# 2IPC0 Programming Methods

## From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

# Overview

- State Design Pattern
- Open-Closed Principle (OCP)
- Model–View-Controller architecture

# Why Use Design Patterns in Software Development?

?

## Why Use Design Patterns in Software Development?

Major concerns for software developer:

1. Functional correctness ("It works")
2. Performance: speed, memory, ...
3. Verifiability*, incl. testability
4. Maintainability*
   - Design documentation, understandability
   - Modifiability, extendibility, ...
5. Reusability*

*Design patterns help with 3, 4, 5, by standardizing & weakening *dependencies*. There is a trade-off with performance!

## Taxonomy of Design Patterns

- Creational patterns

  Factory Method, Singleton

- Structural patterns

  Composite, Façade, Adapter, Decorator

- Behavioral patterns

  Strategy, Iterator, Observer, Command, State, Template Method

- Concurrency patterns

  "`SwingWorker`"

# State Pattern Motivation

- Multiple states/modes: with same events, different behaviors
- Compare to Finite State Machine

| **Action →** | | **Event** | | |
|---|---|---|---|---|
| **Next State** | | **A** | **B** | **C** |
| **State** | **S1** | `inc(A)` → S1 | → S3 | → S2 |
| | **S2** | → S1 | `inc(B)` → S3 | → S2 |
| | **S3** | → S1 | → S3 | `inc(C)` → S2 |

Concern: How to prevent clumsy conditional selection of behavior?

# State Anti-Pattern

- Use a state variable and **`if`** or **`switch`** to vary behavior per event

See `StateAntiPattern.zip`

## Intent of State Design Pattern (quoted from Eddie Burris)

- If an object goes through clearly identifiable states [modes],
- and the object's behavior is especially dependent on its state,

[then] it is a good candidate for the State design pattern.

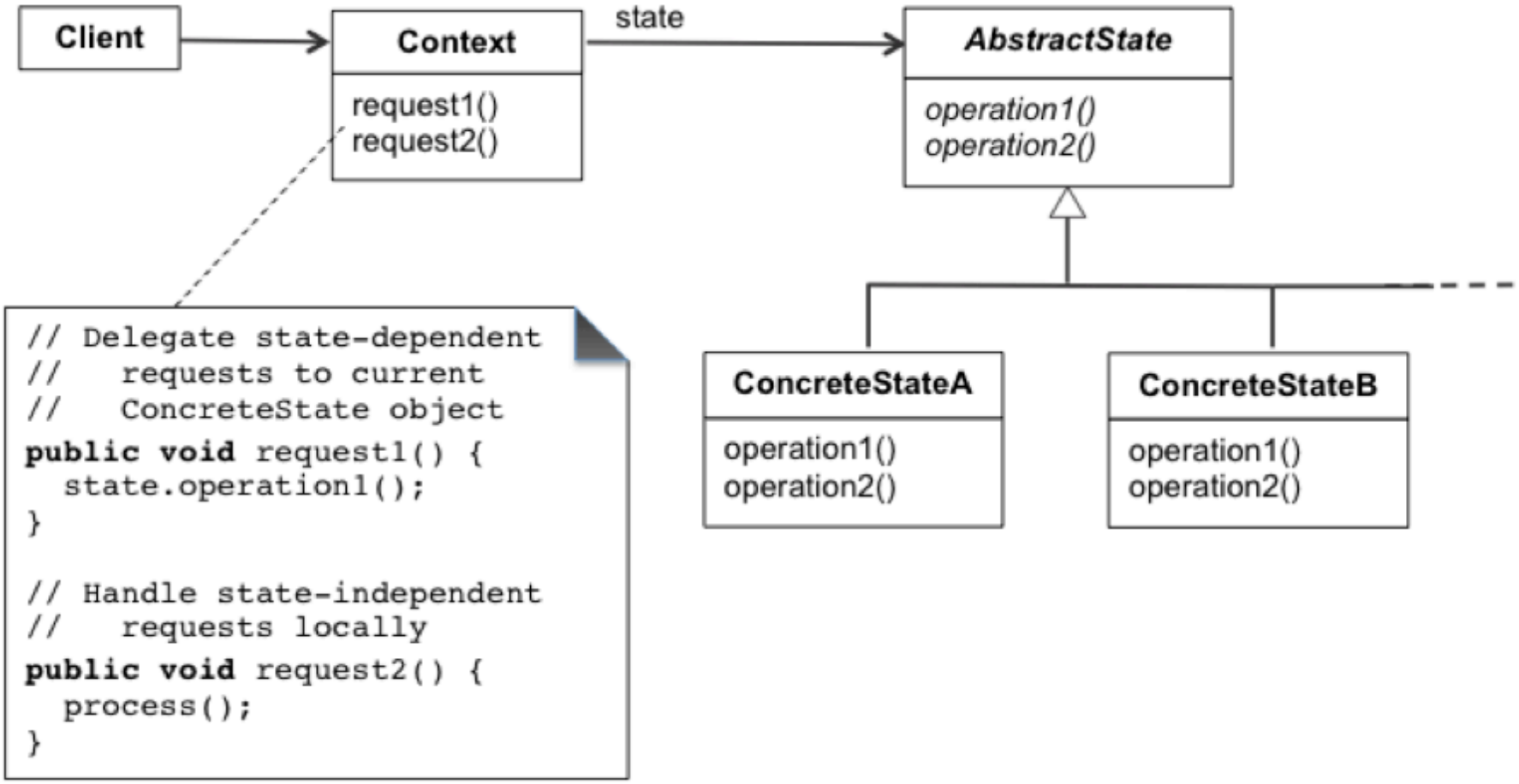
## State Pattern Solution (adapted from Burris)

