Software Engineering

Object Oriented Design Principles

何明昕 HE Mingxin, Max

Send your email to c.max@yeah.net with a subject like: *SE345-Andy: On What …*

Download from c.program@yeah.net /文件中心/网盘/SoftwareEngineering24S

Topics

- SOLID Design Principles
  - https://en.wikipedia.org/wiki/SOLID
- Composition over inheritance (aka Composite reuse principle)
  - https://en.wikipedia.org/wiki/Composition_over_inheritance
- Don't Repeat Yourself - DRY
  - https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
- Inversion of Control - IoC (aka Hollywood Principle)
  - https://en.wikipedia.org/wiki/Inversion_of_control
- You Aren't Gonna Need It - YAGNI
  - https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it
- Law of Demeter - LoD (aka Principle of Least Knowledge)
  - https://en.wikipedia.org/wiki/Law_of_Demeter
- Principle of Least Astonishment - PoLA
  - https://en.wikipedia.org/wiki/Principle_of_least_astonishment
- Minimum Viable Product - MVP
  - http://en.wikipedia.org/wiki/Minimum_viable_product

SOLID Design Principles

Software inevitably changes/evolves over time (maintenance, upgrade)

- **S**ingle responsibility principle (SRP)
  - Every class should have only one reason to be changed
  - If class "A" has two responsibilities, create new classes "B" and "C" to handle each responsibility in isolation, and then compose "A" out of "B" and "C"
- **O**pen/closed principle (OCP)
  - Every class should be open for extension (derivative classes), but closed for modification (fixed interfaces)
  - Put the system parts that are likely to change into implementations (i.e. concrete classes) and define interfaces around the parts that are unlikely to change (e.g. abstract base classes)
- **L**iskov substitution principle (LSP)
  - Every implementation of an interface needs to fully comply with the requirements of this interface (requirements determined by its clients!)
  - Any algorithm that works on the interface, should continue to work for any substitute implementation

