Software Engineering

LECTURE 17: Design Patterns
Publisher-Subscriber

Ivan Marsic
Rutgers University

Topics

- Software Design Patterns
  – What & Why
- Example Pattern:
  Publisher-Subscriber (a.k.a. Observer)

What Developers Do With Software

(besides development)

- Understand (existing software)
- Maintain (fix bugs)
- Upgrade (add new features)

The Power of Patterns

CN NIB MAP PLE

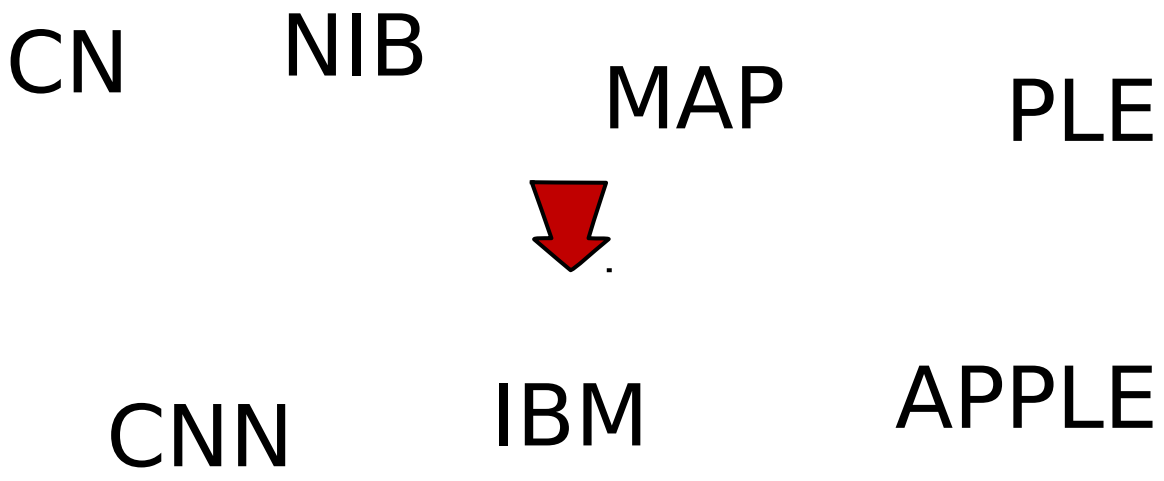
The Power of Patterns

The Power of Patterns

CNN IBM APPLE

The Power of Patterns

The Power of Patterns



- There are many other potential “patterns”, but these somehow ***succeeded*** to became widely known.
- A **pattern** is a noticeable regularity in the world or in a manmade design.
  The elements of a pattern repeat in a predictable manner.

Software Design Patterns

- Design Patterns help anticipate software change
- Change is needed to keep up with the reality
  - Change may be triggered by changing business environment or by deciding to refactor classes that are deemed potentially problematic e.g., using quality-screening metrics (see Chapter 4)
- Change may have bad consequences when there are **unrelated reasons** to change a software module/class
  - Unrelated reasons are usually because of unrelated responsibilities
  - Change in code implementing one responsibility can unintentionally lead to faults in code for another responsibility
- Another target is complex conditional logic (If-Then-Else statements, etc.)

What May Change & How

- What changes in real world are business rules → customer requests changes in software [Change in-the-large]
- Change in-the-small: changes in object responsibilities towards other objects
  - Number of responsibilities
  - Data type, or method signature
  - Business rules
  - Conditions for provision/fulfillment of responsibilities/services
- Sometimes change (in-the-small) is regular (follows simple rules), such as new object state defined, or another object needs to be notified about something ⇒ "patterns"

Object Responsibilities

(toward other objects)

- **Knowing** something (memorization of data or object attributes)
- **Doing** something on its own (computation programmed in a “method”)
  - **Business rules** for implementing business policies and procedures are a special case
- **Calling** methods on dependent objects (communication by sending messages)
  - **Calling constructor methods**; this is a special case because the caller must know the appropriate parameters for initialization of the new object.

