share Project



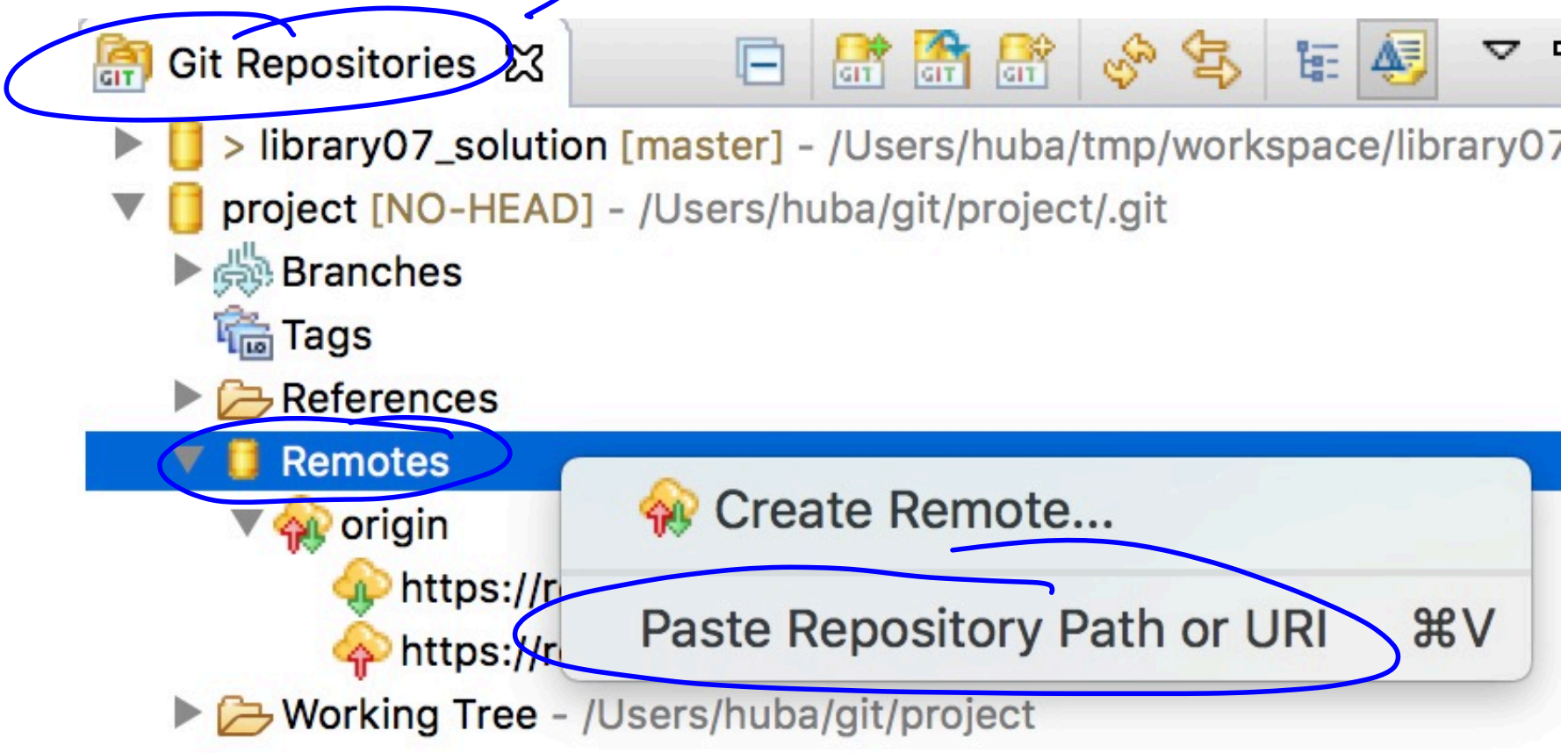
# Starting with a project

3 Create a local repository for the project: Use Team::Share Project



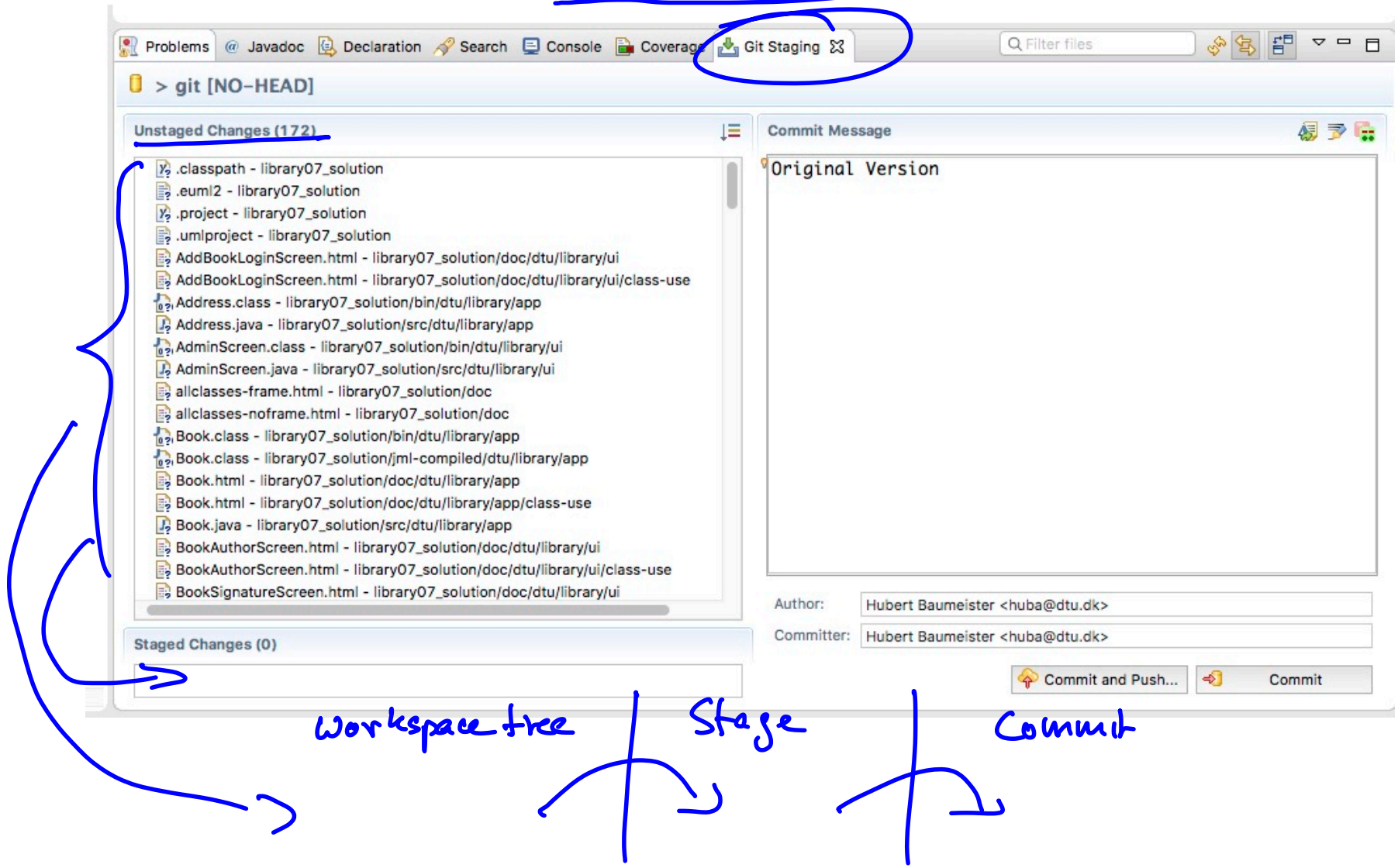
# Starting with a project

4 Attach the central repository as a remote repository to your local repository



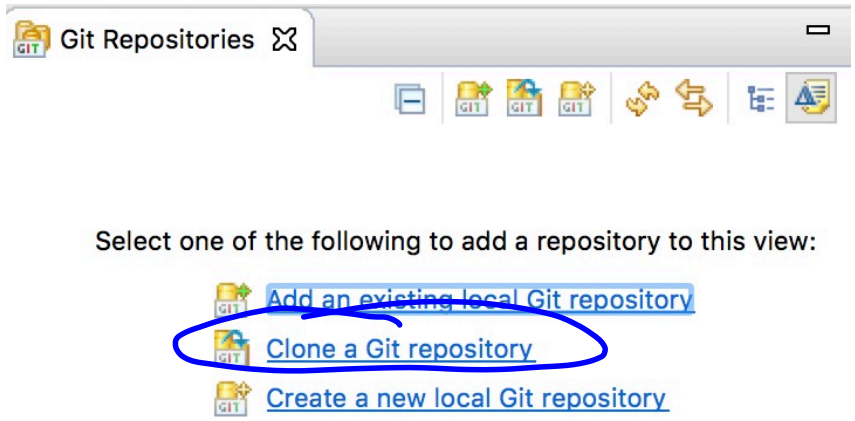
# Starting with a project

5 Stage, commit, and push the initial commit to the remote repository: Team:Push upstream / Push upstream master



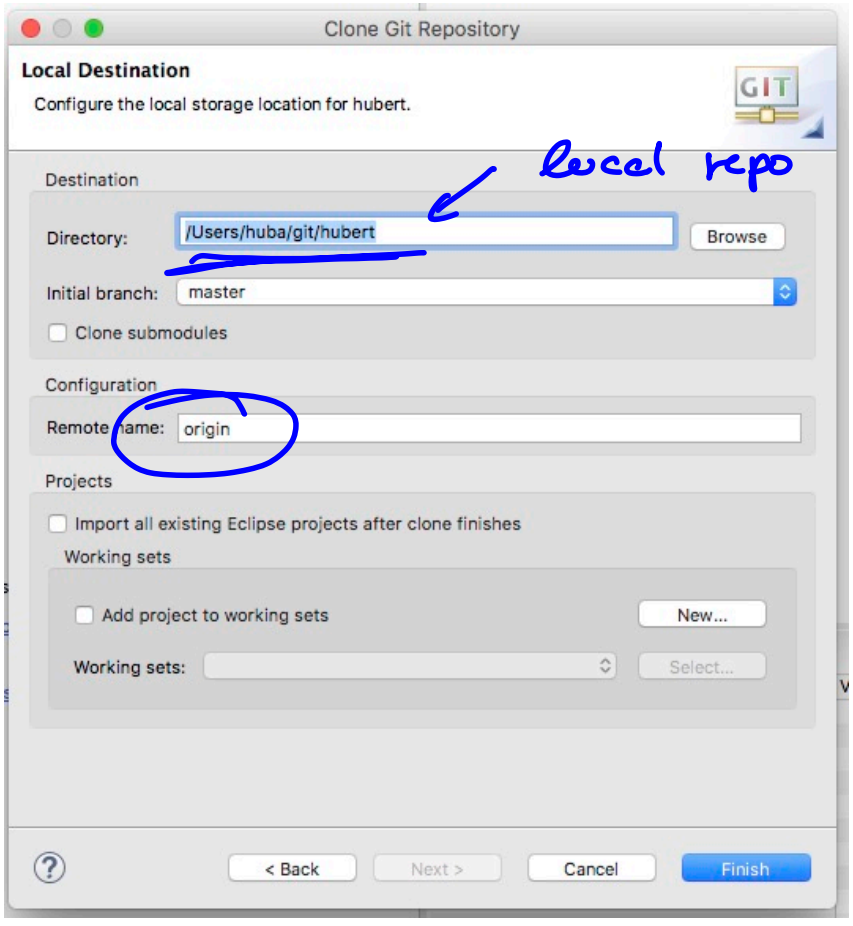