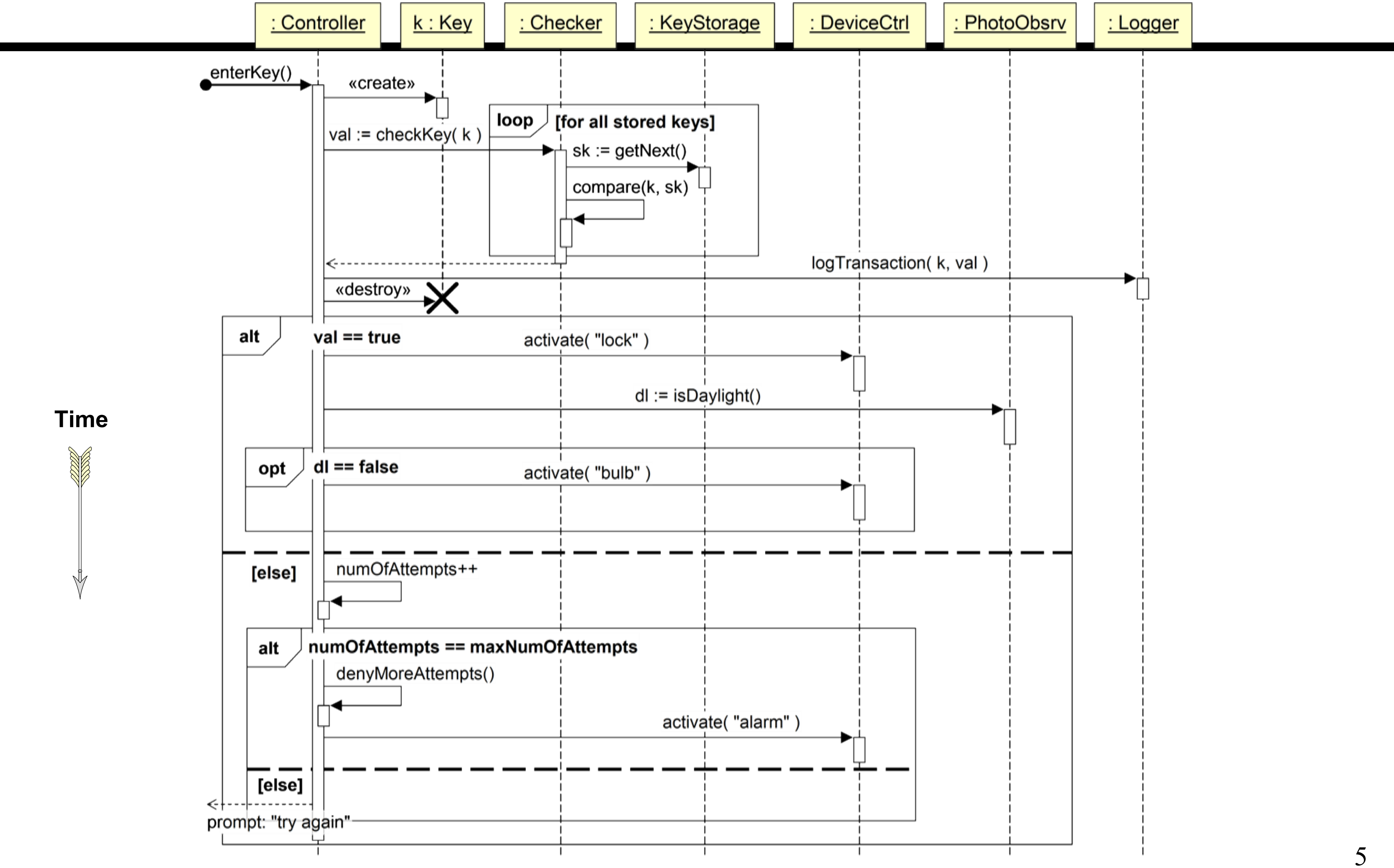
SOLID Design Principles

Software inevitably changes/evolves over time (maintenance, upgrade)

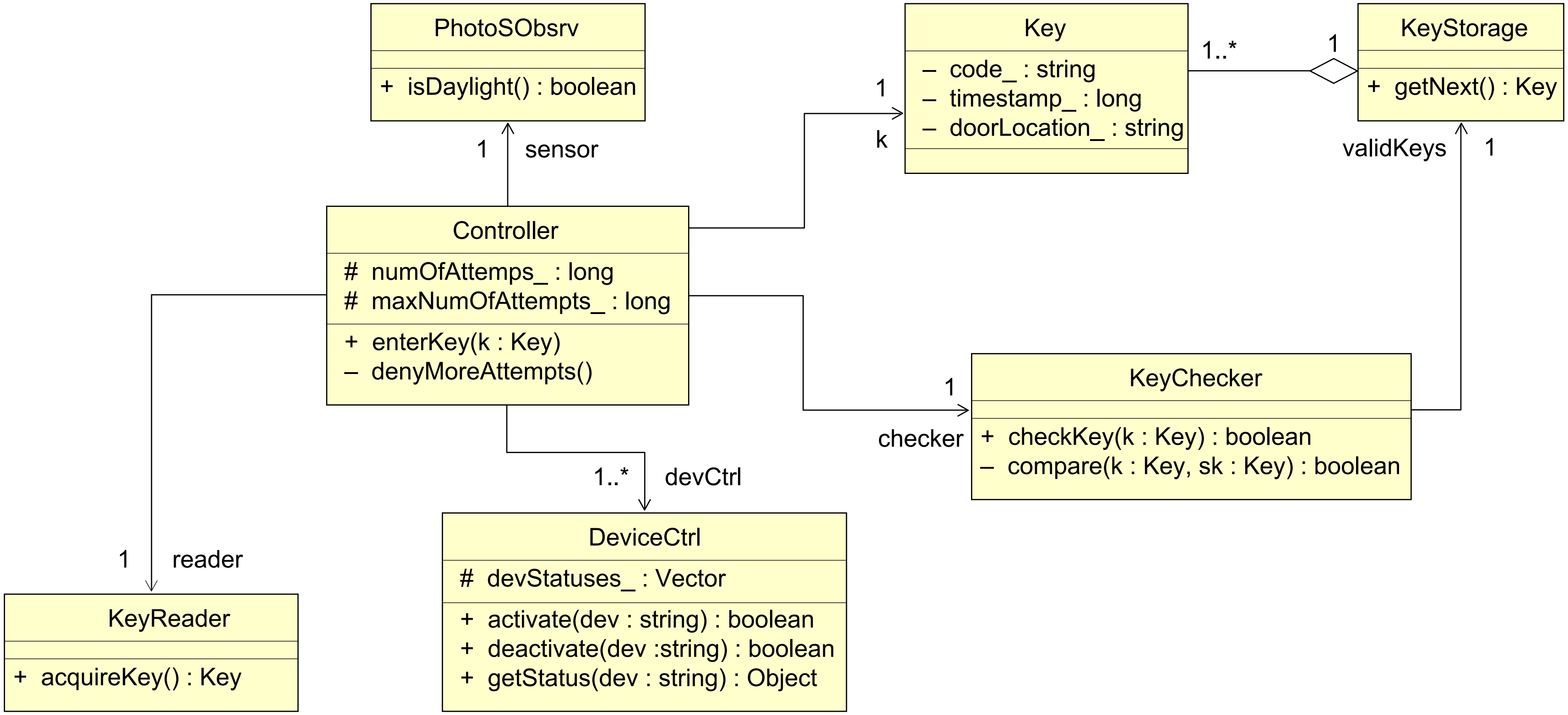
- ❑ **I**nterface segregation principle (ISP)
  - – Keep interfaces as small as possible, to avoid unnecessary dependencies
  - – Ideally, it should be possible to understand any part of the code in isolation, without needing to look up the rest of the system code
- ❑ **D**ependency inversion principle (DIP)
  - – Instead of having concrete implementations communicate directly (and depend on each other), decouple them by formalizing their communication interface as an abstract interface based on the needs of the higher-level class

