

Use case controllers:

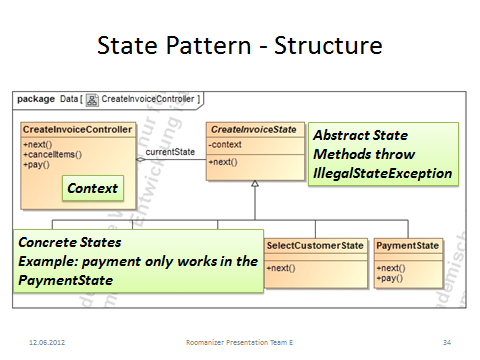
* Check In
* Create Invoice

Each step representing a state in the GUI

* At first we can see the search state (the start of the crate invoice use case).
  + Only search functions are allowed.
* In the payment state – (the last of the create invoice use case)
  + We can pay the invoice
* Abort and Back (first step not) is always allowed)

**Screens (Flip Chart, black board) – see sketch**

* Static problem with class change during runtime
  + Different controller classes (CreateInvoiceController in searchState,…
    - Nice polymorphic problem-solution, but….
      * We are using always the same controller from outside
        + Who would change the controller state? One more controller?
      * Switching the class of the object during runtime is not possible (in common Object Oriented programming languages)
        + Exception: Self (http://en.wikipedia.org/wiki/Self\_(programming\_language))
  + Solution: State Pattern (next Slide)



Our team made the decission to use the state pattern in our use case controllers.  
Example (Class-diagram of the Create Invoice Controller):   
Model-structure with roles

* Context: Controller – interface for gui
* Abstract state: interface for different states to encapsulate them
* Concrete state: implements the behaivour, which is connected to state of the context object

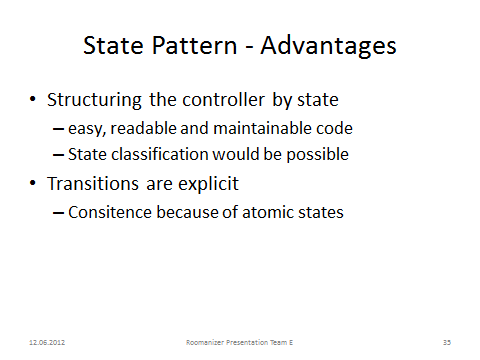
The controller looks now like an object which changed his class (from the outside – the gui or something else)!

For each step in the process, displayed as a single view in the user interface we created one state.

From the outside, we always use the same controller, but depending on the current state, different actions are performed or an exception is thrown in the case, that this operation is not accessible in the current state.

In example: the next method is implemented in every state, but depending on the state, the next state is another one.

Other functions, like for instance the pay function, are only implemented in one state.

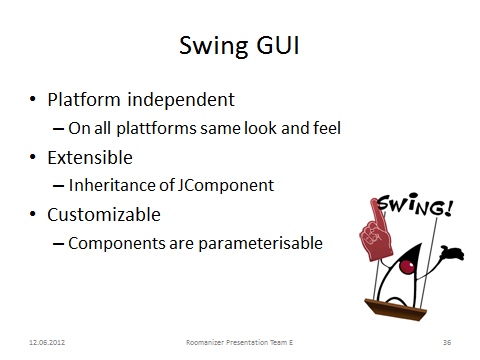


Structure:

* the structure of the state-pattern prevents long switch-case statements in the operations
* Classification is possible, because super and sub classes in the states are supported

An advantage of the state pattern is the easy way to create new states. You can just insert a new state-class and link it depending on the called functions together with the either states.

Transitions explicitly: the state is an external class; so – from the perspective of the context (Controller) – the state-transition is atomic, because there is only one specific state



Introduction: Swing is between SWT *THE Java GUI Framework*   
Platform independent (comparison AWT)

* Operating System native implementation with AWT 🡪 in swing, supports its own rendering 🡪 so it is possible to have the same design on different system (for instance MAC OSX, Windows or Linux)

Extensible: creating own components (for example the split and storno screen, if we use it on other positions 🡪 reusability)

* able to use the whole power of JComponents (Swing) like fire events, bound properties, java beans, many methods (which can be overwritten)

Customizable: standard set of elements, such as a border, inset, decorations, and other properties are available to customize tables, panels or buttons

Project:

Requirement 🡪 easy to integrate other guis, IF THEY USE THE SAME!