# Starting with a project

6 Other members: clone the repository from the central repository: Git repository view



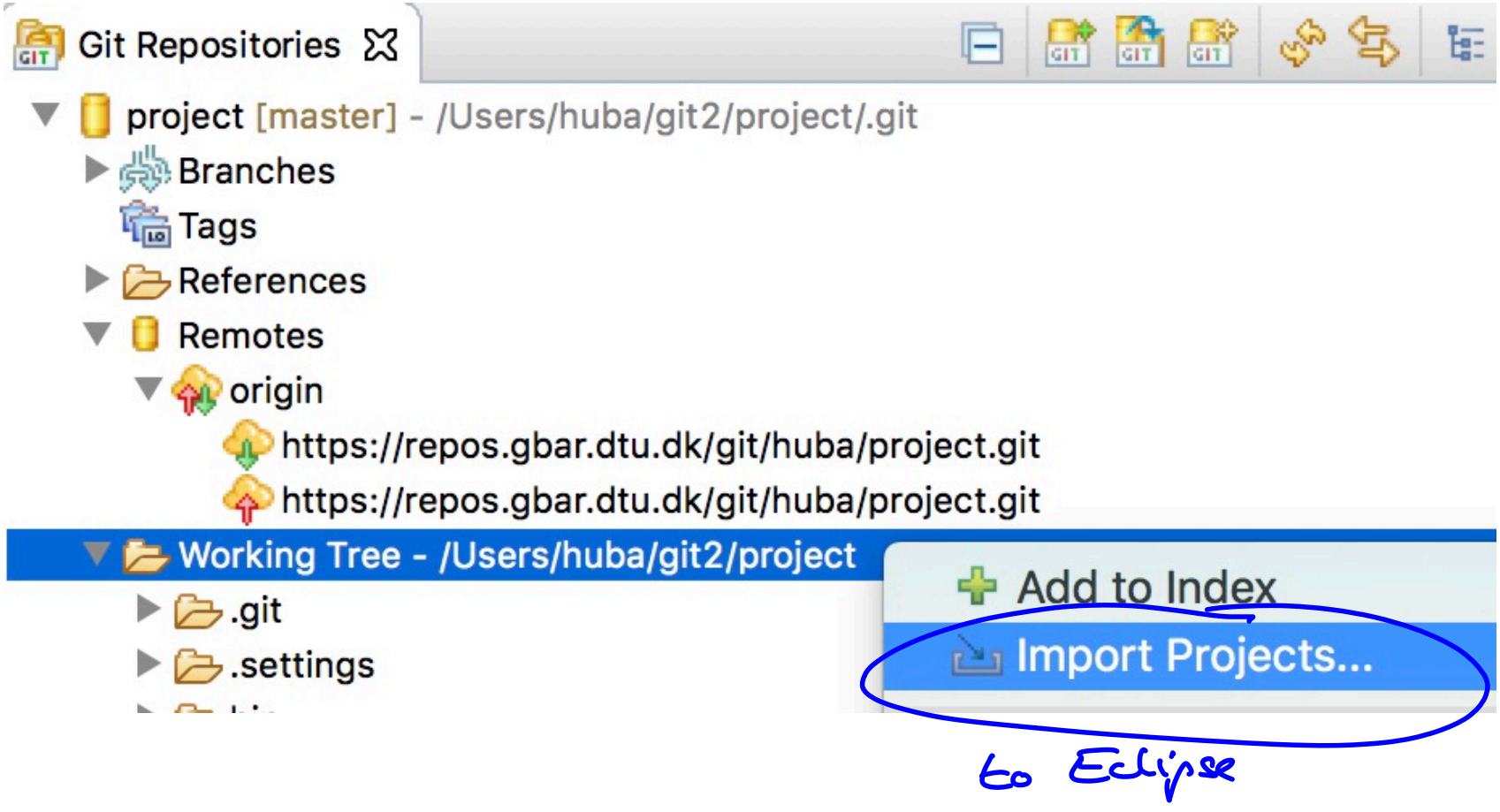
# Starting with a project

6 Other members: clone the repository from the central repository: Git repository view



# Starting with a project

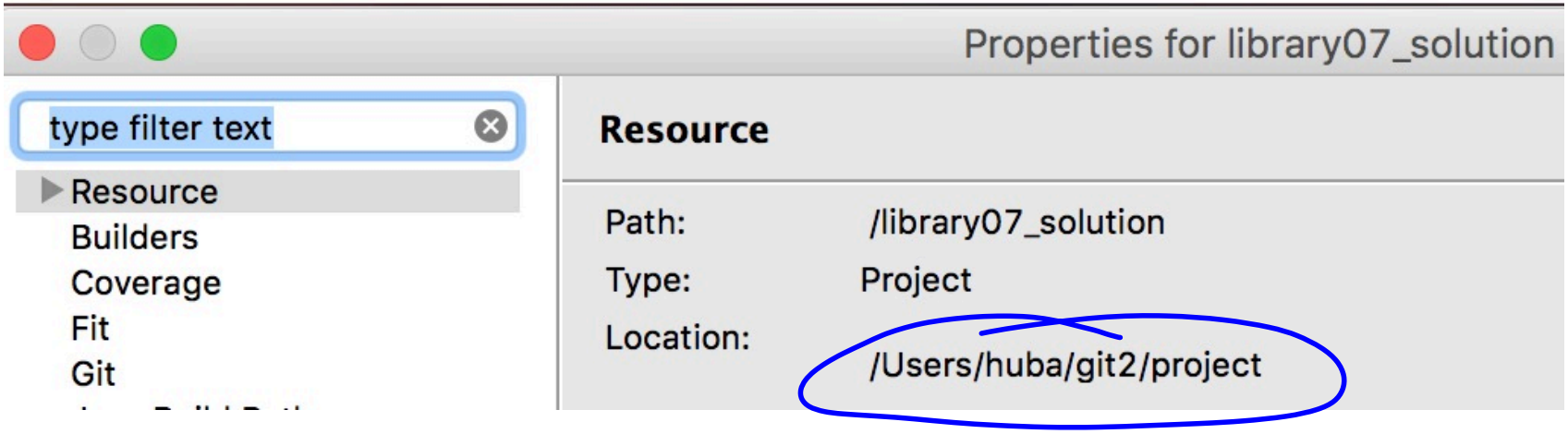
7 Other members: Import the Eclipse project: Git repository view



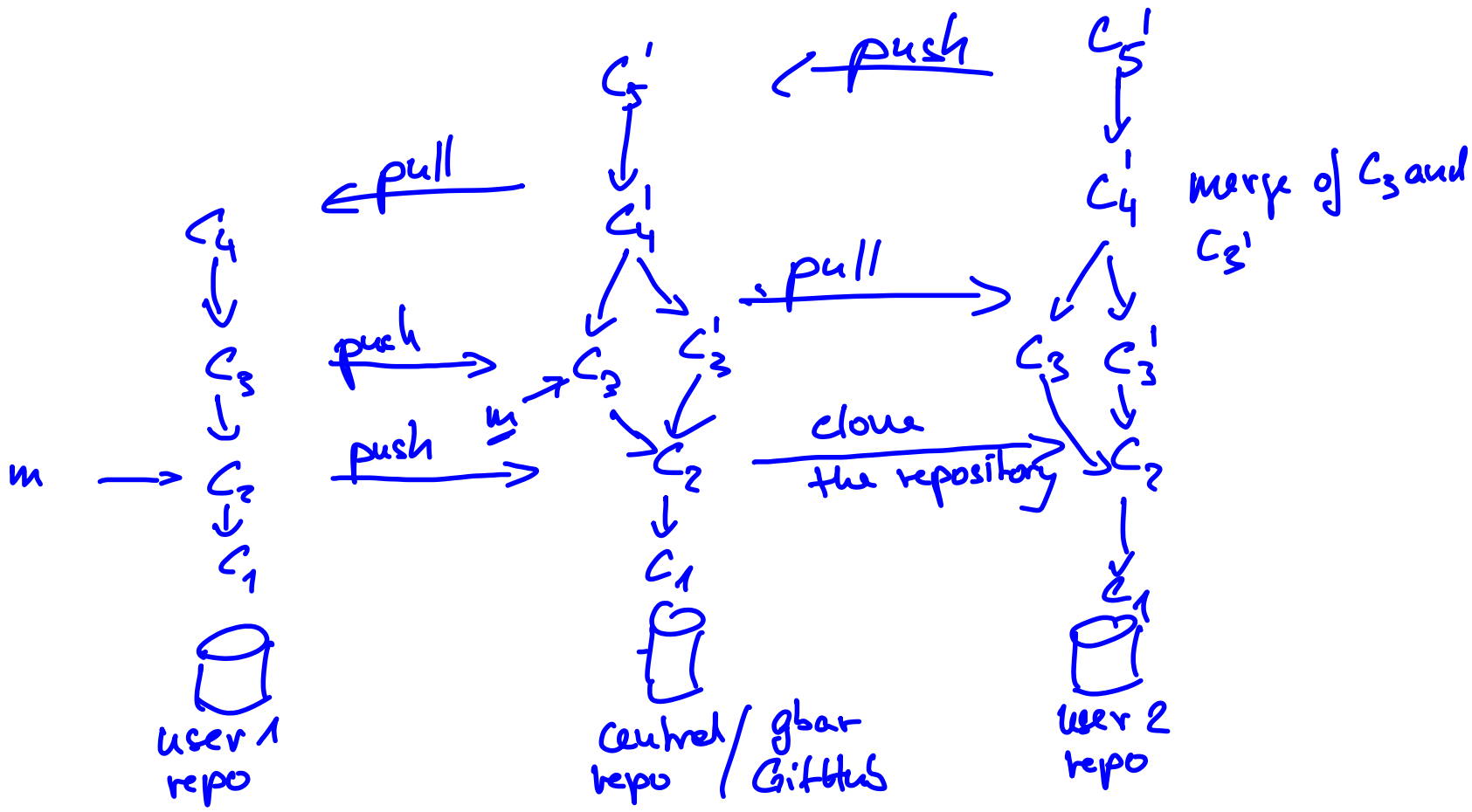
# Storage of the Eclipse project

- Option one: In the Eclipse workspace
- Option two: In a special Git repository directory

