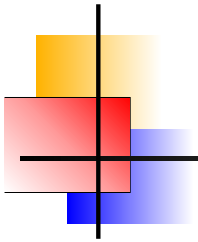


Programming Project:
Object-Oriented Design

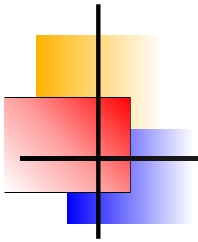
Guillermo Román Díez

groman@fi.upm.es

E.T.S. Ingenieros en Informática

Universidad Politécnica de Madrid

2019-2020



Illusion of Simplicity

The task of the software development team is to engineer the illusion of simplicity [1]

[1] Booch, 1993. *Object-Oriented Analysis and Design with Applications*

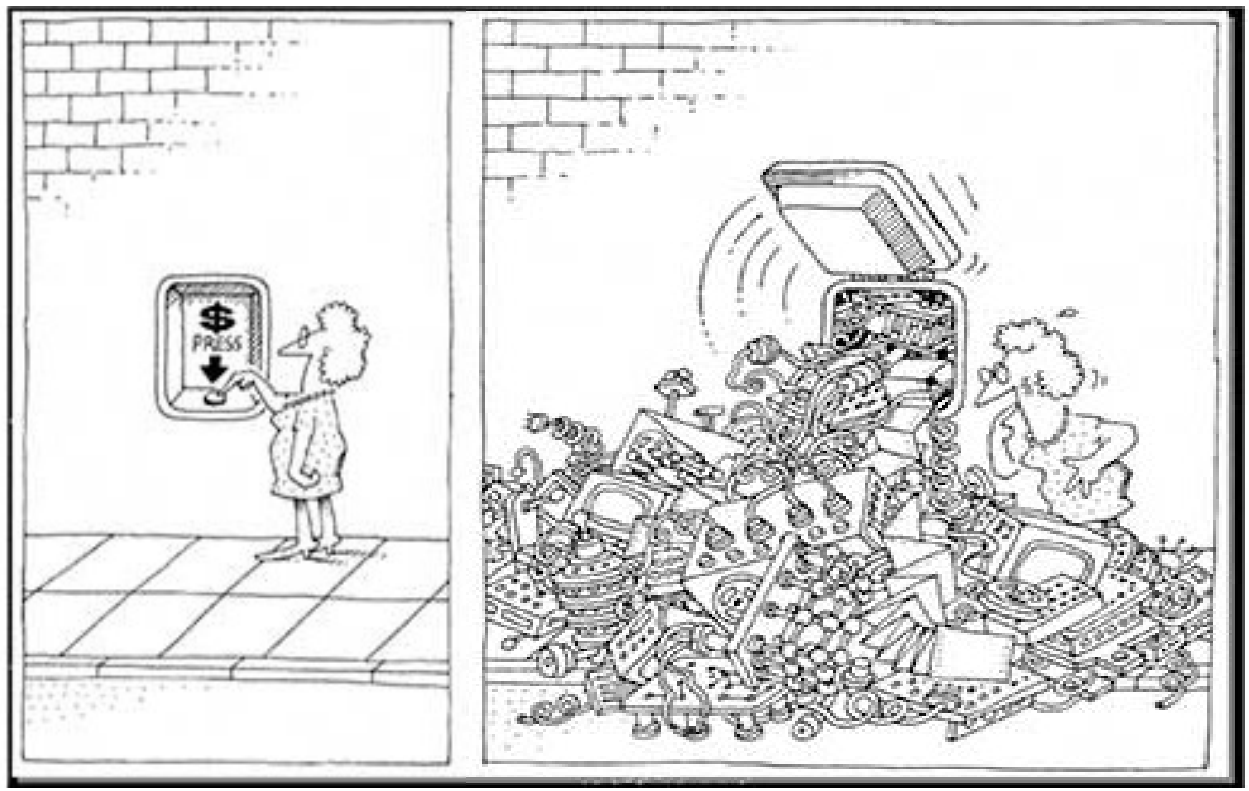
Illusion of Simplicity

The task of the software development team is to engineer the illusion of simplicity [1]

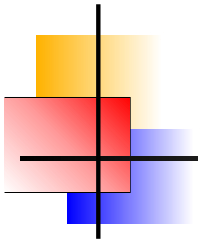
[1] Booch, 1993. *Object-Oriented Analysis and Design with Applications*

Illusion of Simplicity

The task of the software development team is to engineer the illusion of simplicity [1]

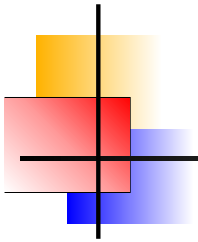


[1] Booch, 1993. *Object-Oriented Analysis and Design with Applications*



Software Complexity

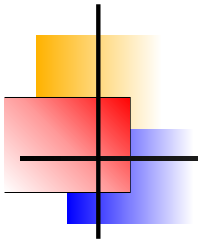
- Not all software systems are complex
  - There are systems developed and maintained by one person
  - This kind of systems usually have a very limited purpose and a very short life
  - Frequently, it easier to replace the full system than extend their functionality
- Developing industrial software is much more challenging
  - Industrial software is intensively difficult for the individual developer to comprehend all the details of its design
    - It could be thousands or even millions of lines of code
    - Multiple systems interacting at the same time
  - The complexity of such systems exceeds the human intellectual capacity



Software Complexity

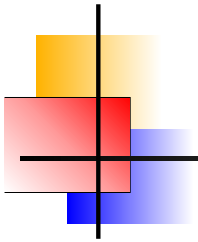
"The **complexity** *of software is an* **essential property**, *not an accidental one"*

- This *essential* complexity comes from four characteristics of software



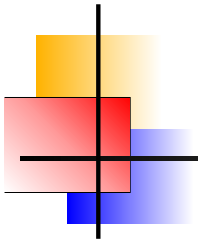
Software Complexity: Elements

- The complexity of the problem domain
  - Functionality of systems is difficult to comprehend
  - There exists a *communication gap* between users and developers
  - We have few instruments for precisely capturing the requirements
  - The requirements often change during the project
- The difficulty of managing the development process
  - No matter what its size, there are always significant challenges associated with team development
  - Big teams mean more complex communication and hence more difficult coordination



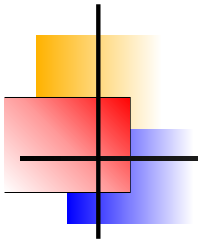
Software Complexity: Elements

- The flexibility possible through software
  - It is possible for a developer to express almost any kind of abstraction
  - Software development lacks of established standards
- The problems of characterizing the behavior of discrete systems
  - A large system might have thousands of variables accessed by multiple threads at the same time
  - A minimal impact in the input might produce an unpredictable change in the output
  - As exhaustive testing is impossible, we aim at having acceptable levels of confidence regarding their correctness



Software Decomposition

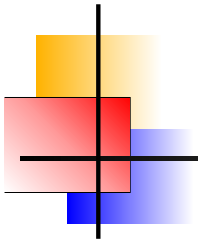
- There are fundamental limitations of the human capacity for dealing with complexity
- Some people might have exceptional capacities to do so, but



Software Decomposition

- There are fundamental limitations of the human capacity for dealing with complexity
- Some people might have exceptional capacities to do so, but

The world is only sparsely populated with geniuses. There is no reason to believe that the software engineering community has an inordinately large proportion of them

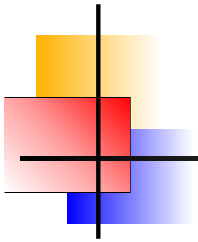


Software Decomposition

- There are fundamental limitations of the human capacity for dealing with complexity
- Some people might have exceptional capacities to do so, but

The world is only sparsely populated with geniuses. There is no reason to believe that the software engineering community has an inordinately large proportion of them

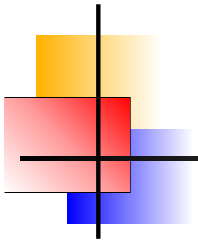
- We must consider more disciplined ways to deal with this complexity



Software Decomposition

The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)

- When designing a complex software system it is crucial to decompose the system into smaller parts such that:
  - We can refine each part independently
  - To understand a level we only need to comprehend some parts
- However, . . .

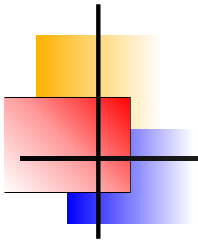


Software Decomposition

The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)

- When designing a complex software system it is crucial to decompose the system into smaller parts such that:
  - We can refine each part independently
  - To understand a level we only need to comprehend some parts
- However, . . .

There is no magic, no silver bullet



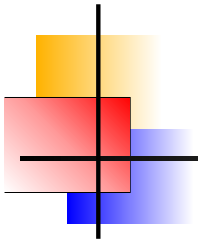
Software Decomposition

The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)

- When designing a complex software system it is crucial to decompose the system into smaller parts such that:
  - We can refine each part independently
  - To understand a level we only need to comprehend some parts
- However, . . .