- Superclass or interface specifies all events handled by a state
- The concrete event handlers are implemented in concrete states

  Different states can handle the same event in different ways
- *Context* code
  - holds a current state object
  - *delegates* event handling to the current state object
  - decides on change of state
- Alternatively: event handlers decide on change of state

See `StatePattern.zip`

# State Pattern Class Diagram (from Burris)



## State Pattern Variations

- Where to decide on new state? In context or in handler?
- How to avoid creating new objects for recurring states?

  Global concrete state constants...

  Singleton concrete state classes...

See: `StatePattern2`, `StatePattern3`

# State Pattern: How Does It 'Work'?

?

## State Pattern: How Does It 'Work'?

Based on

- Polymorphism: Concrete state object can be assigned to abstract state variable.

  ```
  State state = new ConcreteState1();
  ```

- Dynamic binding: When a method is called on the abstract variable, the implementation is looked up dynamically based on the runtime type.

  ```
  state.event1()
  ```

  This involves a look-up of the implementation of method `event1()`.

– O – O – O – O – O –

NEW TOPIC

## SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, treated in this lecture)
- **L**iskov Substitution Principle (LSP, see Lecture 5)
- **I**nterface Segregation Principle (ISP, treated later)
- **D**ependency Inversion Principle (DIP, see Lecture 10)

See Downloads page for links.

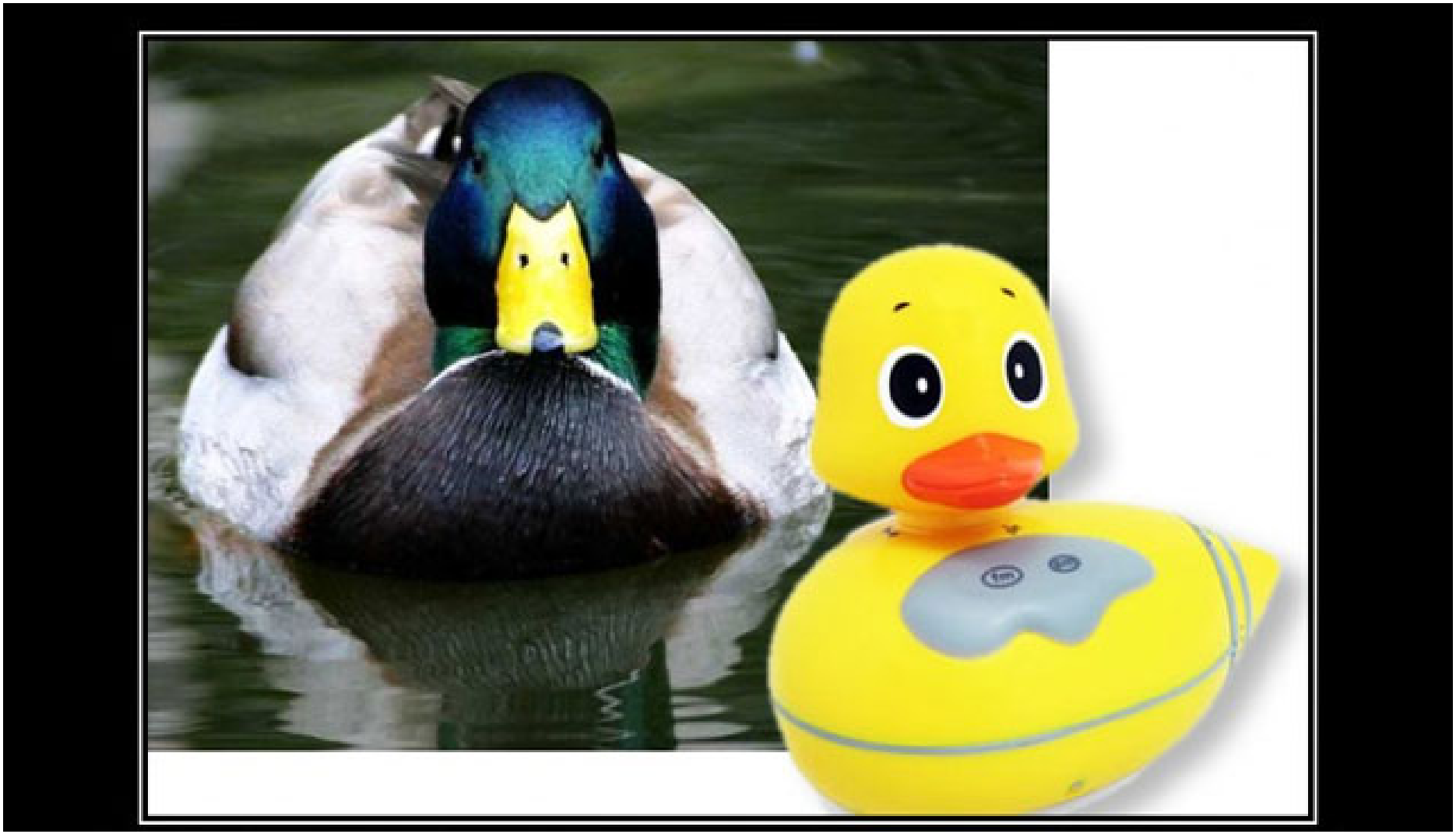
The following pictures were obtained from Kaur Mätas:

`zeroturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-sol`

# Which Principle?

# Which Principle?

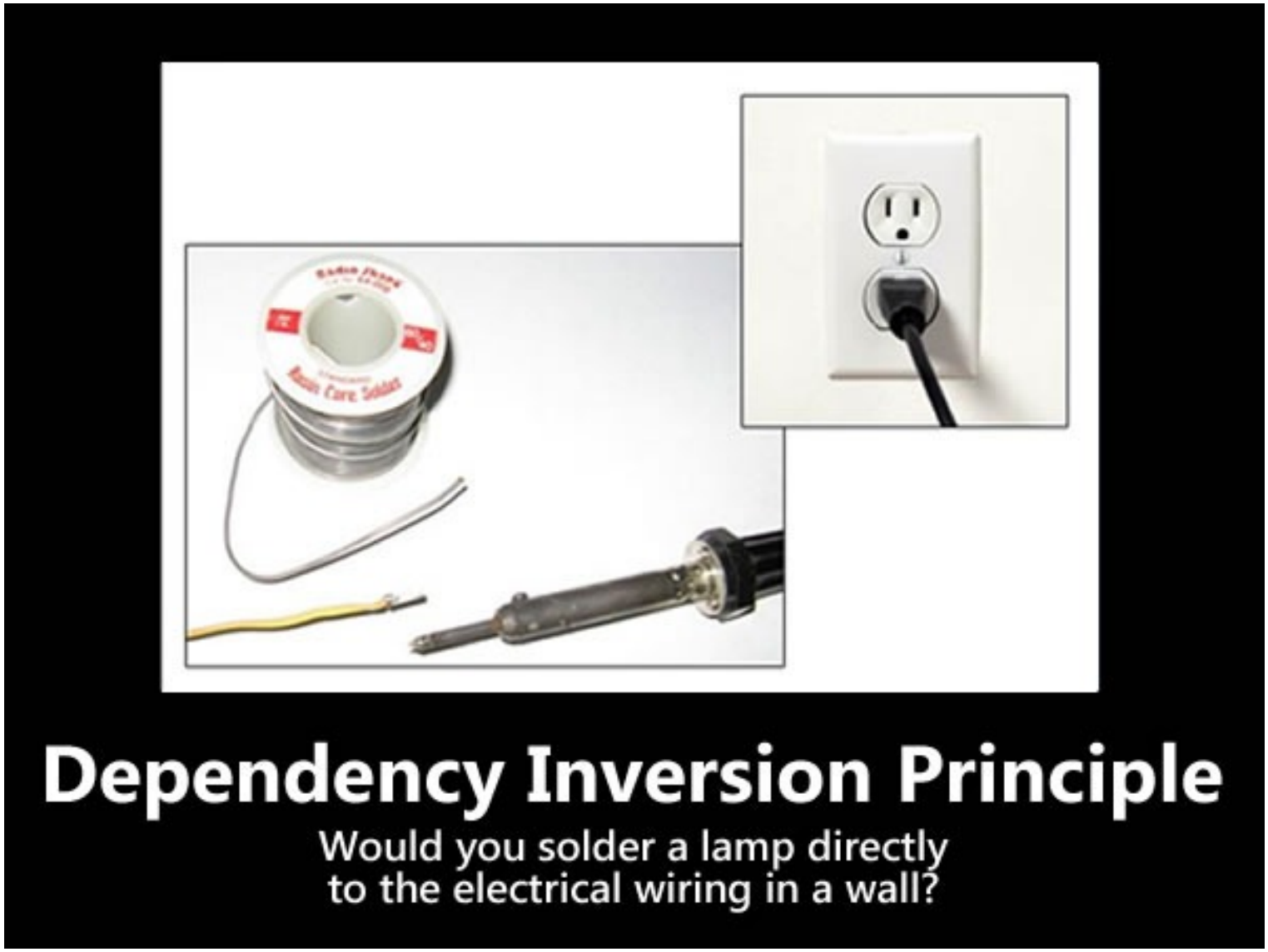
# Which Principle? If it looks/quacks like a duck, but . . .