→ Use project properties to find out



# Working with Git



# Working with Git

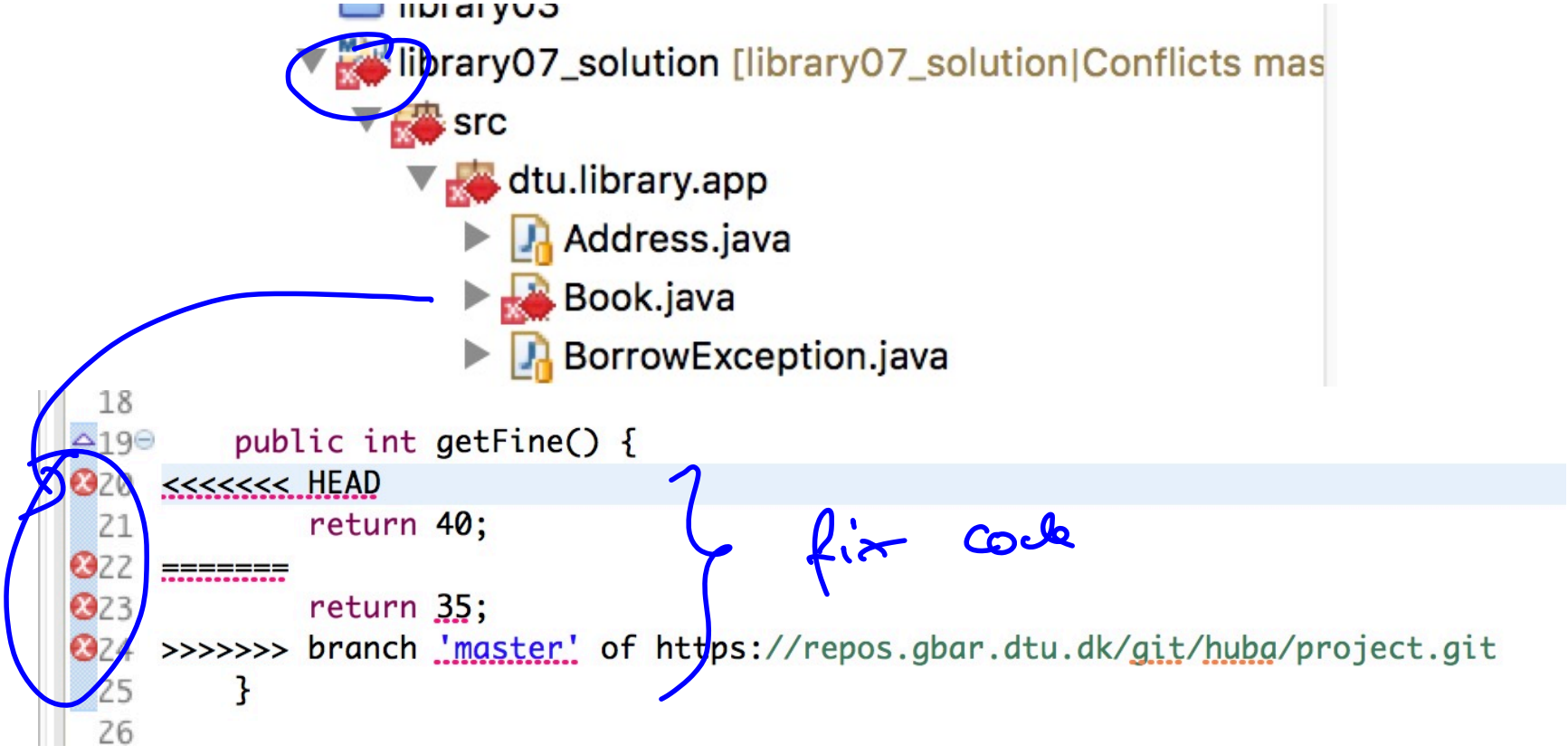
1. Pull the latest changes from the central repository
2. Work on a user story with commits to the local repository as necessary (Team::Commit)
3. Once the user story is done (all tests are green) stage and commit the result
4. Before pushing your commits first pull all commits done in the meantime by others from the central repository
   - $\rightarrow$ this will merge their commits with the local ones and create a new merged commit
5. Fix any merge conflicts until all tests are green again
6. push your final commit to the central repository

   Important: Never push a commit where the tests are failing

# When Pushing commits fail

- Pushing fails if someone else as pushed his commits before: No fast-forward merge possible
  1. pull from central repository
     - this automatically tries to merge the changes,
  2. compile: fix possible compilation errors
  3. run the tests: fix failing tests
  4. commit and push again

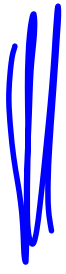
# Merge conflicts when pulling




- Git is in a merge state

1. Resolve conflicts
2. Stage your changes
3. Commit and push changes

# Git resources



- Git is more complex than shown: e.g. we didn't cover branching (not really needed for the project though)
- Git tutorial `https://www.sbf5.com/~cduan/technical/git/`
- Git Book: `https://git-scm.com/book/en/v2`

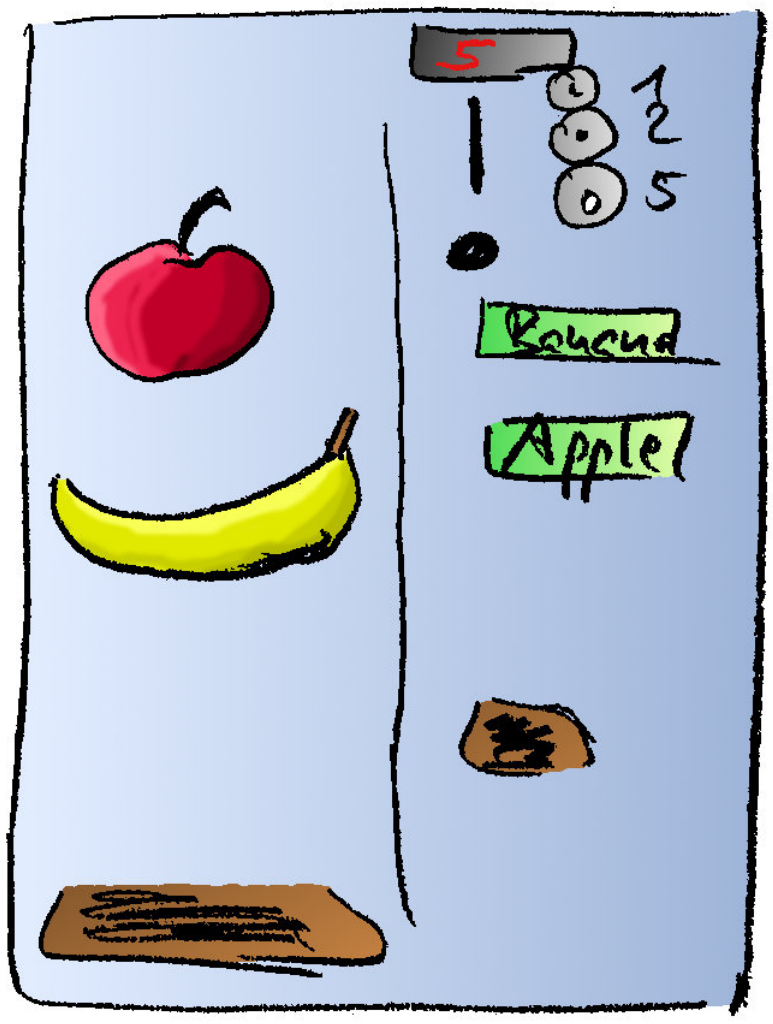
# Contents

# UML State Machines

- UML structure diagrams
  - e.g. class diagram
- UML behaviour diagrams
  - Activity diagrams:Focus is on activities
  - Sequence diagrams: Focus is message exchange between objects
  - State machine: Focus is on states and events
    - How does the system react when an event occurs?