… is this a good design?

Unlock Use Case



Class Diagram



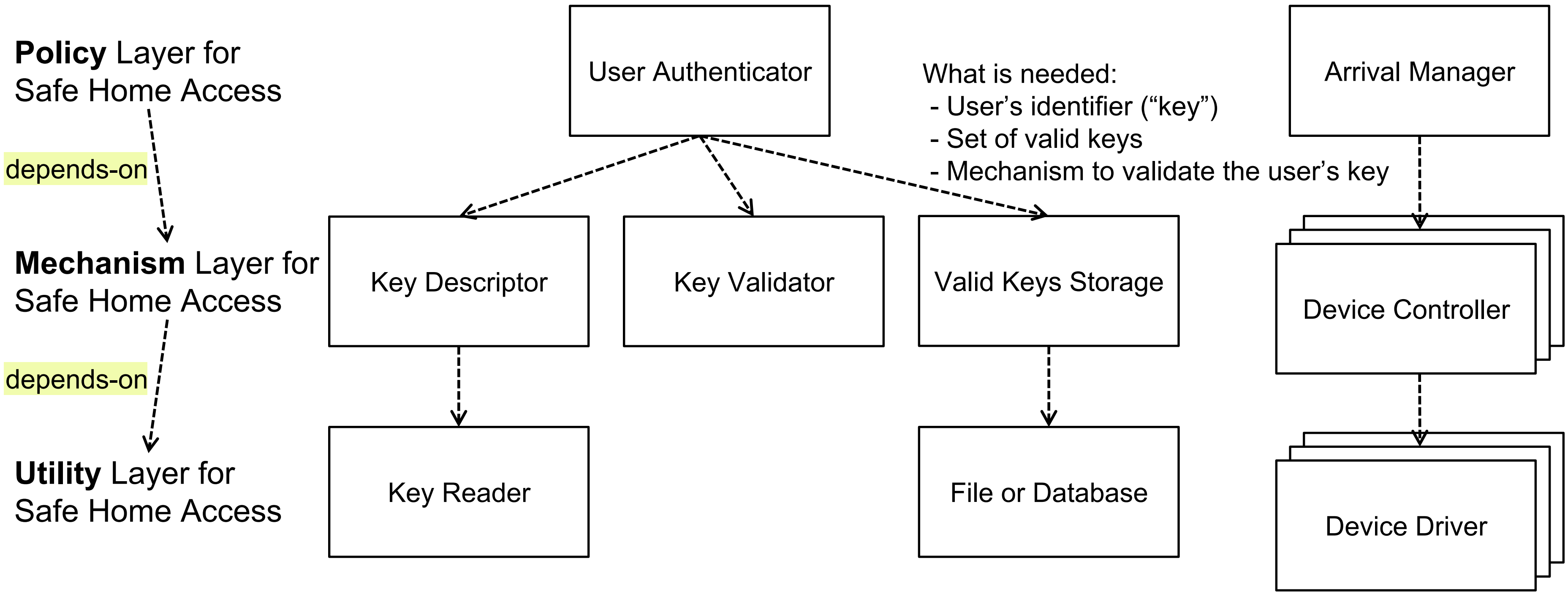
Purpose of Design Principles

- **Principles** are used to *diagnose* problems with designs
- **Patterns** are used to *address* the problems