There is no magic, no silver bullet

- The design of complex systems involves an incremental and iterative process
- We will focus on **object-oriented design**



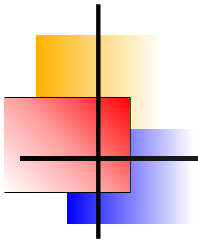
Object Oriented Programming

Object-oriented programming

Is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships

- It supports objects that are data abstractions with an interface of named operations and a hidden local state
- Objects have an associated type (class)
- Types (classes) may inherit attributes from supertypes (superclasses)

The essential tool for designing an object-oriented program is the use of **abstractions**

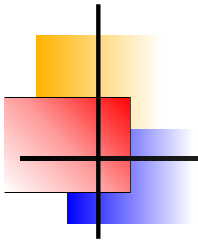


Abstraction

Abstraction

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer

- The main goal of abstraction is to separate the object's essential behavior from its implementation
- An object should try to follow this two principles:
  - *Principle of least commitment*: The interface provides the essential behavior of the object, and nothing more
  - *Principle of least astonishment*: The behaviour of an object does not offer surprises or side effects that go beyond the scope of the abstraction
- As we have multiple objects collaborating in a program, the abstraction of an object should precede the decisions about its implementation: **encapsulation**

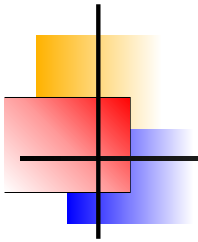


Encapsulation

Encapsulation

Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation

- No part of a complex system should depend on the internal details of another part
- Most times, encapsulation is achieved through **information hiding**, that is, the process of hiding the secrets of an object that do not contribute to its essential characteristics
  - The structure of an object is hidden (attributes should be private and non-static)
  - Methods implementation should not be relevant

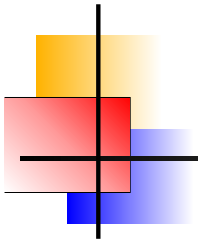


Encapsulation

- Now we will see some **principles** that any object-oriented software should try to follow:

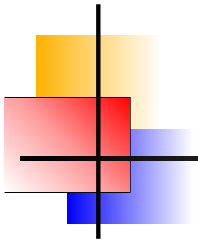
The S.O.L.I.D. Principles

1. **Single responsibility principle**
2. **Open/closed principle**
3. **Liskov substitution principle**
4. **Interface segregation principle**
5. **Dependency inversion principle**



Single responsibility principle

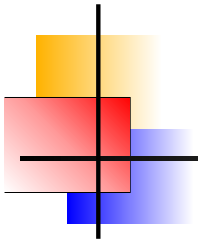
A class should only have one reason to change



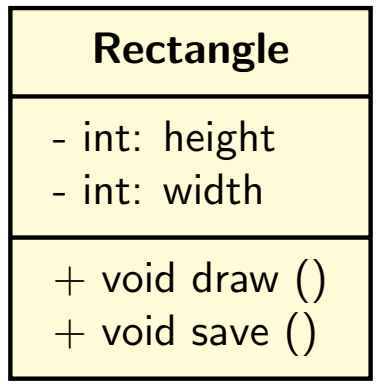
Single responsibility principle

A class should only have one reason to change

- We say that an object has a single responsibility when it does only one thing, that is, *it does not do two things*
- In this context, a responsibility can be considered *one reason to change* a class
  - If we have 2 reasons to change for a class, we should split the functionality in two classes or two interfaces
- If we have to change a functionality we are going to change the class which handles it
  - If one class has two responsibilities, a change in a functionality might affect other functionalities



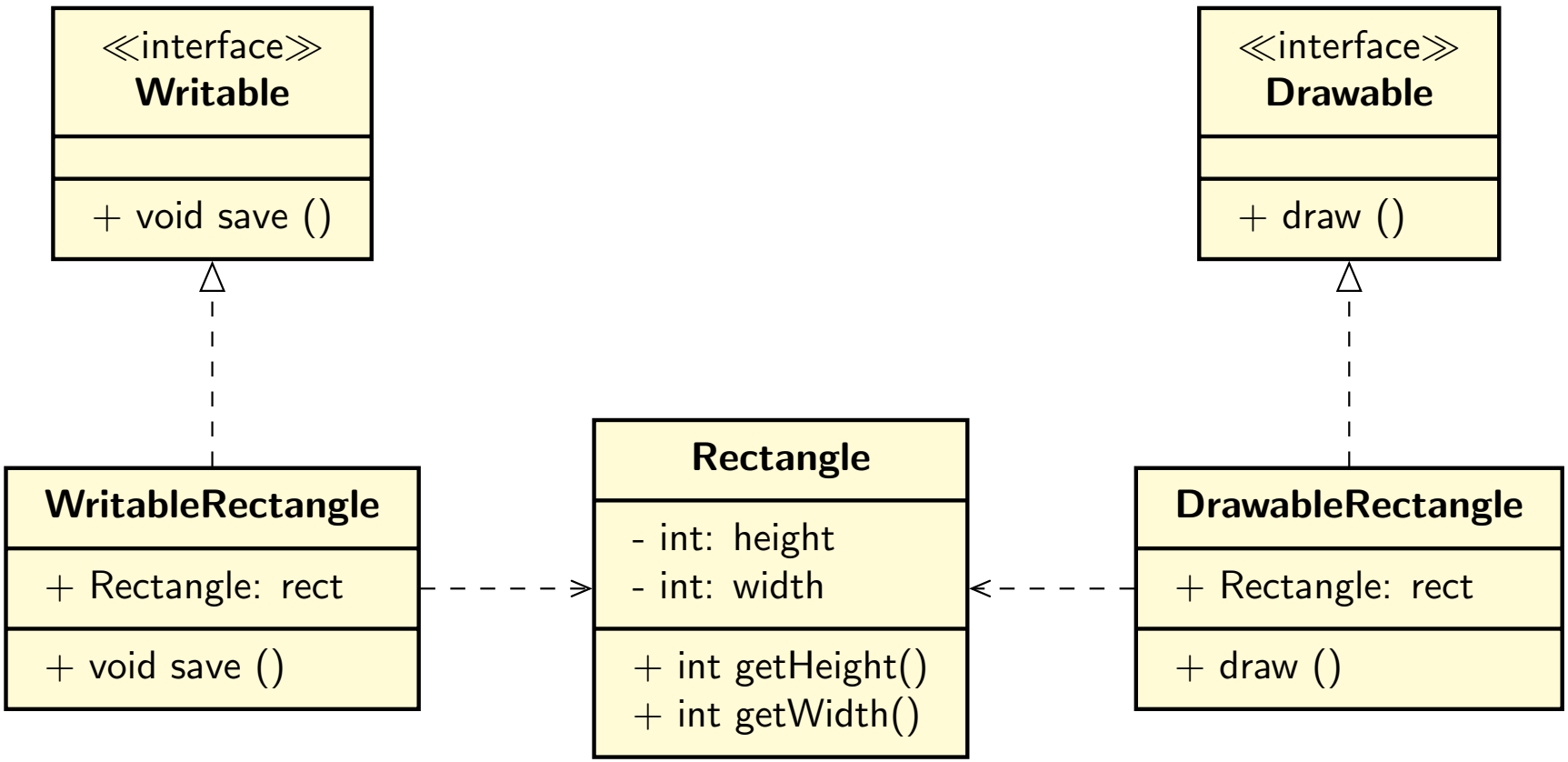
Single responsibility principle: Example

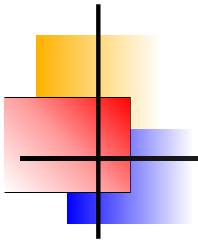


Question

Do you think that this design follows the Single Responsibility Principle?

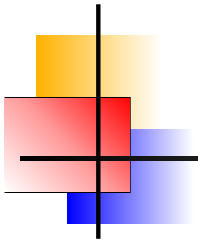
Single responsibility principle: Example





Open/closed principle

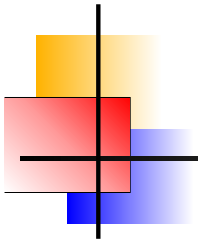
Software entities should be open for extension, but closed for modification



Open/closed principle

Software entities should be open for extension, but closed for modification

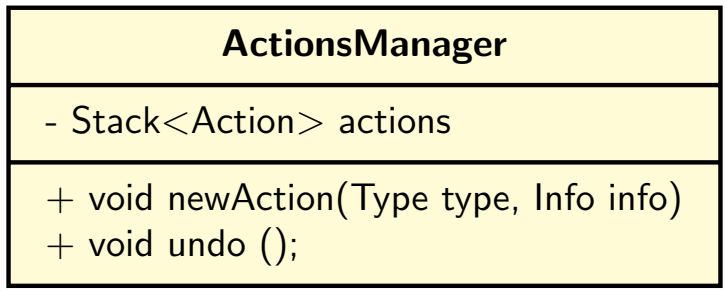
- The behaviour of a module should be able to be extended but not modified. Is that possible?
- The answer is yes and the key point is the use of **abstractions**
- Try to create abstractions that are fixed and yet represent an unbounded number of possible behaviors
- And follow some good practices to avoid modifications in the behaviour of your objects:
  - Make all attributes private (encapsulation)
  - Don't use global variables (except for constants)
  - Avoid the use of run-time type verification (`instanceof`)