# Which Principle? If it looks/quacks like a duck, but . . .

# Which Principle?

# Which Principle?



# Which Principle?

# Which Principle?

## The Two Ways of (Re-)Using a Class

1. By regular client code:
   - Instantiate **`new`** objects, and call methods on them
   - Contracts usually specify this interface
   - Client code can access **`public`** members only
2. By code in (possibly anonymous) subclass that **`extends`** the class:
   - Access **`super`** functionality
   - Contracts for this use are harder to specify (usually not done)
     E.g., `execute()` in abstract command of `CommandPattern`
   - Subclass code can access **`protected`** members

## Open Closed Principle (OCP)

- Modules should be open for extension (via inheritance)

  Inheritance allows reuse of implementation:

  - representation (instance variables)
  - operation definitions (method bodies)

  Can add instance variables/methods, and override methods *without modifying the superclass*

- Modules should be closed for modification (by clients)

  **`private`**, or at least **`protected`**, instance variables

  Do not modify code in order to add or adapt functionality

## Adapter/Decorator Pattern and OCP

- Patterns usually offer ways to adhere to OCP
- The adapted class (via inheritance) is not modified but extended
- The decorated class (via composition) is not modified but composed into decorator

## State Pattern and OCP

- The State Pattern is a way of adhering to OCP
- Problem: Add a new state with different behaviors for the events
- Antipattern: Modify the code
- Pattern: Add subclass for new state with new event behaviors

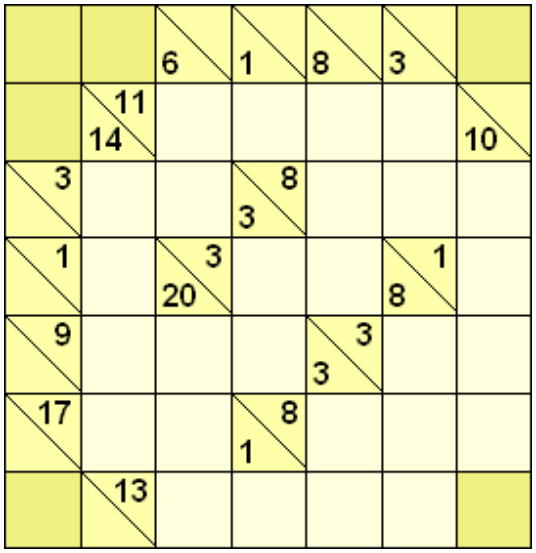
## Criticizing Design Patterns & SOLID OO Design Principles

- Do not take anything for granted.
- Try to understand it "deeply".
- Criticism is possible (and useful).