Examples of Software Change

- What could change in the home access system: means of user authentication and the controlled devices. These sources of change are independent, so one should not affect the code for the other
- Scenario #1: replace numeric-code based keys with magnetic card or RFID chip
  - What part of the system needs to be replaced?
  - What is the "interface" seen by the rest of the system that needs to remain invariant?
- Scenario #2: same as above
  - Policy for handling dictionary attacks becomes inadequate: what kind of attack can be mounted by an adversary in case of magnetic or RFID codes?
  - What policy (or policies) are appropriate for this scenario?

Dependencies for Ordinary Layered Style



Policy Layer for Safe Home Access

depends-on

Mechanism Layer for Safe Home Access

depends-on

Utility Layer for Safe Home Access

User Authenticator

Key Descriptor

Key Validator

Valid Keys Storage

Key Reader

File or Database

What is needed:
- User's identifier ("key")
- Set of valid keys
- Mechanism to validate the user's key

Arrival Manager

Device Controller

Device Driver

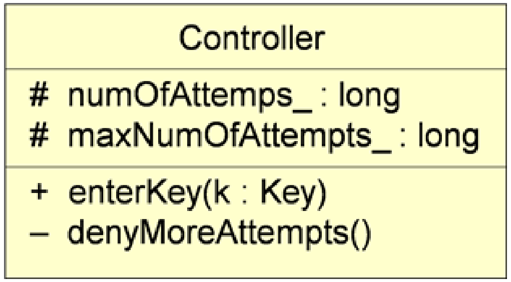
- Note the dependencies from *top to bottom*
- That is, higher-level modules depend on the lower-level modules
- ➔ Any changes in lower-level modules may propagate up to the higher levels

Need *Inverted* Dependencies

- Low-level modules are more likely to change
- High-level modules are more likely to remain stable: they implement the business policies, which is the purpose of the system and is unlikely to change
- Dependency Inversion Principle (DIP)

Example Business Policy & Mechanism

```
IF key $\in$ ValidKeys THEN disarm lock and turn lights on
ELSE
    increment failed-attempts-counter
    IF failed-attempts-counter equals maximum number allowed
        THEN block further attempts and raise alarm
```



Controller

# numOfAttemps_ : long
# maxNumOfAttempts_ : long

+ enterKey(k : Key)
– denyMoreAttempts()