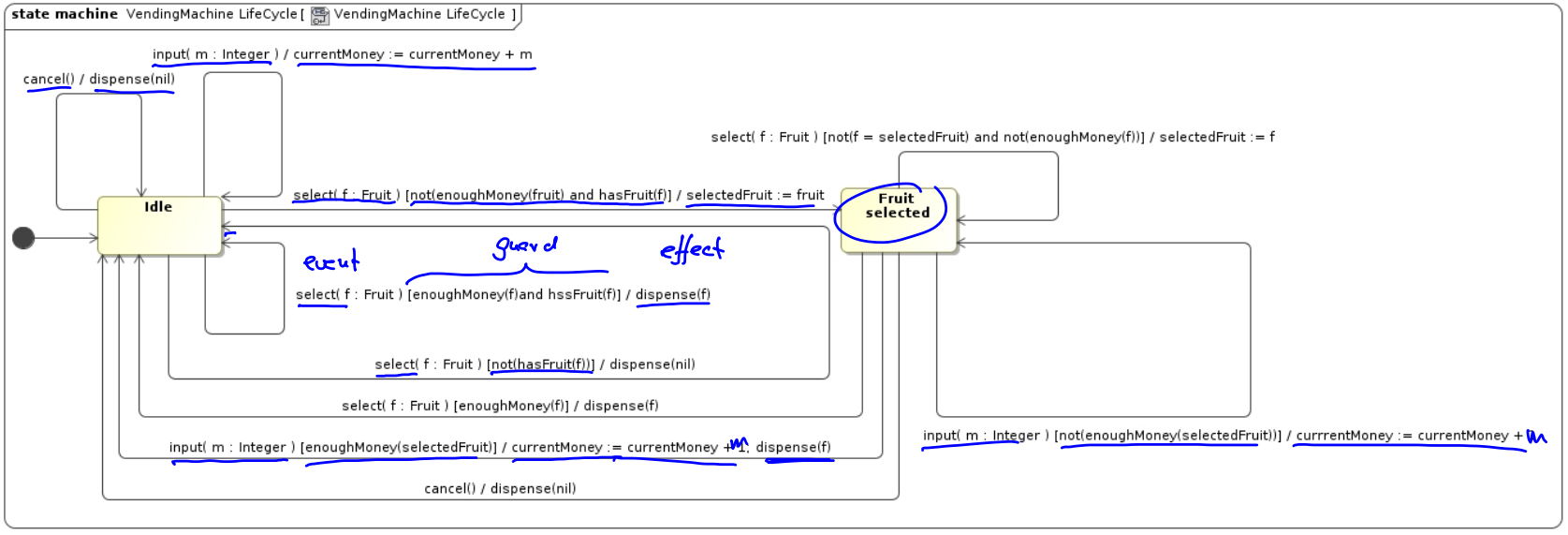
    $\rightarrow$ Automata

# Example Vending Machine




- Events
  - Input coins
  - Press button for bananas or apples
  - Press cancel
- Displays
  - current amount of money input
- Effects (Actions the machine performs)
  - Return money
  - Dispense banana or apple

# UML State Machines




- Easy to check for completeness: Does every state implement a reaction to every event?
- Easy to describe behavior: finite number of events and states

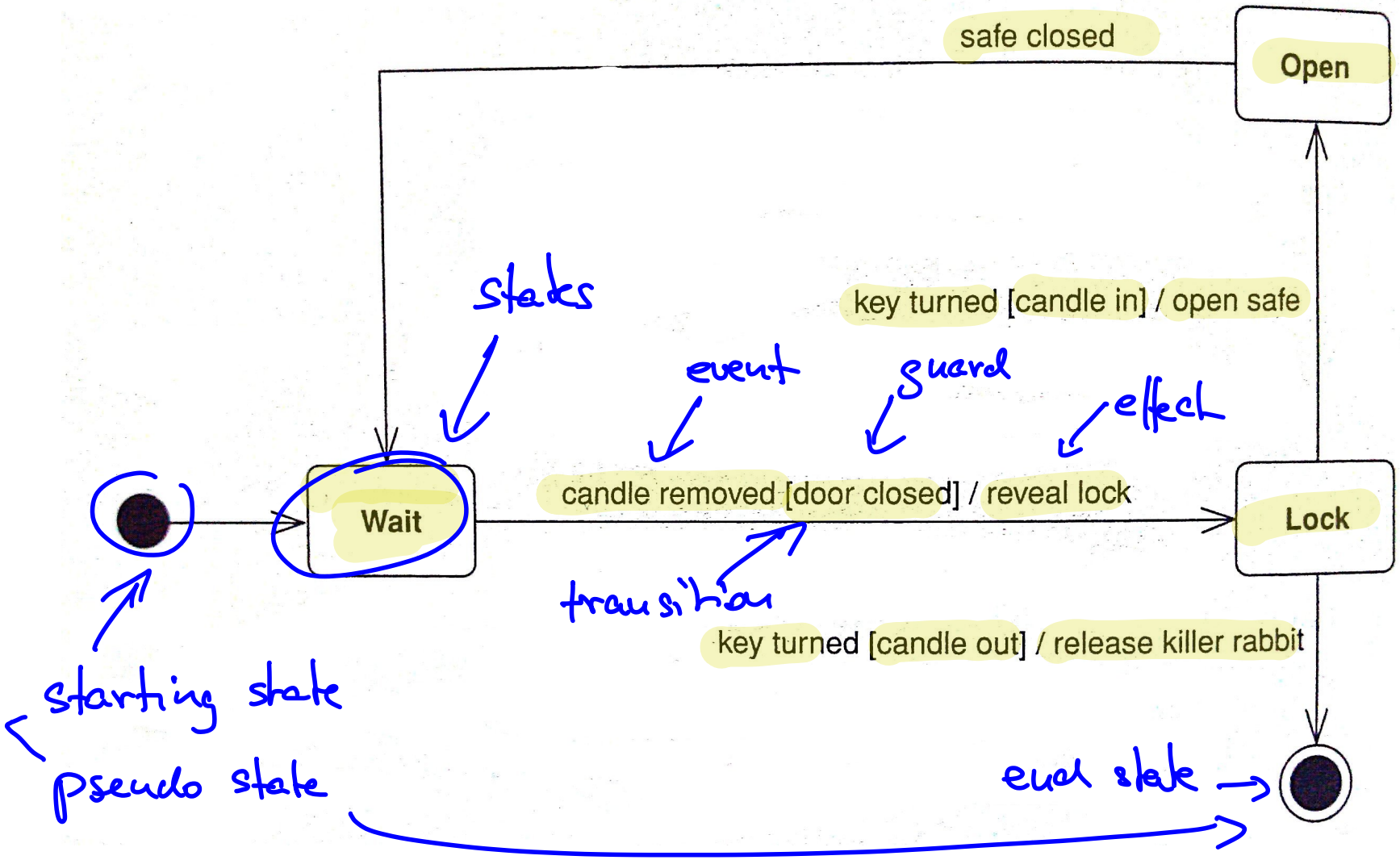
→ Good for this type of situations. For example, embedded systems

# Example: Safe

- Task: Implement a control panel for a safe in a dungeon
- The lock should be visible only when a candle has been removed
- The safe door opens only when the key is turned after the candle has been replaced again
- If the key is turned without replacing the candle, a killer rabbit is released