Example Patterns for Tackling Responsibilities

- Delegating “knowing” and associated “doing” responsibilities to new objects (*State*)
- Delegating “calling” responsibilities (*Command*, *Publisher-Subscriber*)
- Delegating non-essential “doing” responsibilities (when the key doing responsibility is incrementally enhanced with loosely-related capabilities) (*Decorator*)

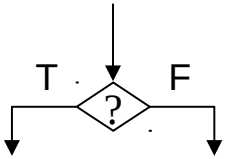
Design patterns provide systematic, tried-and-tested, heuristics for subdividing and refining object responsibilities, instead of arbitrary, ad-hoc solutions.

Key Issues

- When a pattern is needed/applicable?
- How to measure if a pattern-based solution is better?
- When to avoid patterns because may make things worse?
- ← All of the above should be answered by comparing object responsibilities before/after a pattern is applied

Publisher-Subscriber Pattern

- A.k.a. “Observer”
- Disassociates unrelated responsibilities
  - Decrease coupling, increase cohesion
- Helps simplify/remove complex conditional logic and allow seamless future adding of new cases
- Based on “Indirect Communication”



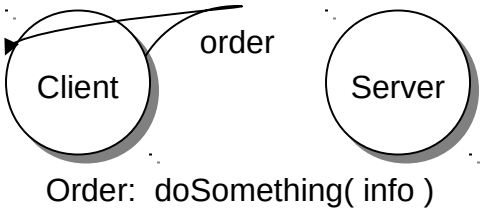
Like many other design patterns

Request- vs. Event-Based Comm.

Issues:

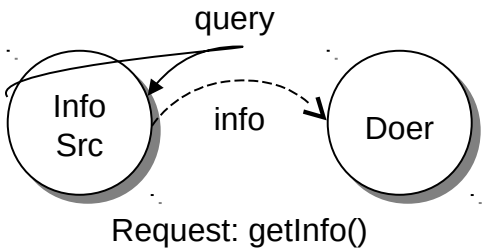
- Who creates the client-server dependency
- Who decides when to call the dependency (server/doer)
- Who is responsible for code changes or new dependencies

Client knows who to call, when, and with what parameters:



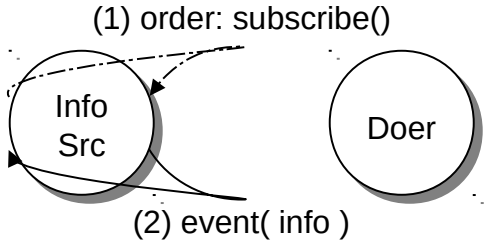
(a)

Doer (server) keeps asking and receives information when available:



(b)

Doer (in advance of need) expresses interest in information and waits to be notified when available:



(c)

(tight coupling)

Direct Communication

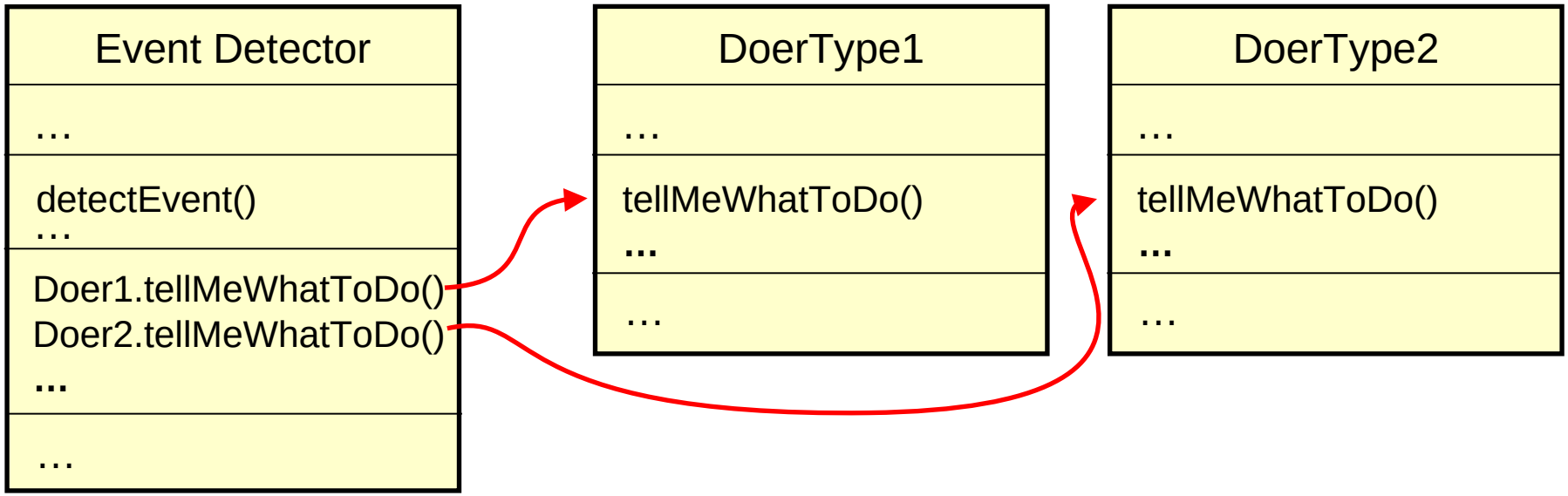
Directly ordering the server what to do or demanding information from a source