Safe Home Access Policy Level

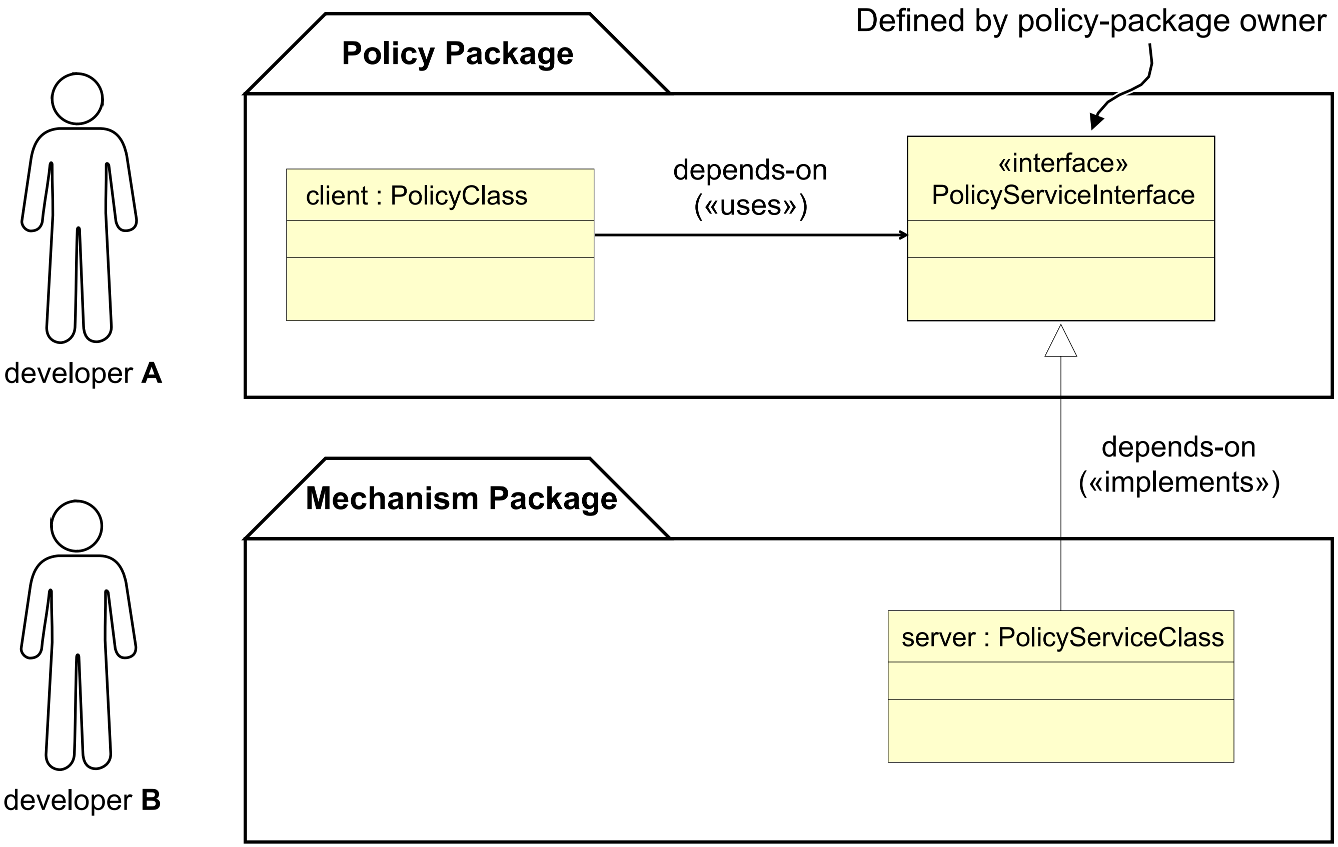
- *Detailed* statement of the problem:
  - Read the numeric code typed-in by the user
  - Validate the code, and
  - Set the voltage high to disarm the lock, turn on the light
- *Abstract* statement of the problem:
  - Acquire the user code
  - Validate the code, and
  - Enable access and assist user’s arrival activties

Dependency Inversion Principle

- Instead of high-level module (policy) depending on low-level module (mechanism/service/utility):
  - High-level module defines its desired interface for the low-level service (i.e., high-level depends on itself-defined interface)
  - Lower-level module depends on (implements) the interface defined by the high-level module
  - → *Dependency inversion*
    (from low to high, instead the opposite)

Dependency Inversion Pattern

Package diagram



❑ Note the dependencies from ***bottom to top***

❑ Both Policy Class and Policy Service Class *depend* on Policy Service Interface but the former *uses* (and *defines*) the interface and the latter *implements* the interface