– O – O – O – O – O –

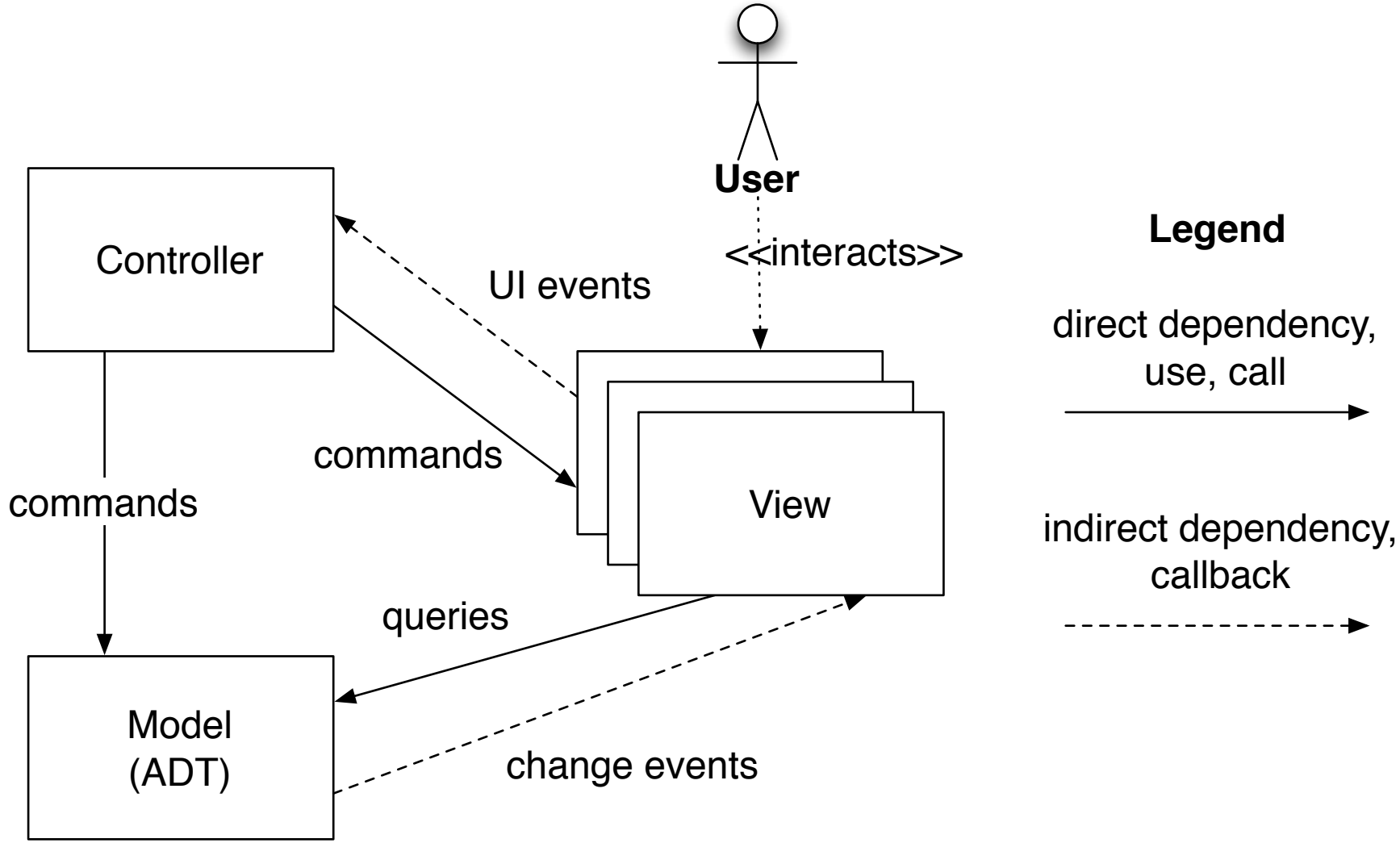
# NEW TOPIC

# GUI Organization Problem: Model–View–Controller




- Need to accommodate:
  - Underlying abstract model (Abstract Data Type, possibly $> 1$)
  - Present (possibly $> 1$) views on model to user (queries)
  - Provide user with control over model (commands)
- Separation of concerns
- Avoid unnecessary dependencies between modules

# Model–View–Controller Architecture



## Model–View–Controller Architecture

- Uses Observer Pattern (Strategy Pattern) to adhere to DIP
- Concrete models depend on abstract model listener interface, implemented by concrete views; not on concrete views

  Model notifies its listeners, without knowing their implementation
- Concrete views depend on abstract model interface, implemented by concrete models; not on concrete model

  View can receive concrete model as parameter of notification
- Cf. 2IPD0 (Software Engineering) / 2IW80 (Software Specification and Architecture)

## Model–View–Controller Example

- See `ModelViewController.zip`
- Model is ADT for resettable and incrementable counter
- Views show count in different notations (binary, decimal, . . . )
- Model does not depend on controller or concrete views

  Model depends on listener interface, implemented by views
- Concrete views do not depend on model:

  Notification by model provides sufficient information to view

# MVC and State Pattern Example

- See `StatePattern4`
- Observable ADT to maintain score of players in a game
- Model pushes all data, so that view does not need model

## Summary

- State design pattern
- Open-Closed Principle (OCP)
- Model–View–Controller architecture