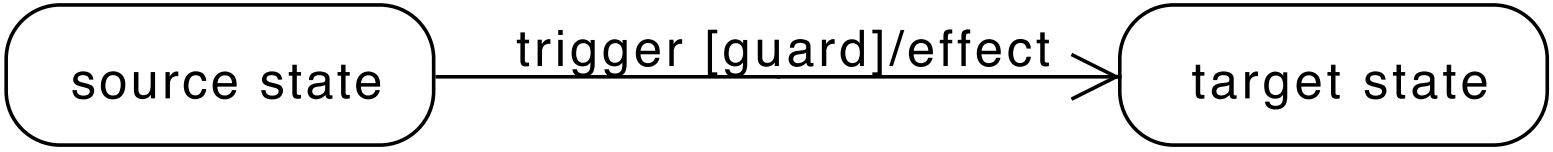

→ UML Distilled

# Example: Safe



# Transitions

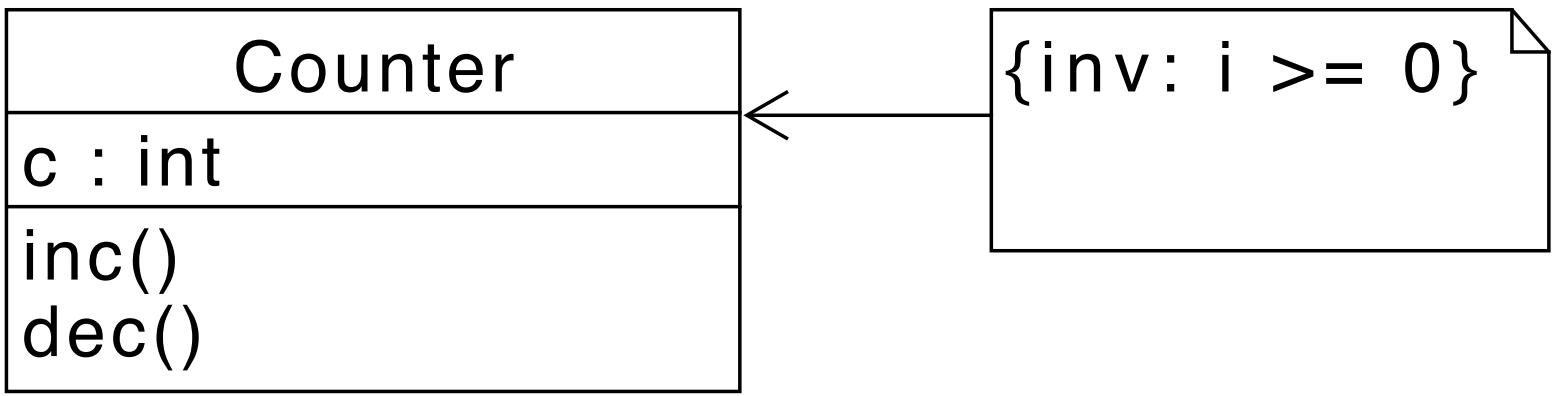
- General form



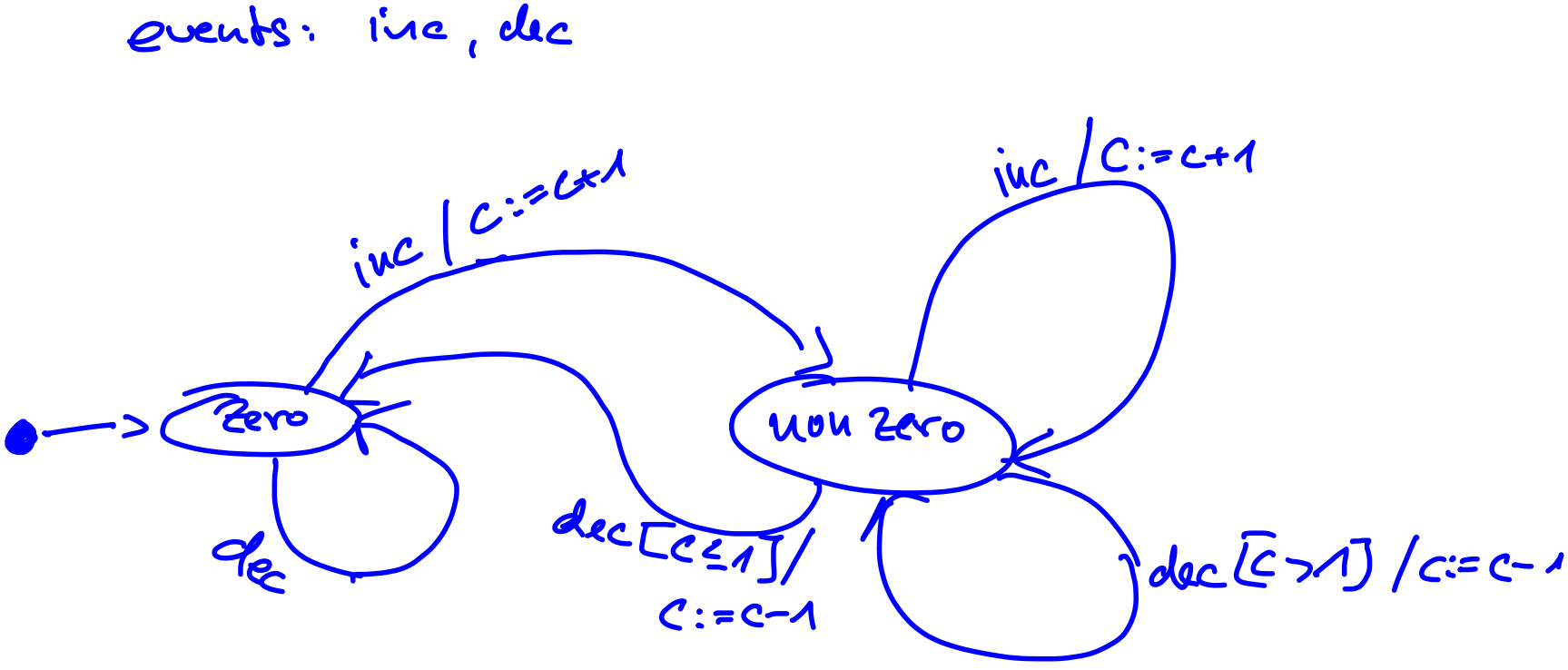

- Triggers (events, method calls)
- Guard: boolean expression
- Effect: a statement
- Fireing a transition
  - trigger + guard is true then the effect is executed

# Exercise

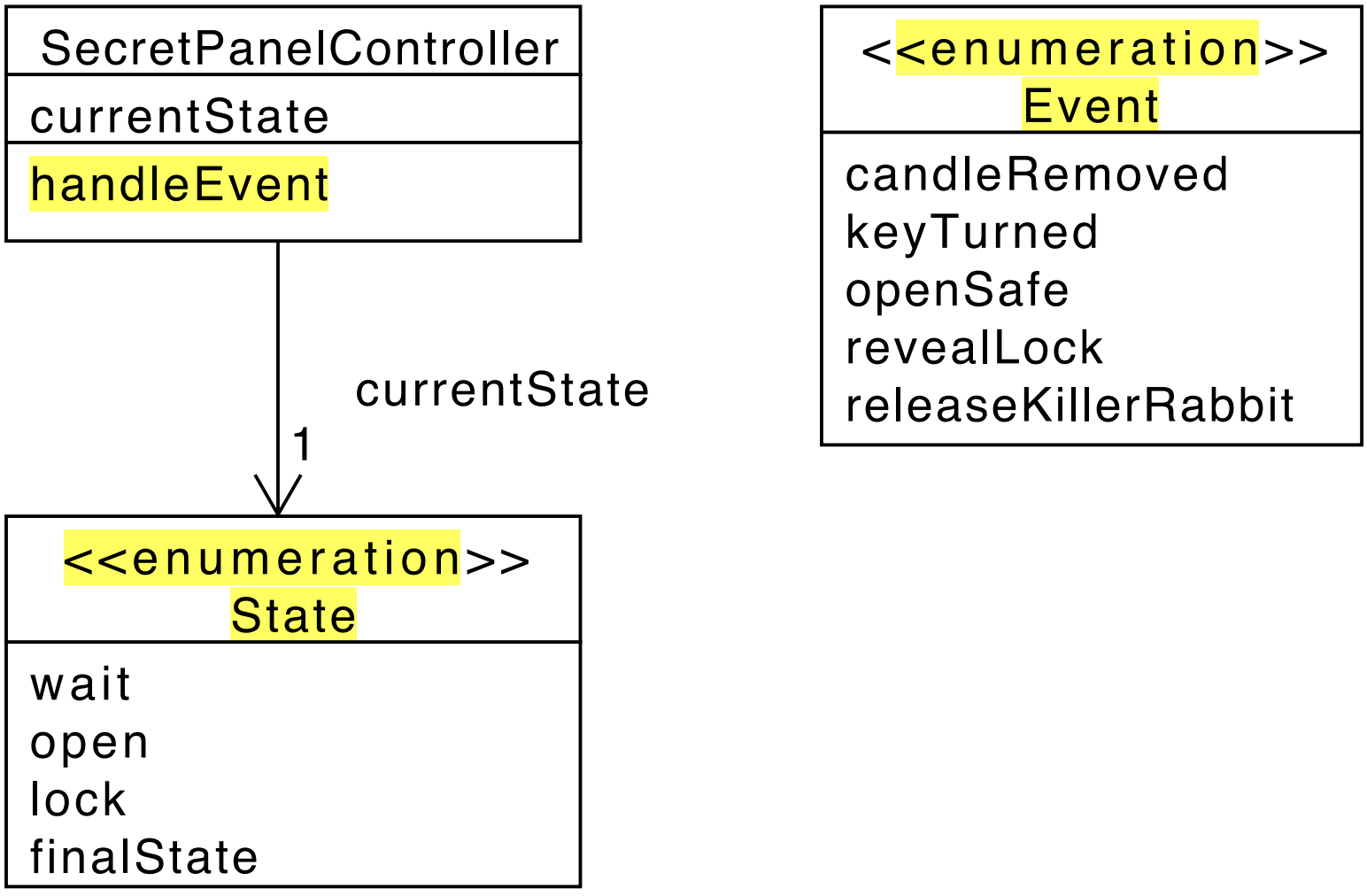
- Create a state machine for a counter object. A counter has two operations (= events): inc and dec and an attribute *c* that is never allowed to go below 0.
- Inc increments *c* by 1
- Dec decrements *c* by 1, but if *c* is 0, it does not do anything.



# The state machine for the counter



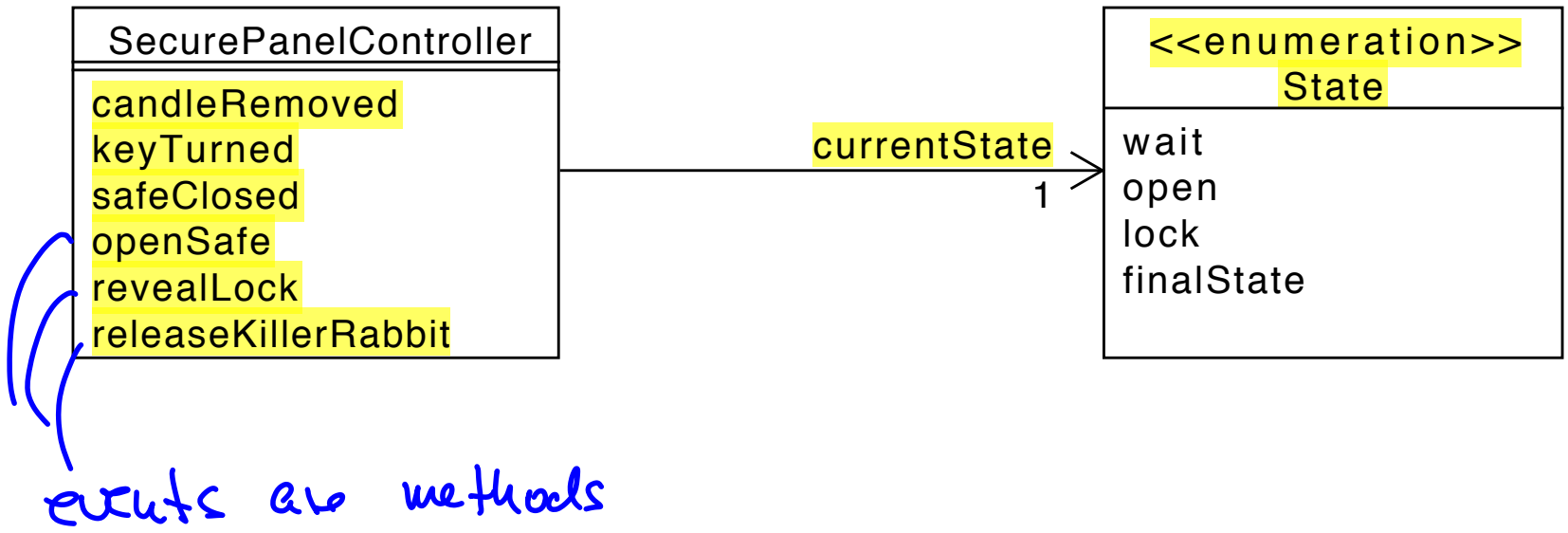
# Implementation 1: Class diagram



# Implementation 1

```
public class SecretPanelController {
 enum State = { wait, lock, open, finalState };
 enum Event = { candelRemoved, keyTurned, openSafe,
                revealLock, releaseKillerRabit };
 private State state = States.wait;
 public void handleEvent (Event anEvent) {
   switch (currentState) {
     case open :
       switch (anEvent) {
         case safeClosed :
           CurrentState = state.wait;
           break;
       }
       break;
     case wait :
       switch (anEvent) {
         case candleRemoved :
           if (isDoorOpen) {
             RevealLock();
             currentState = state.lock;
           }
           break;
       }
       break;
     case lock :
       switch (anEvent) {...}
       break;
    }
   }
 }
}
```