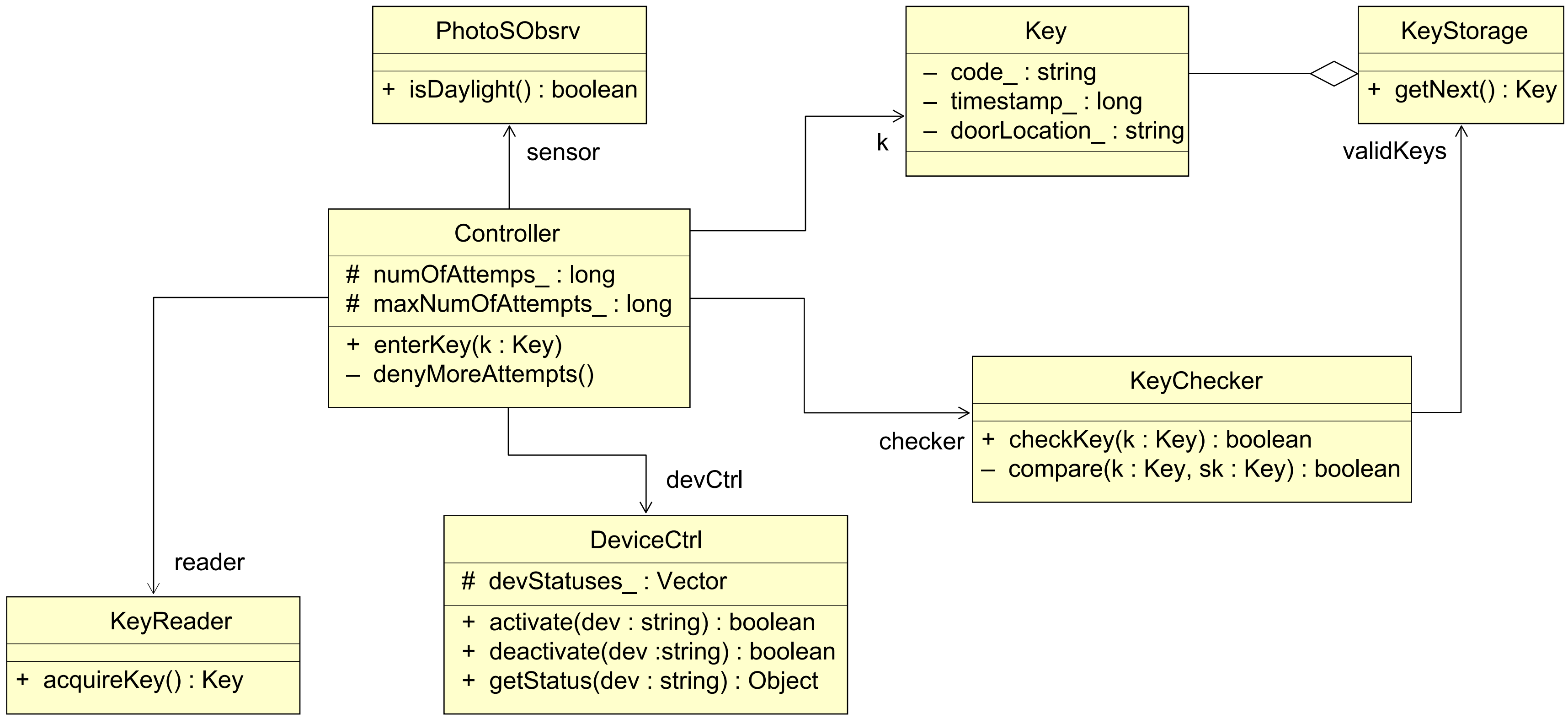
Additional Object: Intruder Detector?

- Based on an invalid key decides an intruder
- However, the notion of a “dictionary attack” may not make sense for non-numeric keys, not acquired from a keypad
- If the “key” is a magnetic-card code, it cannot be assumed that the user made a mistake and should be given another try
  - If a key is transmitted wirelessly, it may be intercepted and reproduced
- What if the “key” is user’s fingerprint or another biometric feature?
  - A biometric identifier could be faked, although also user may have dirty or greasy fingers which would prevent correct fingerprint-based identification or user’s face may not be properly illuminated…
- ➔ Need a new intruder-detection mechanism …