Open/closed principle: Example

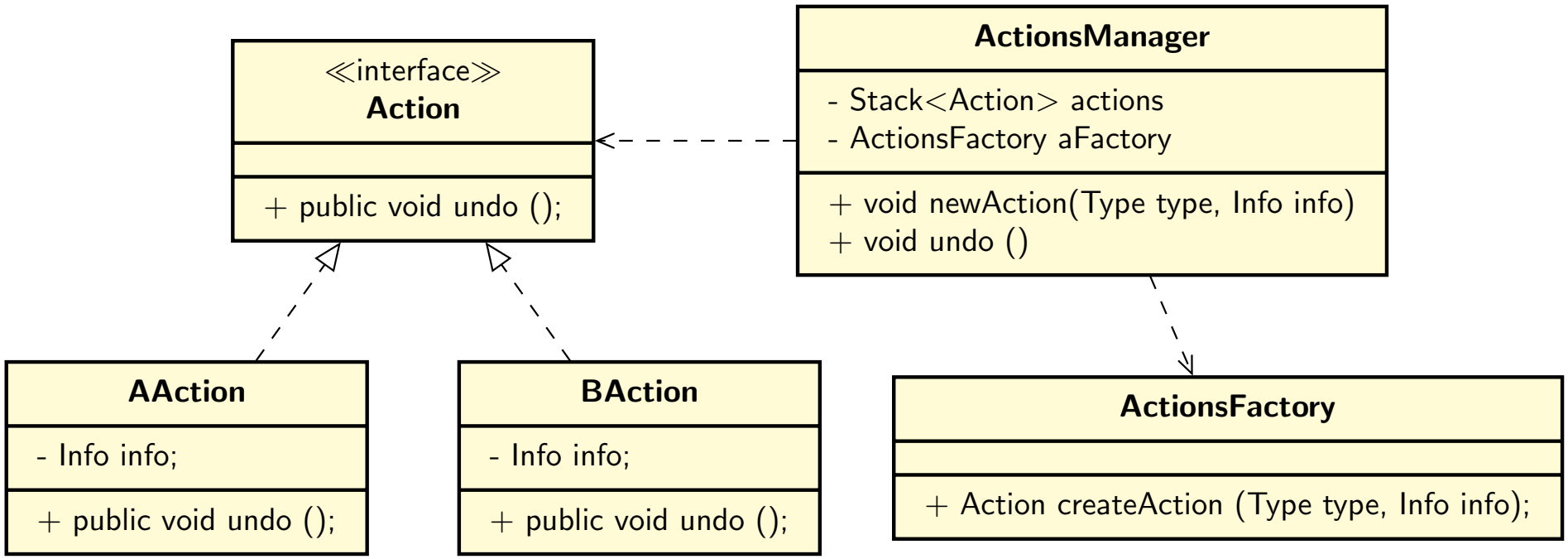
ActionsManager
- Stack<Action> actions
+ void newAction(Type type, Info info) + void undo ();

Open/closed principle: Example



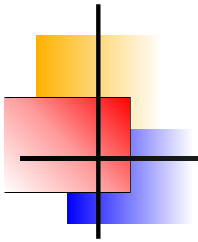

```
void newAction (Type type, Info info) {
  switch (type) {
  case 'A':
    actions.push(new AAction(info));
  case 'B':
    actions.push(new BAction(info));

  .
  .
  .
}
void undo () {
  Action a = actions.pop();
  if (a instanceof AAction)
    // undo action AAction
  else if (b instanceof BAction) {
    // undo action AAction

  .
  .
  .
}
```

Open/closed principle: Example



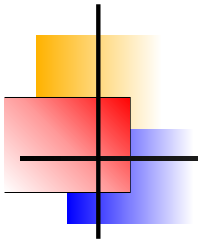

```
void newAction (Type type, Info info) {
  actions.push(aFactory.create(type,info));
}
void undo () {
  actions.pop().undo();
}
```

```
void createAction (Type type,
     Info info) {
  switch (type) {
  case 'A':
   return new AAction(info);
  case 'B':
   return new BAction(info);
  .
  .
  .
}
```



Liskov substitution principle

Methods that use pointers of references to base classes must be able to use objects of derived classes without knowing it

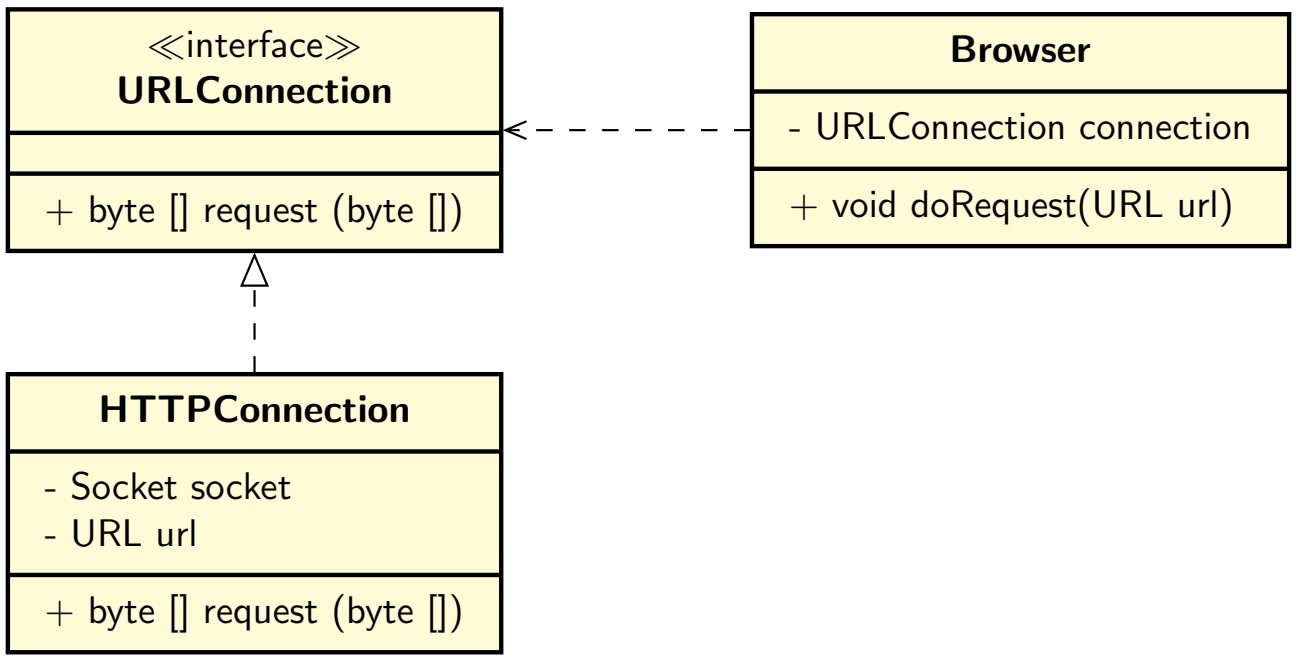


Liskov substitution principle

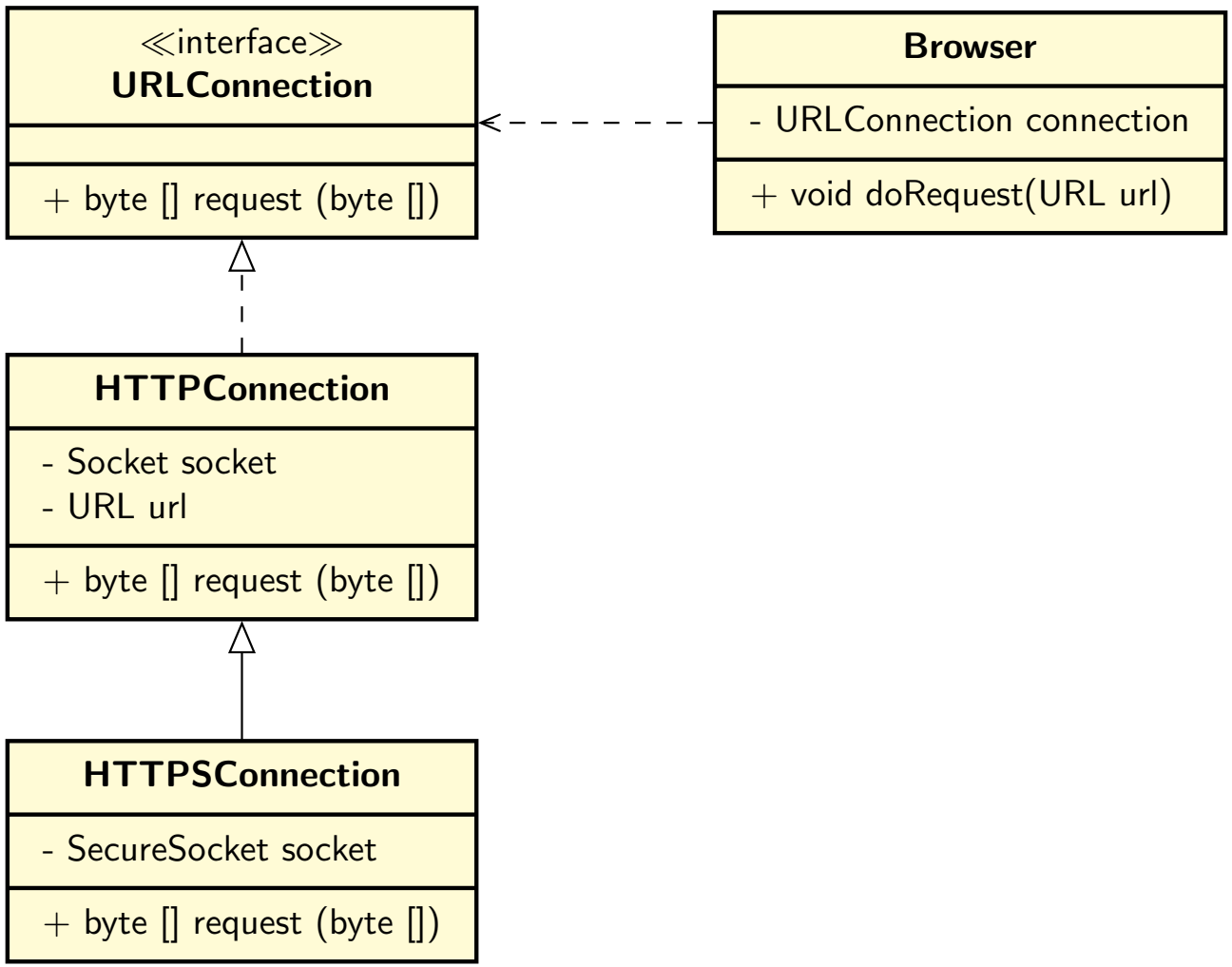
Methods that use pointers of references to base classes must be able to use objects of derived classes without knowing it