# Implementation 2: Class diagram



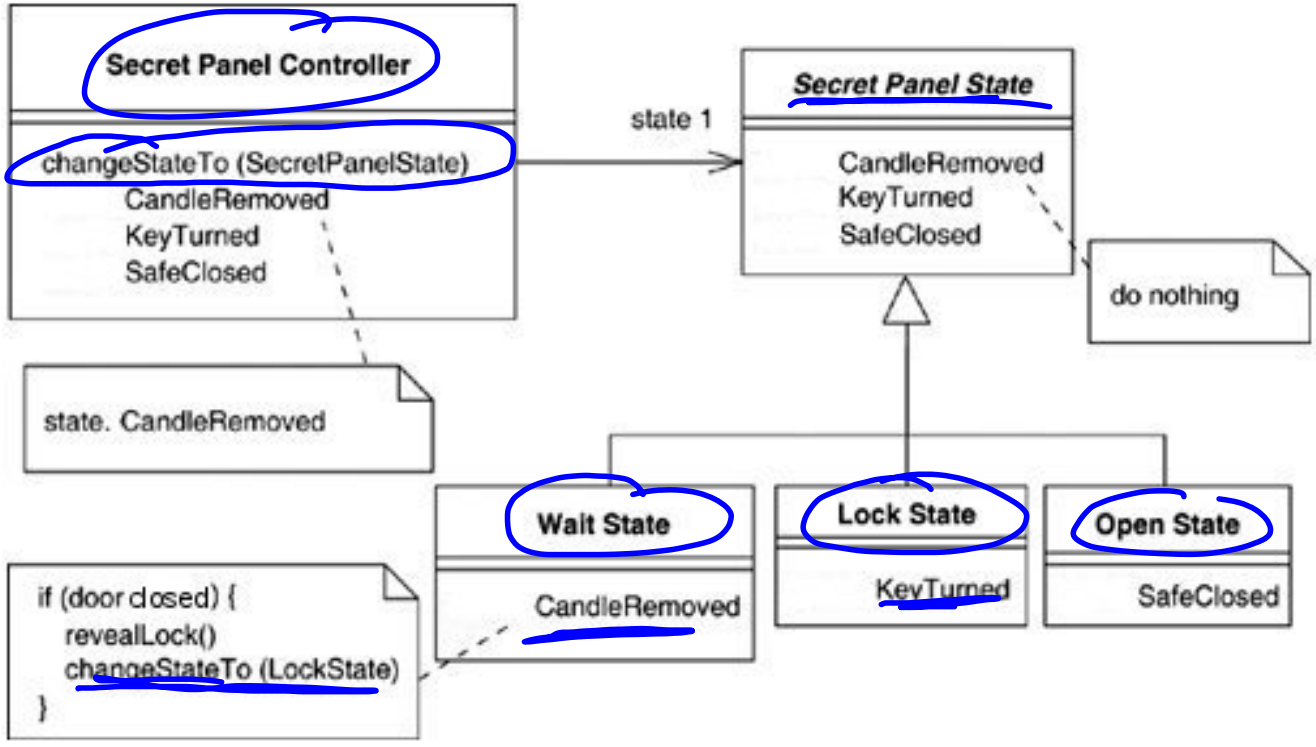
# Implementation 2

```
public class SecretPanelController {
   enum State { wait, lock, open, finalState };
   State state = State.wait;

   public void candleRemoved() {
      switch (state) {
      case wait:
         if (doorClosed()) {
            state = states.lock;
            break;
         }
      }
   }

   public void keyTurned() {
      switch (state) {
      case lock:
         if (candleOut()) {
            state = states.open;
         } else
         {
            state = states.finalState;
            releaseRabbit();
         }
         break;
      }
   } ... }
```