Examples of Software Change

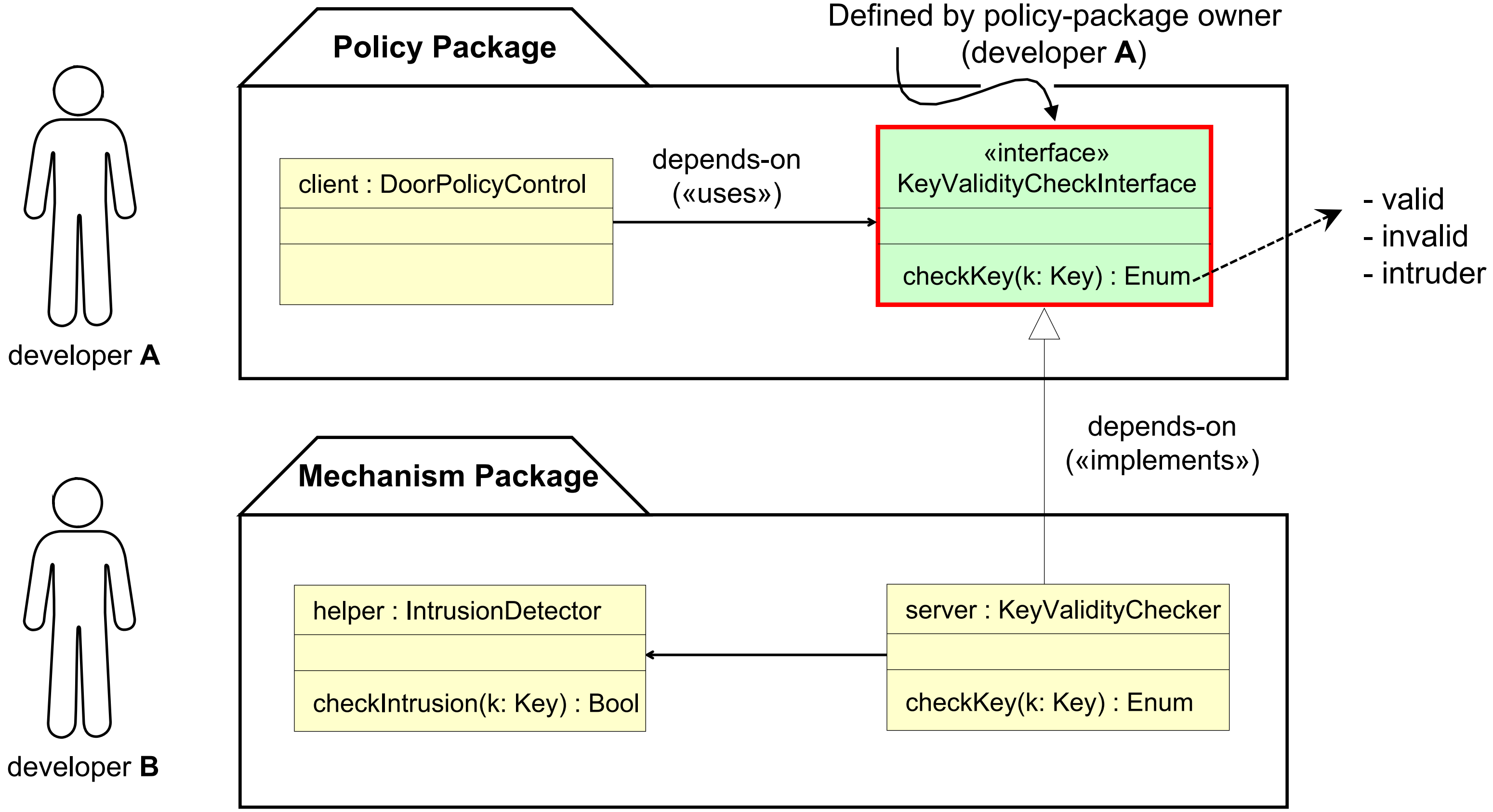
- What could change in the home access system: means of user authentication and the controlled devices. These sources of change are independent, so one should not affect the code for the other
- Scenario #1: replace numeric-code based keys with magnetic card or RFID chip
  - What part of the system needs to be replaced?
  - What is the "interface" seen by the rest of the system that needs to remain invariant?
- Scenario #2: same as above
  - Policy for handling dictionary attacks becomes inadequate: what kind of attack can be mounted by an adversary in case of magnetic or RFID codes?
  - What policy (or policies) are appropriate for this scenario?

… where is the “interface” when the ID mechanism changes?



Dependency Inversion Pattern

Package diagram



- ❑ Previous solution: Intrusion detection mechanism was entangled with door policy and hidden
- ❑ Current solution: Intrusion detection ***mechanism*** moved from Policy Level to Mechanism Level

Liskov Substitution Principle

- Every implementation of an interface needs to fully comply with the requirements of this interface
- Any algorithm that works on the interface, should continue to work for any substitute implementation

Liskov Substitution Principle

- This principle is often misinterpreted that the two objects are equivalent if they provide the same API
  - However, the API is a program-level interface that does not capture the ***use*** of the object’s methods, attributes, nor its dependency graph
  - The API cannot represent the preconditions, postconditions, invariants, etc.
  - An object’s LSP substitute must ensure that the physical aspects for resource footprint do not affect the rest of the system
- Dependency graph of the substituted object must be completely replaced with the dependency graph of its substitute object