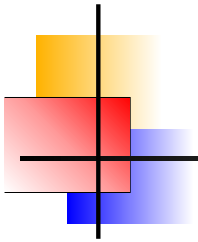
- In essence, it says that a class must be interchangeable by any of its subclasses
- Some properties must hold when implementing subclasses:
  - *Preconditions* cannot be strengthened in a subclass
  - *Postconditions* cannot be weakened in a subclass
  - *Invariants* of the class must be preserved in a subclass

Liskov substitution principle: Example



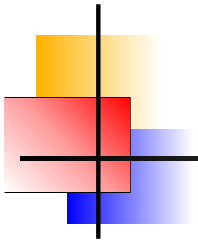
Liskov substitution principle: Example





Interface segregation principle

Clients should not be forced to depend on methods they do not use

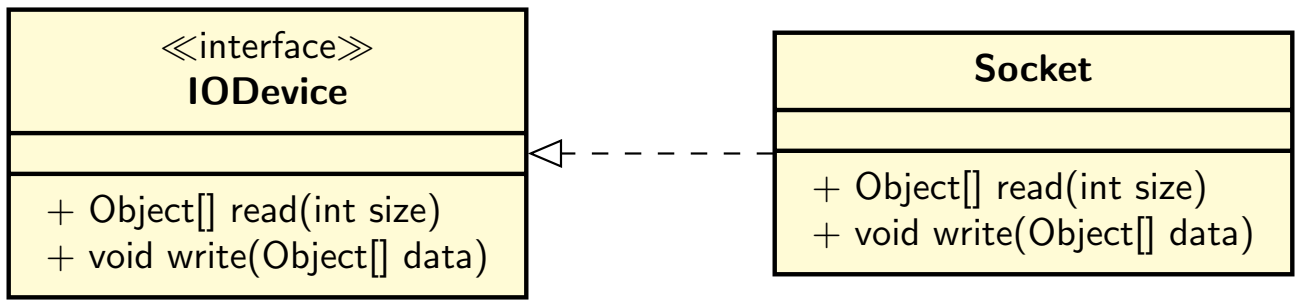


Interface segregation principle

Clients should not be forced to depend on methods they do not use

- Many client-specific interfaces are better than one general-purpose interface
- Big interfaces might lead to high coupling
- We should have different, smaller and specific interfaces for each (set of) clients

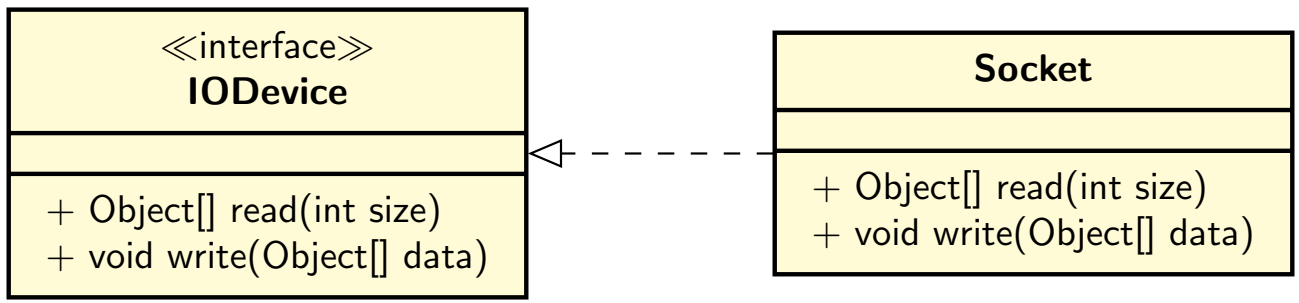
Interface segregation principle: Example



Question

Do you think that this design follows the Interface Segregation Principle?

Interface segregation principle: Example



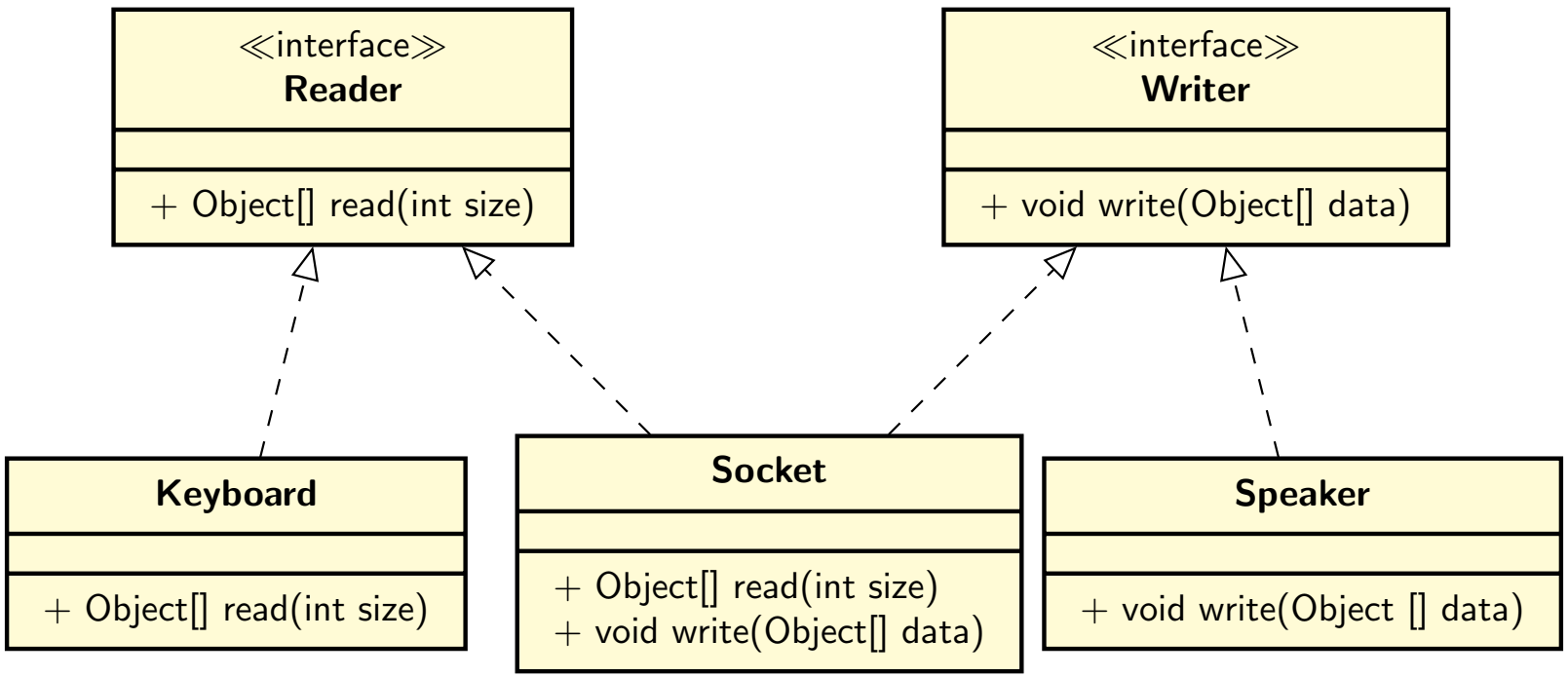
Question

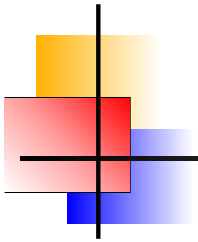
Do you think that this design follows the Interface Segregation Principle?

Question

What happens if we would like to add a class to model a *Keyboard* or a *Speaker*?

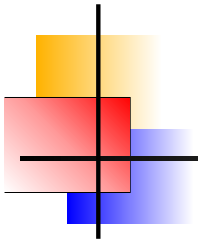
Interface segregation principle: Example





Dependency Inversion Principle

1. *High level modules should not depend on low level modules; both should depend on abstractions*
2. *Abstractions should not depend on details. Details should depend upon abstractions*

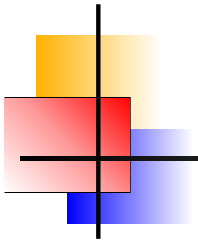


Dependency Inversion Principle

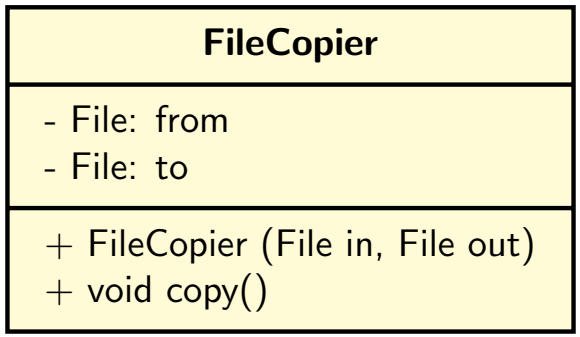
1. *High level modules should not depend on low level modules; both should depend on abstractions*
2. *Abstractions should not depend on details. Details should depend upon abstractions*

- High level modules should depend on interfaces, ignoring the technical details of the implementations
- It achieves loosely coupling as each of its components has, or makes use of, little or no knowledge of the definitions of other separate components
- Following this principle the abstractions and details are isolated from each other:

Program to interfaces, not to implementations!!