Code small switch statement

# Implementation 3: Using the state pattern

↳ one of the Design Patterns

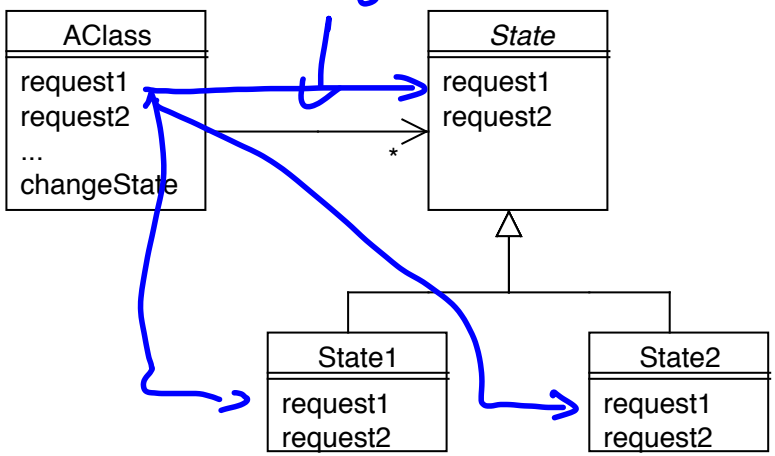


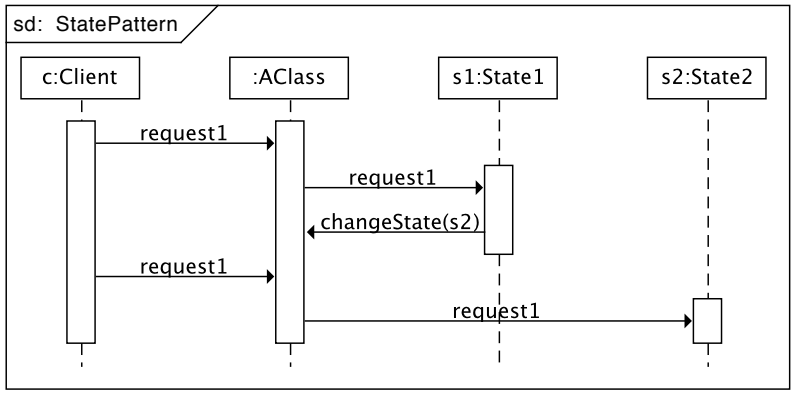
# State Pattern

## State Pattern

”Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.” Design Pattern book

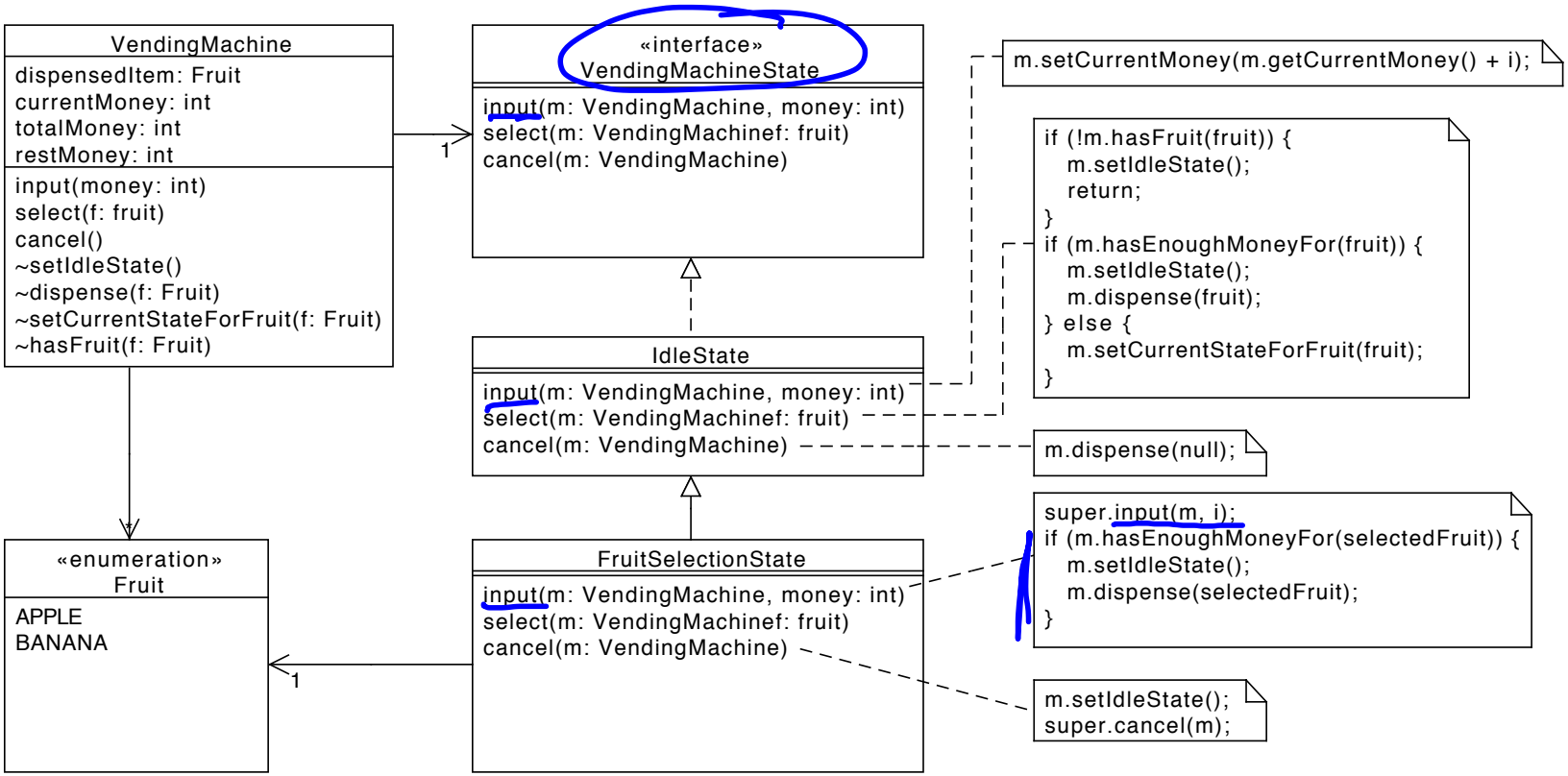
delegation



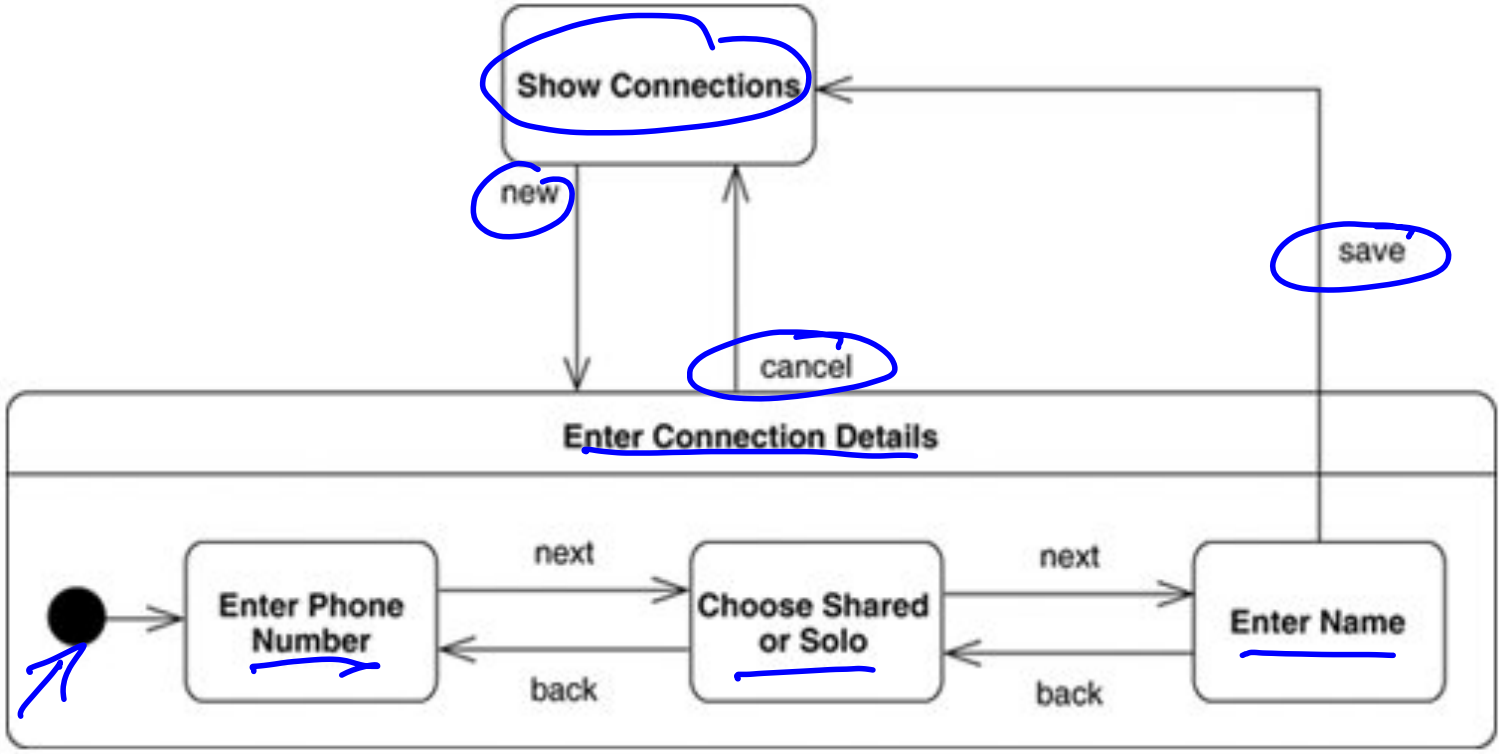
# Vending machine Implementation

Uses the state pattern



# Sub states

- Substates help structure complex state diagrams (similar to subroutines)



# Next week

- Layered architecture: persistency layer
- Software Development Processes
- Project Planning

# Contents

# Library Application: Text based UI

## User Screen

0) Exit
1) Login as administrator

*1*

## Login Screen

password
*adminadmin*

Logged in.

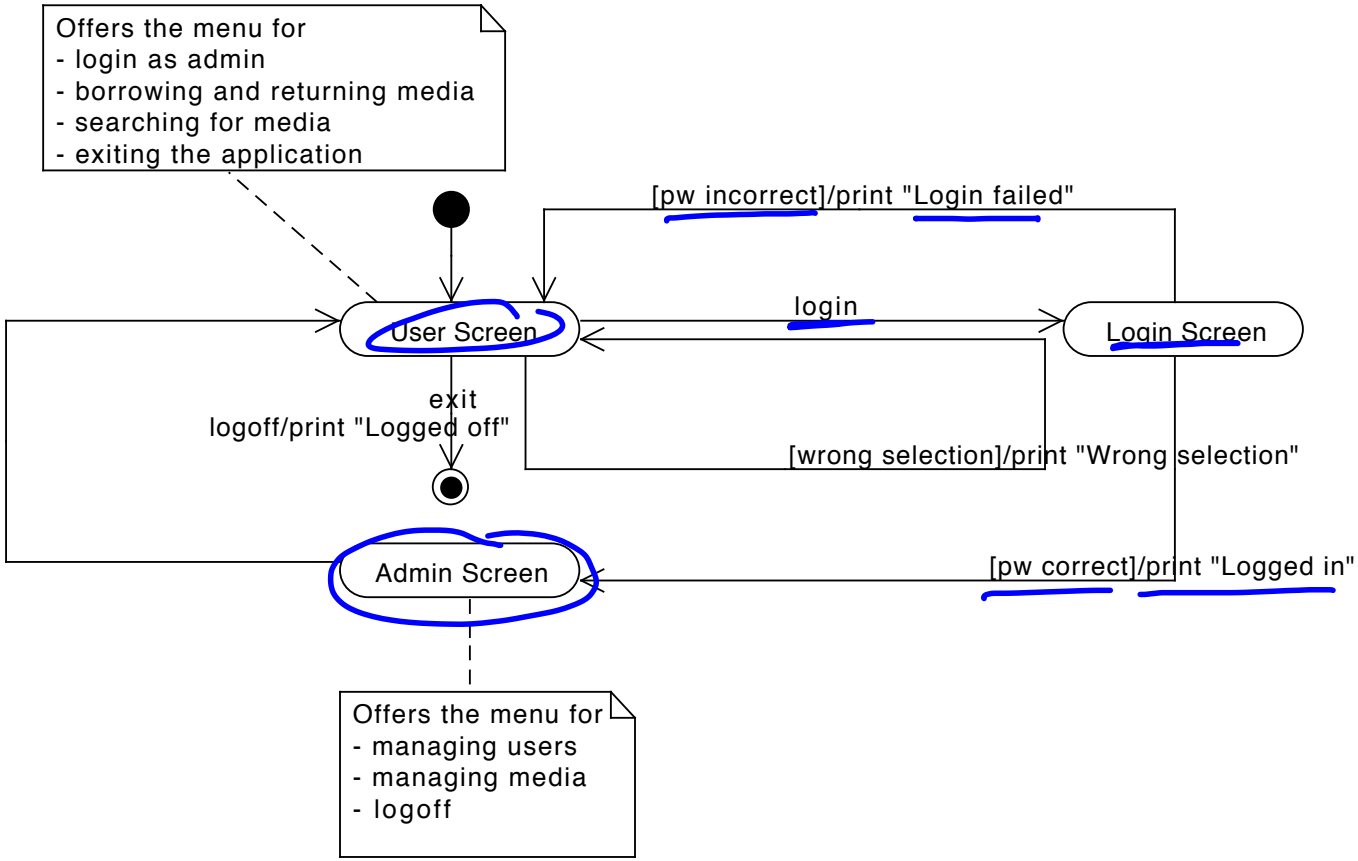
## Admin Screen

0) Logoff

*0*

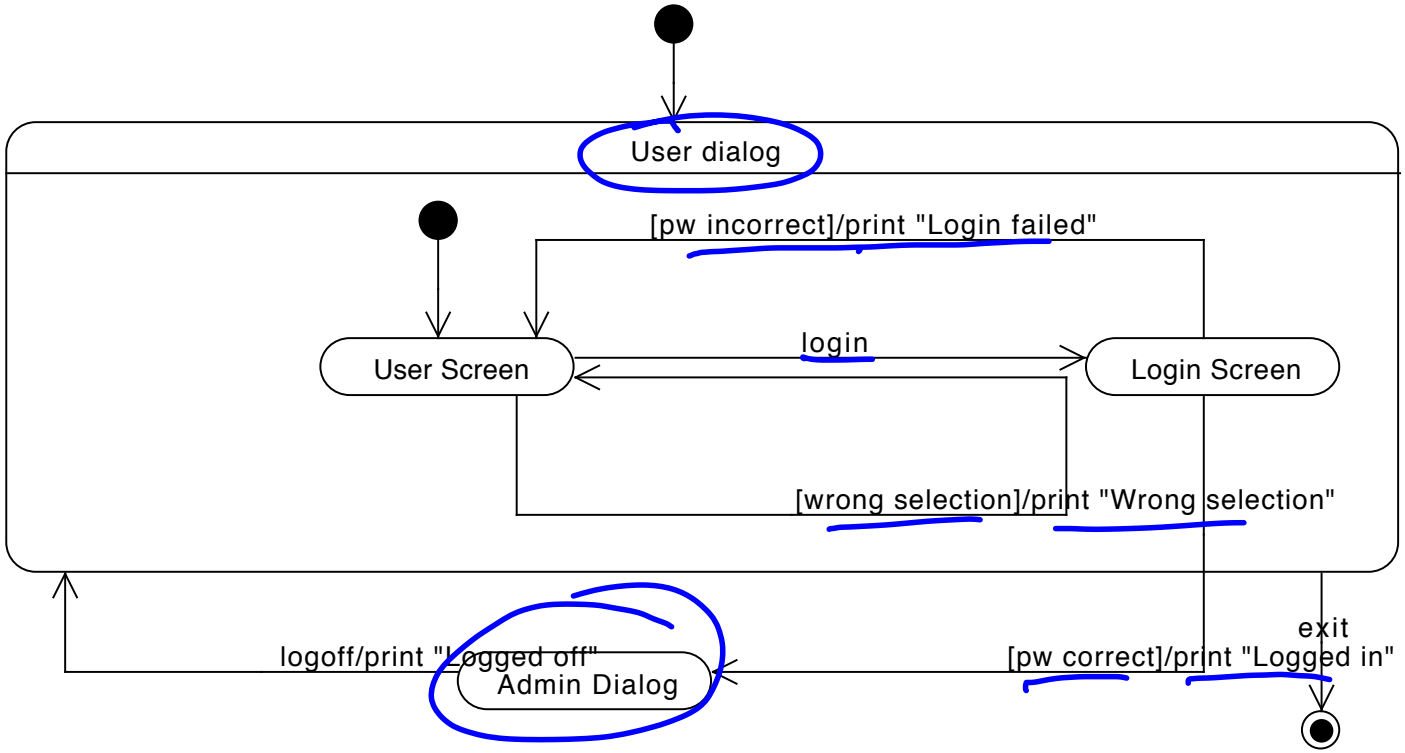
Logged off.