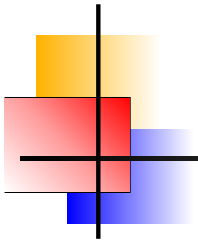


Dependency Inversion Principle: Example

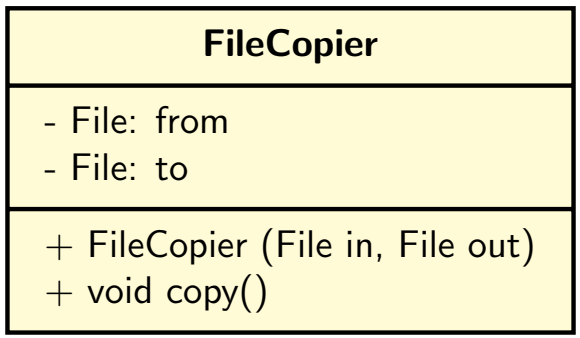


Question

Do you think that this design follows the Dependency Inversion Principle?



Dependency Inversion Principle: Example



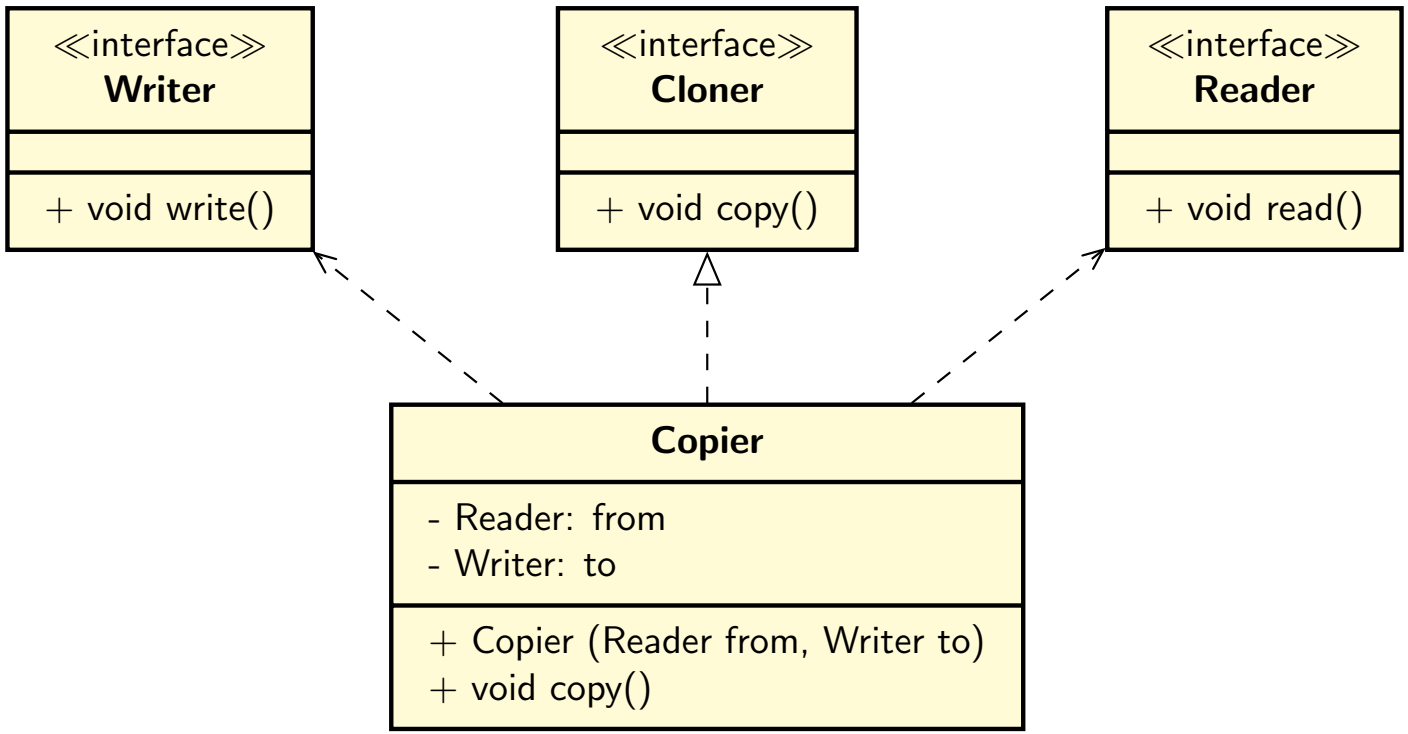
Question

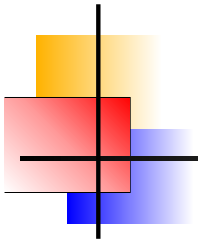
Do you think that this design follows the Dependency Inversion Principle?

Question

What happens if we would like to copy from a *keyboard* to a *file*? or from a *file* to a *socket*?

Dependency Inversion Principle: Example



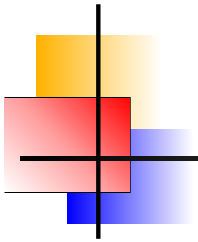


Design Patterns

- We have seen some general principles of a good object-oriented design
- Any design needs different levels of details:
  - System, subsystems, packages, classes, public methods, relationships, ...
  - It is an iterative process: starts from the problem domain, define the message passing between objects, new object appears, new responsibilities to be assigned to objects ...

Deciding what methods belong where, and how the objects should interact, is terribly important and anything but trivial [2]

[2] Craig Larman

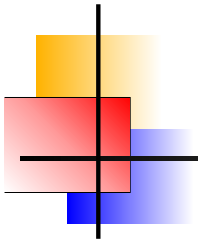


Patterns

- Software design is an *art*, there is no magic formulas, no recipes to reach a *good* design
  - The experience helps to find better designs
- Fortunately, we have a set of advices from experienced developers facing similar problems
- The *design patterns* summarizes this experience

Design Patterns

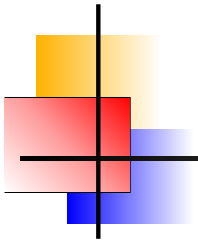
Design Patterns are named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs.



GRASP Patterns

- Craig Larman describes the **GRASP** patterns in his book *Applying UML and Patterns*

  General **R**esponsibility **A**ssignment **S**oftware **P**atterns

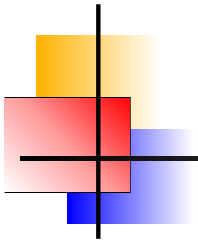


GRASP Patterns

- Craig Larman describes the **GRASP** patterns in his book *Applying UML and Patterns*

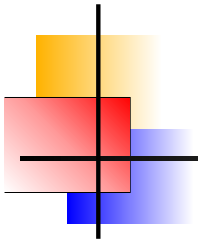
  General **R**esponsibility **A**ssignment **S**oftware **P**atterns

- Are a learning aid to help one understand essential object design
- Aims at applying design reasoning in a methodical, rational and explainable way
- Are based on the assignment of *responsibilities* to objects



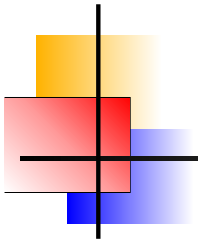
GRASP: Responsibilities

- Responsibilities are related to the obligations of an object in terms of its behavior
- Two kind of responsibilities: **to know** and **to do**
- Knowing responsibilities of an object
  - Which information is responsible to store
  - Which other objects should be known by an object
  - Which information can an object derive or calculate
- Doing responsibilities of an object
  - Which objects should be created by an object
  - Which calculations should be done by an object
  - Which actions should be done or activated by an object



GRASP: 9 patterns

1. Information Expert
2. Creator
3. High Cohesion
4. Low Coupling
5. Controller
6. Polymorphism
7. Indirection
8. Pure Fabrication
9. Protected Variations



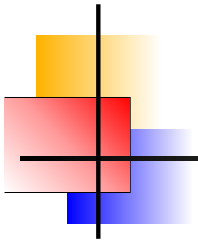
GRASP: Information Expert

Question

What is a general principle of assigning responsibilities to objects?

Assign a responsibility to the information expert, that is, the class that has the information necessary to fulfill the responsibility

- It expresses the common *intuition* that objects do things related to the information they have
- A responsibility often requires information that is spread across different classes of objects: we have multiple experts that collaborate in the task



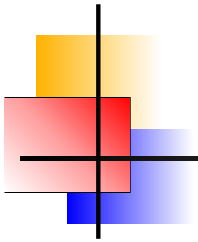
GRASP: Creator

Question

Who should be responsible for creating a new instance of some class?

Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B **aggregates** A objects
- B **contains** A objects
- B **records** instances of A objects
- B **closely** uses A objects
- B has the **initializing data** for creating A



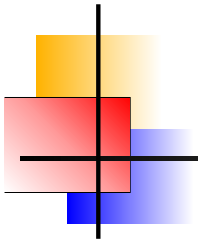
GRASP: Low Coupling

Question

How to support low dependency, low change impact, and increased reuse?

Assign a responsibility so that coupling remains low

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements
- A class with high (or strong) coupling relies on many other classes
  - Changes in related classes force local changes
  - Harder to understand and test in isolation
  - Harder to reuse
- Inheritance should be carefully considered: a subclass strongly depends on its superclass
- High coupling to stable elements and to pervasive elements is seldom a problem (i.e. dependencies on Java Libraries)



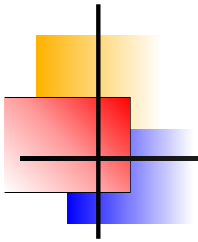
GRASP: High Cohesion

Question

How to keep complexity manageable?

Assign a responsibility so that cohesion remains high

- Cohesion is a measure of how strongly related and focused the responsibilities of an element are
- A class with *low* cohesion does many unrelated things, or does too much work
  - Hard to comprehend, reuse and maintain
- In general, a class with high cohesion has:
  - A relatively small number of methods
  - Highly related functionality
  - Does not do too much work



GRASP: Controller

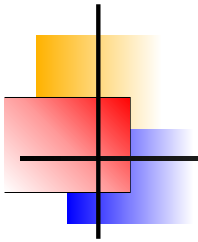
Question

Who should be responsible for handling an input system event?

Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices

1. Represents the overall system, device, or subsystem (*facade* controller)
2. Represents a use case scenario within which the system event occurs called `<UseCaseName>Handler` or `<Use-CaseName>Session` (*use-case* or *session* controller)

- An input system event is an event generated by an external actor (click on a button, menu option, modify a cell, . . .)



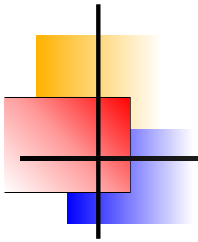
GRASP: Polymorphism

Question

How to handle alternatives based on type? How to create pluggable software components?

When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies

- Again: use **interfaces**! Are useful in two scenarios:
  - Alternatives based on type: *if-then-else* statements require changes when new alternatives appear
  - Pluggable software components: Viewing components in client-server relationships
- Abstract classes can also be considered when there are multiple default implementations in common
- Be realistic about the likelihood of variability



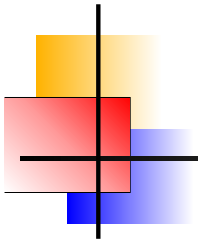
GRASP: Pure Fabrication

Question

Which object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate

Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept

- Frequently classes are concepts that appear in the problem domain
- There are situations in which assigning responsibilities to the object domain classes leads to cohesion or coupling problems
  - Who inserts an object information into a database or showing it in the GUI?
- Designers create specific (pure) objects who are the responsible of executing this actions



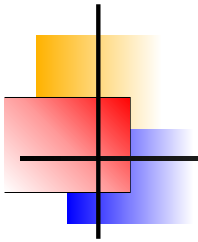
GRASP: Indirection

Question

Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled

- The intermediary creates an *indirection* between the other components



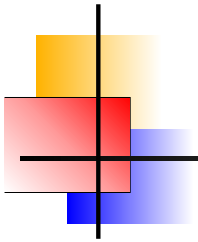
GRASP: Protected Variations

Question

How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

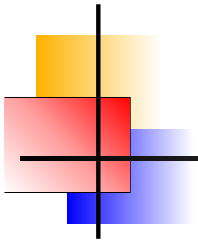
Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them

- As an example we have programs using different front-ends (e.g. different input formats) or using different back-ends (e.g. different output formats)
  - Having an interface for abstracting the different back-ends (or front-ends) would help to add new back-end (or a new front-end)



Gang of Four Patterns

- There are another famous design patterns described in the book: *Design Patterns: Elements of Reusable Object-Oriented Software*
  - The authors of the book are the *Gang of Four*: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- These 23 patterns are divided in three types:
  - Creational: Abstract factory, Builder, Factory, Prototype, Singleton
  - Structural: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
  - Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor

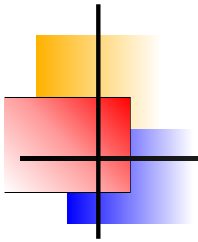


Model View Controller (MVC)

- We have seen the SOLID principles that a good object-oriented design should follow
- We have already seen the GRASP patterns to assign responsibilities to the objects
- Now we will see one of the most used **architectural pattern** for developing applications with graphical user interfaces

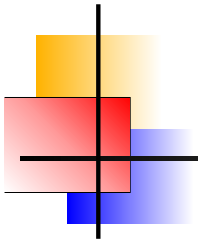
Model - View - Controller

- The MVC was introduced for Smalltalk in the 80's
- MVC is a pattern whose main goal is to separate the business logic from the user interface



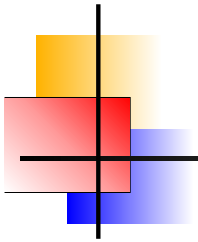
Model View Controller (MVC)

- **Model**
  - Defines the essential properties of the system: the data and the business logic
  - Offers an interface with the operations that the system can do



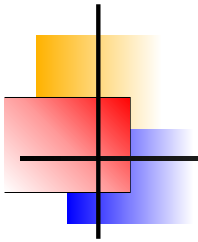
Model View Controller (MVC)

- **Model**
  - Defines the essential properties of the system: the data and the business logic
  - Offers an interface with the operations that the system can do
- **View**
  - Corresponds to the information presentation
  - Captures the user inputs, that is, the events and the information filled by the user



Model View Controller (MVC)

- **Model**
  - Defines the essential properties of the system: the data and the business logic
  - Offers an interface with the operations that the system can do
- **View**
  - Corresponds to the information presentation
  - Captures the user inputs, that is, the events and the information filled by the user
- **Controller**
  - Responds to the user events receiving the inputs (and validating them) from the user and invokes the model to perform some actions
  - Takes the response from the model and passes the information to the view
  - Decides which part (or parts) of the view are shown at any moment



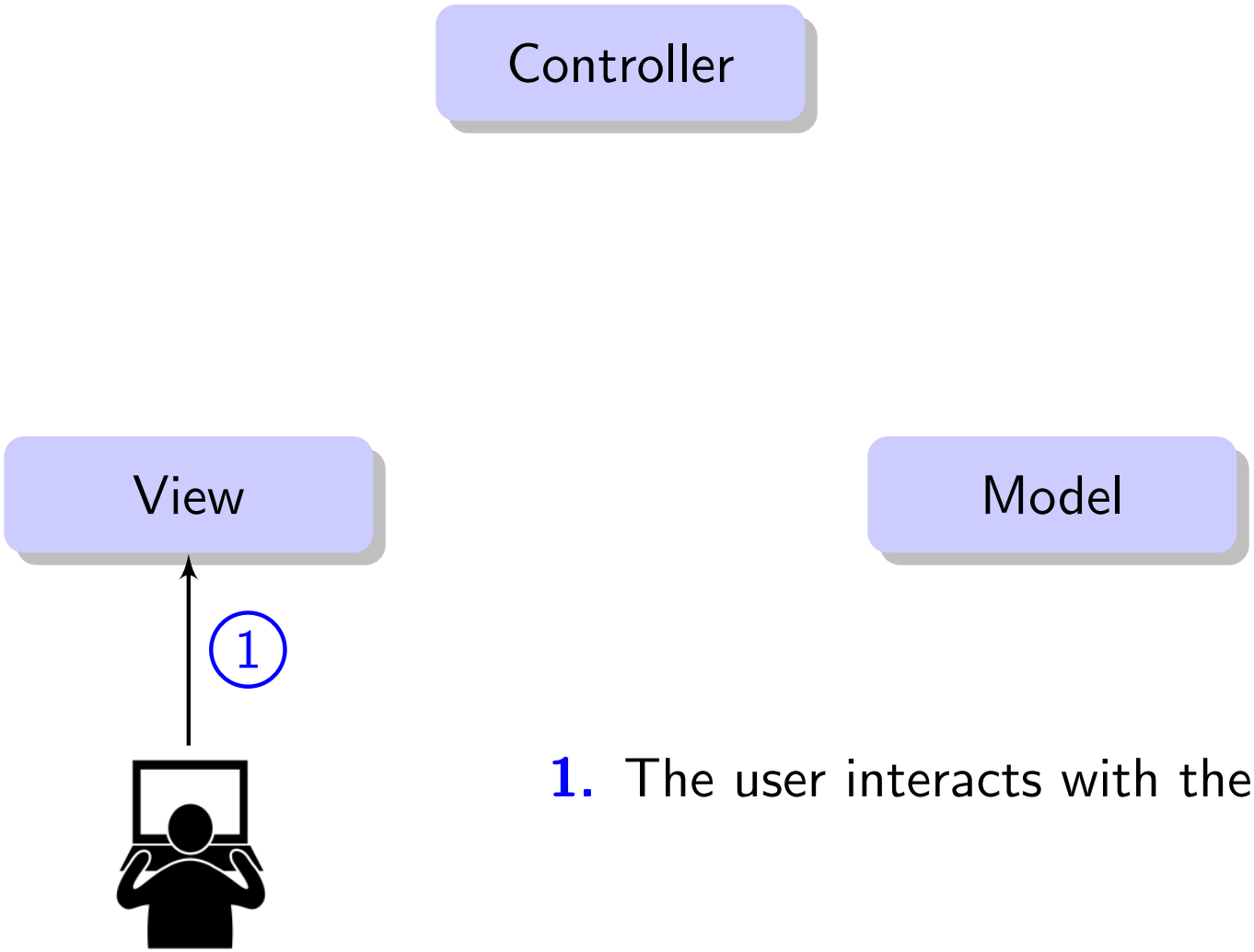
Model View Controller (MVC)