# Example Library Application



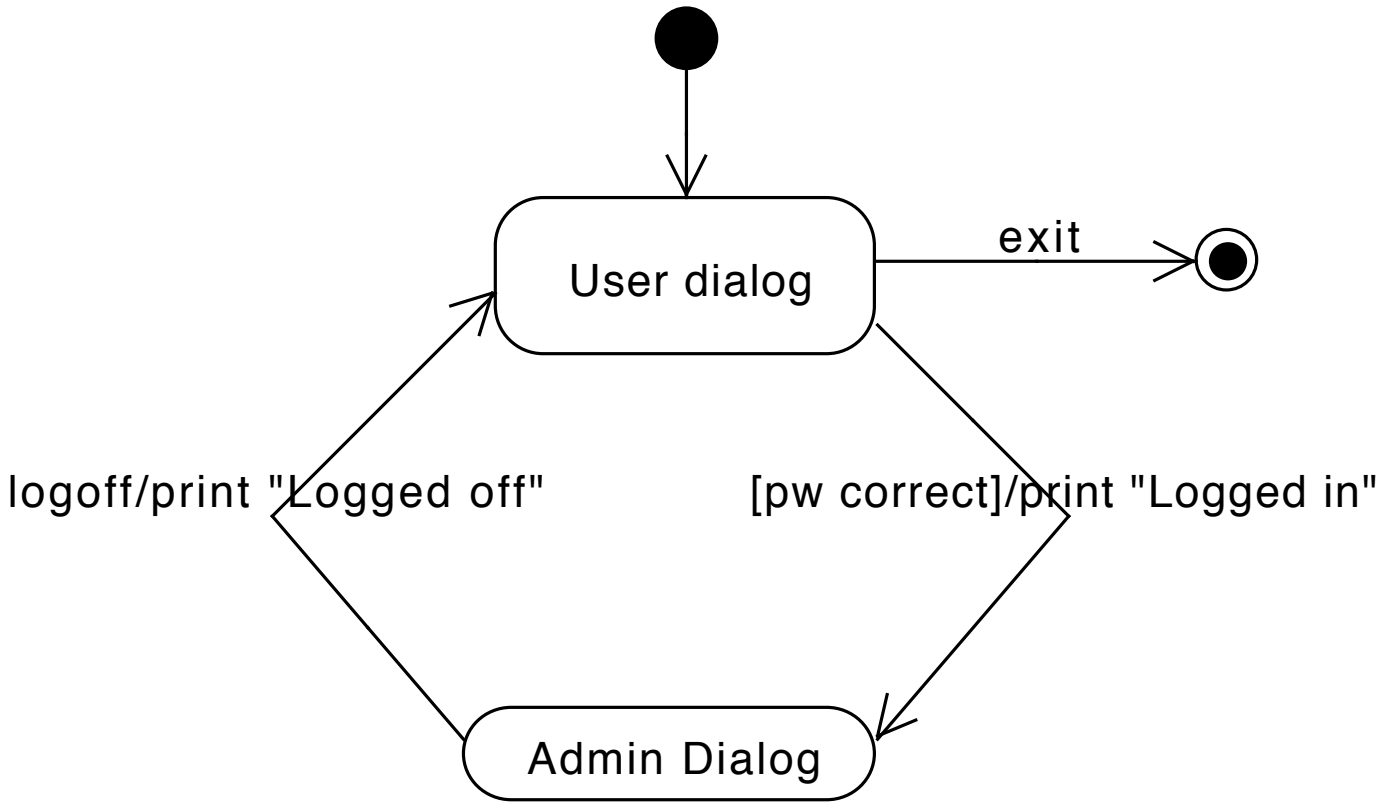
# Library App: Focus on user dialog

Use state UserDialog to group the user screen activities



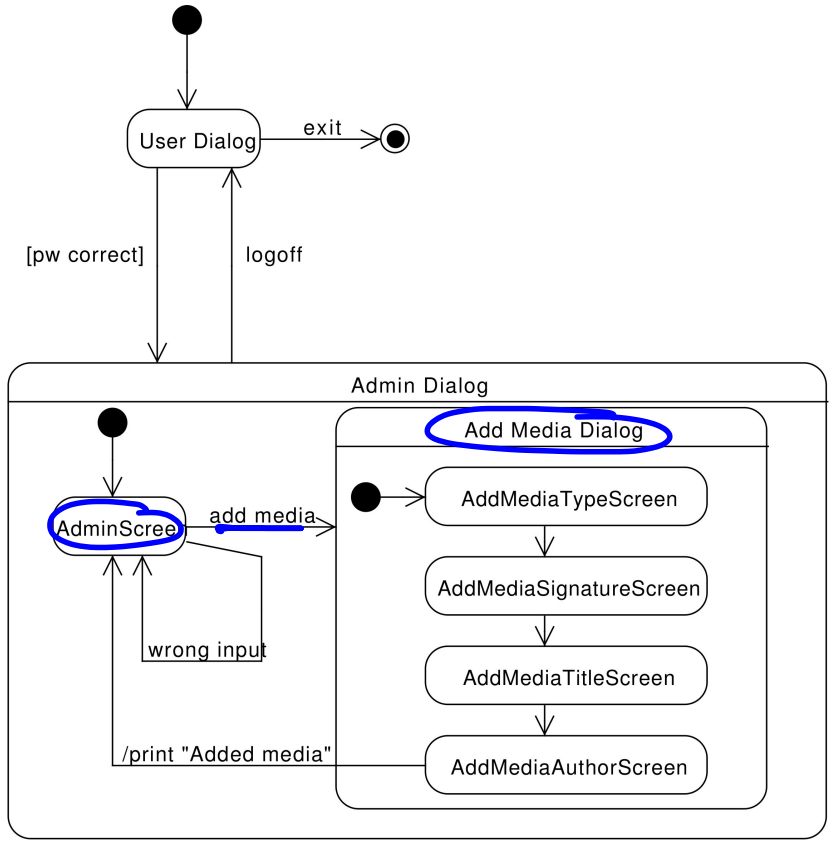
# Library App: Overview

Focus on the sequence of dialogs instead of screens

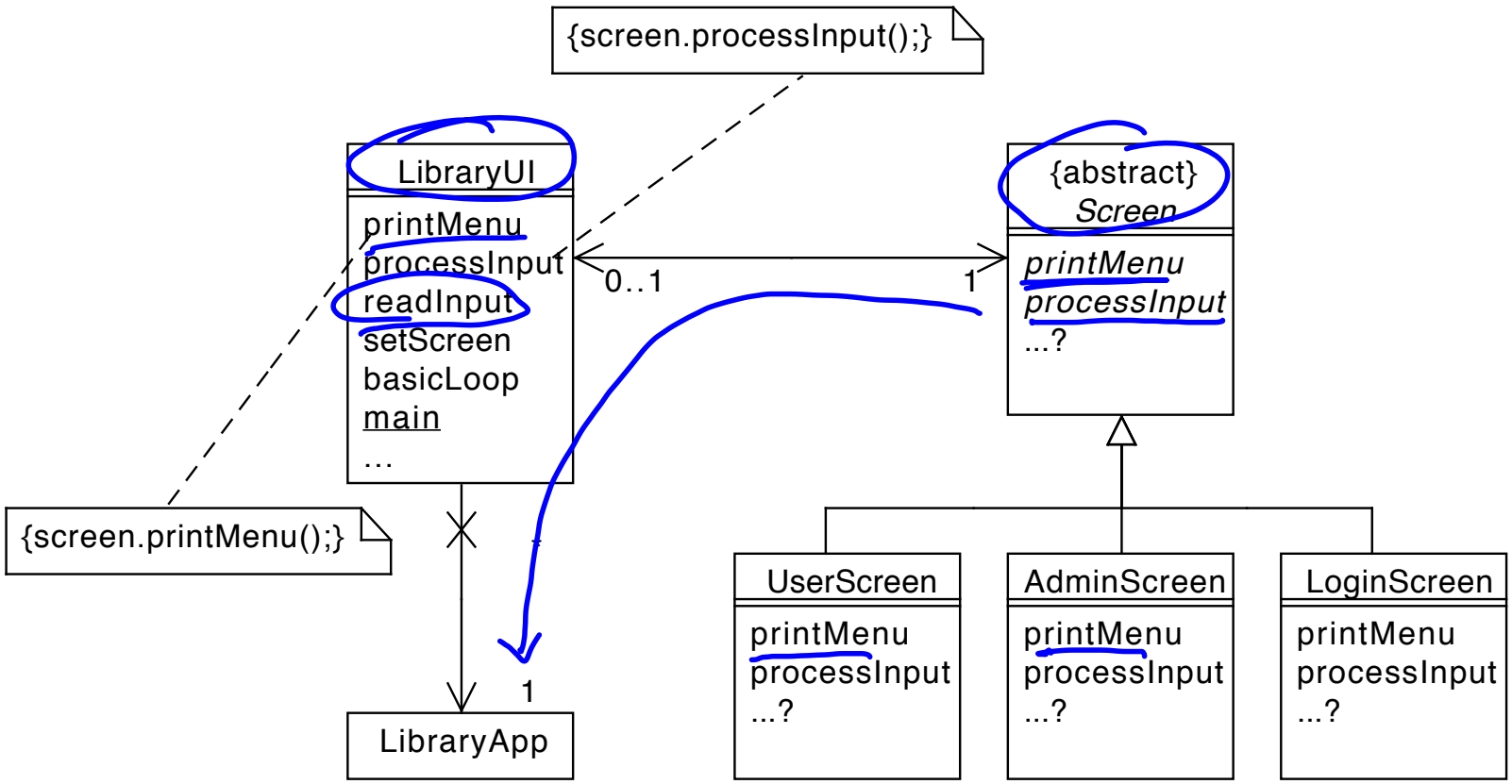


# Library App: Focus on admin dialog

Use state AdminDialog to group the admin screen activities



# Library App UI: State Pattern



# Library App: main application

```
public static void main(String[] args) throws IOException {
   BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));
   PrintWriter out = new PrintWriter(System.out, true);
   LibraryUI ui = new LibraryUI();
   ui.basicLoop(in, out);
}
```

can & testable
Feed System.in and System.out

## Basic loop

```
public void basicLoop(BufferedReader in, PrintWriter out)
      throws IOException {
  String selection;
  do {
    printMenu(out);
    selection = readInput(in);
  } while (!processInput(selection, out));
}

public void printMenu(PrintWriter out) throws IOException {
   screen.printMenu(out);
}

public boolean processInput(String input, PrintWriter out)
                                                  throws IOException {
   return screen.processInput(input,out);
}
```

# Library App user interface exercise (programming exercise 5)

1) Given tests for the functionality login; implement the tests using the state pattern
2) Design, test, and implement the remaining functionality of the library application