Controller

View

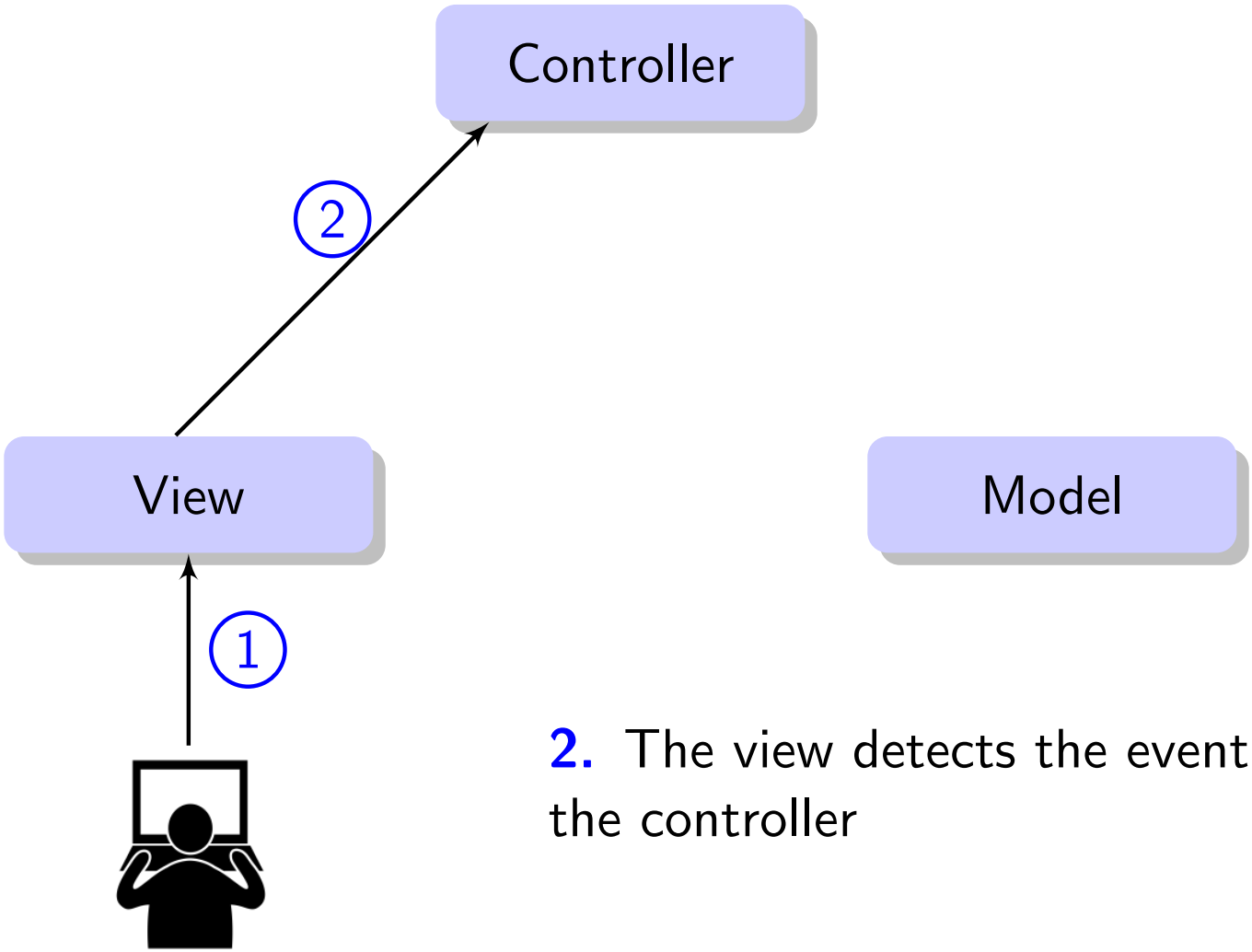
Model

Model View Controller (MVC)



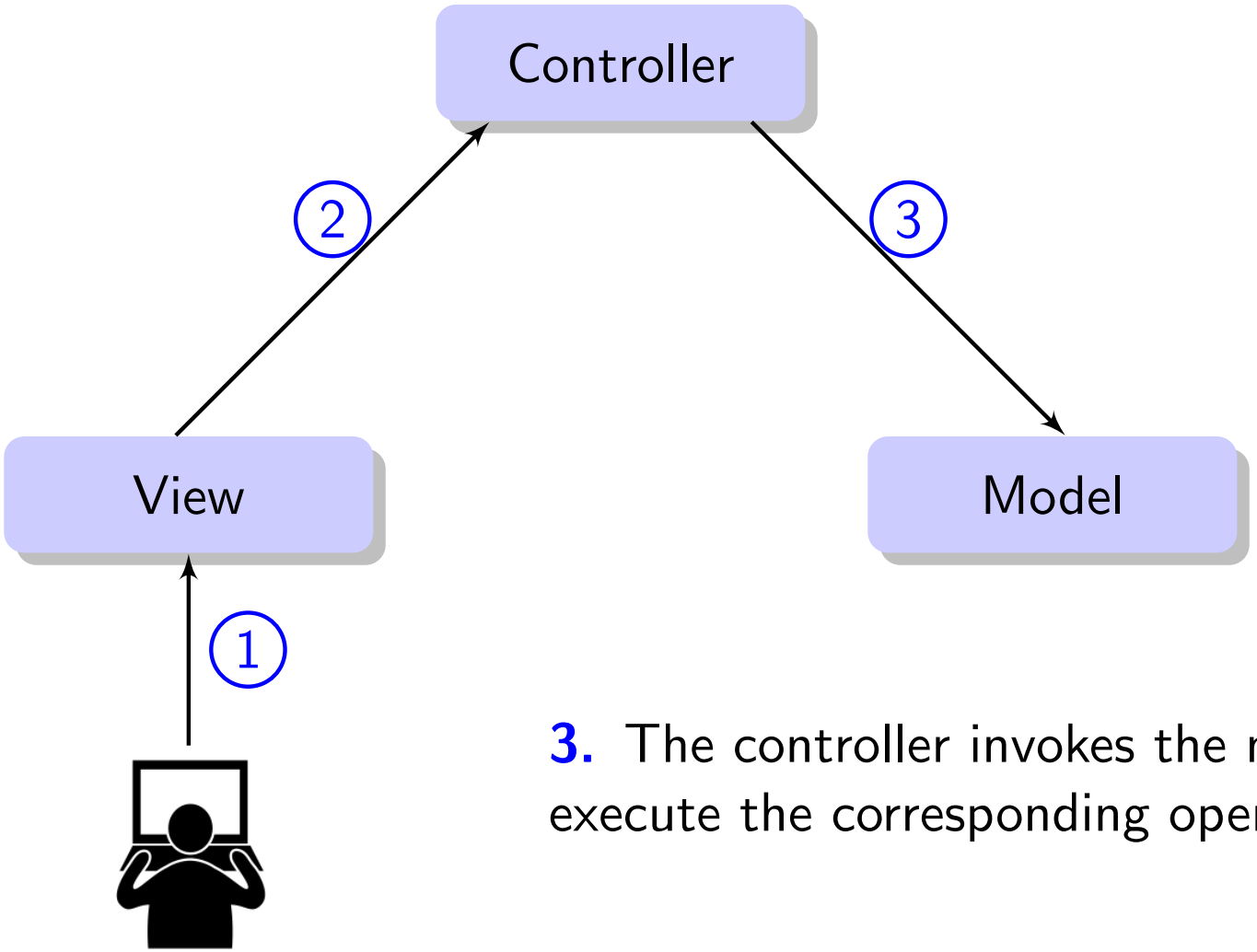
1. The user interacts with the application

Model View Controller (MVC)



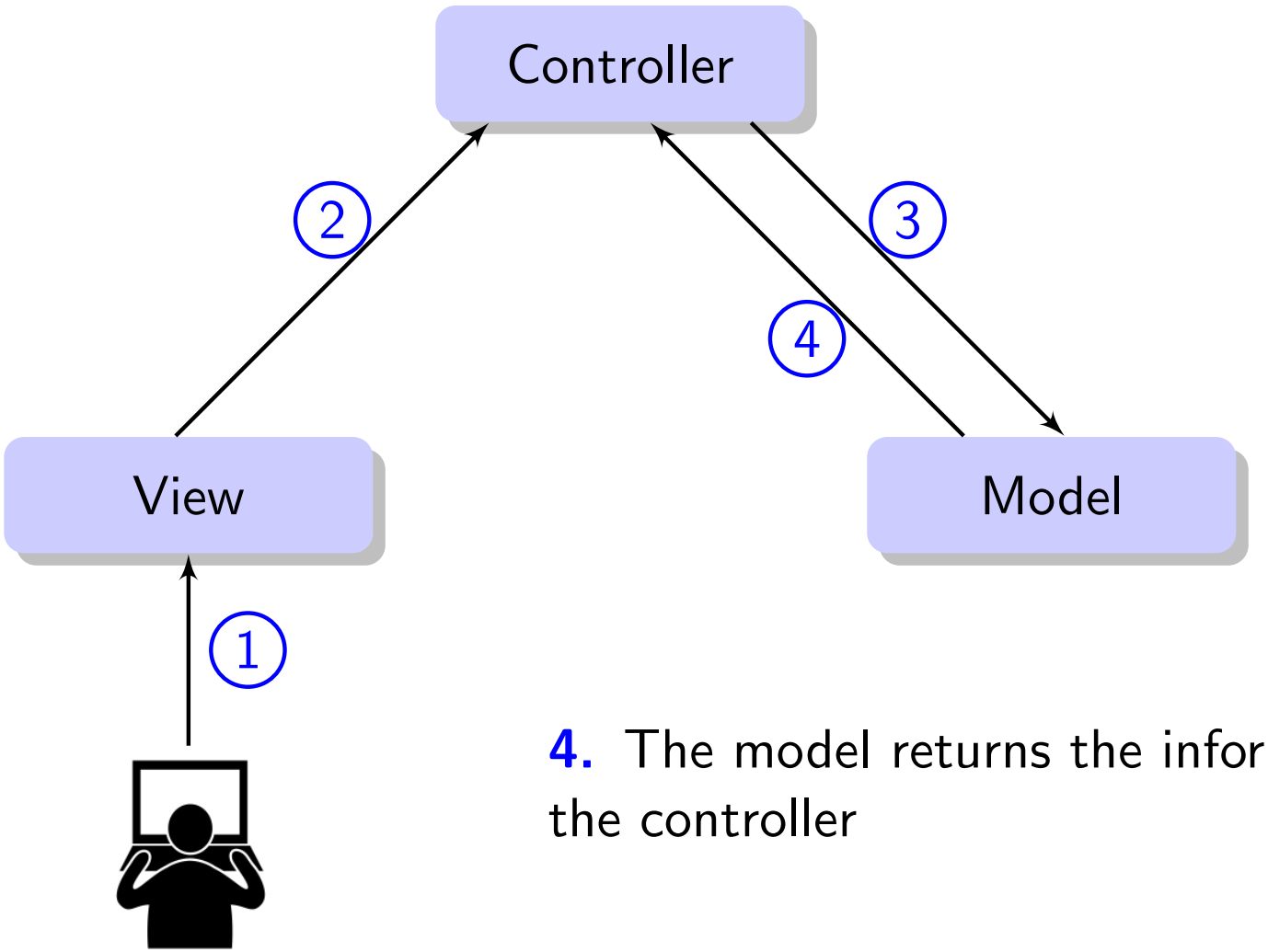
2. The view detects the event and invokes the controller

Model View Controller (MVC)



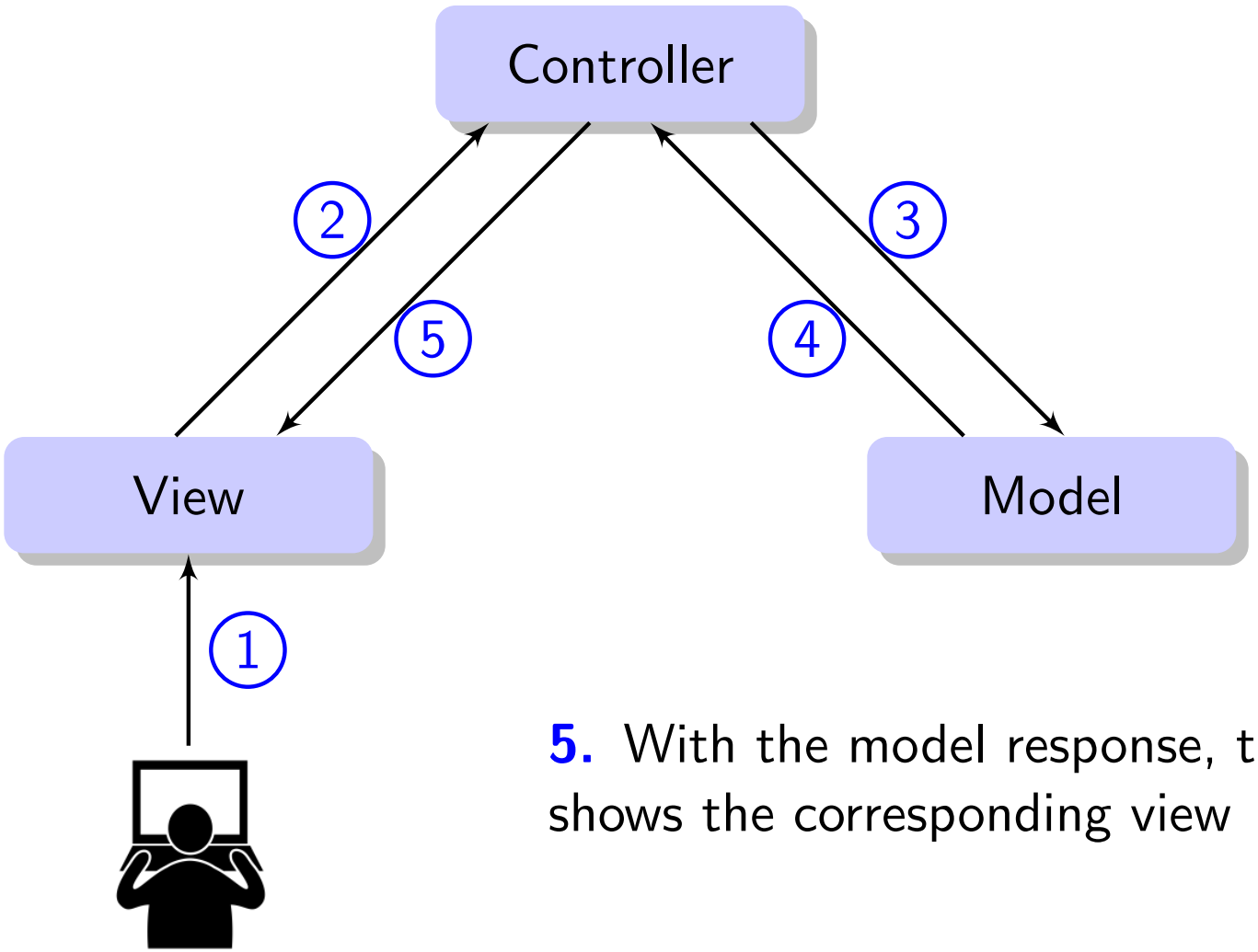
3. The controller invokes the model to execute the corresponding operation

Model View Controller (MVC)



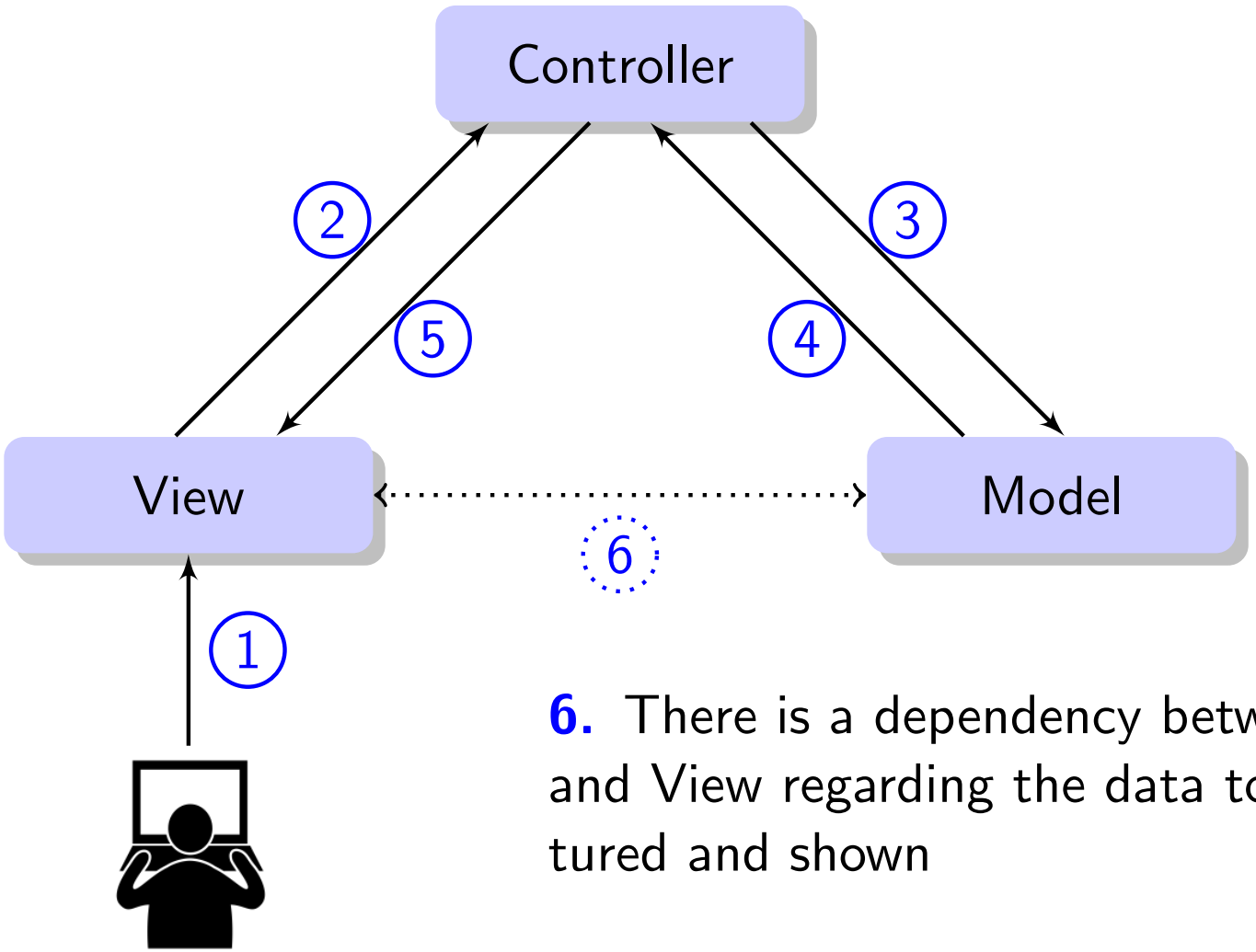
4. The model returns the information to the controller

Model View Controller (MVC)

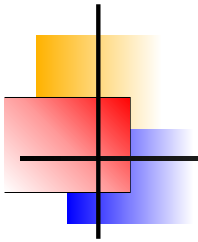


5. With the model response, the controller shows the corresponding view

Model View Controller (MVC)



6. There is a dependency between Model and View regarding the data to be captured and shown



Further Reading

,

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, **Design patterns: Elements of reusable object-oriented software**, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Craig Larman, **Applying uml and patterns: An introduction to object-oriented analysis and design and iterative development (3rd edition)**, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.