(loose coupling)

Indirect Comm.

Indirectly notifying (via event dispatch) the registered doers to do their work (unknown to the event publisher

“Before” == A Scenario Suitable for Applying the Pub-Sub Pattern



Responsibilities

Doing:

- Detect events

Calling dependencies:

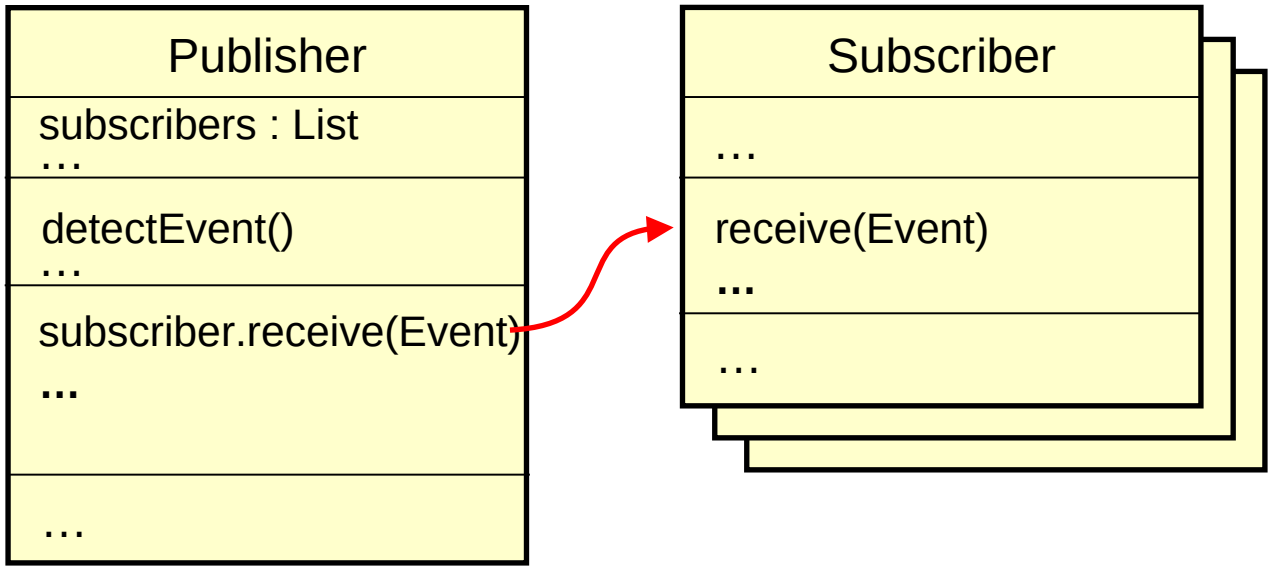
- Tell Doer-1 what to do
- Tell Doer-2 what to do

unrelated!

⇒ unrelated reasons to change the Event Detector:

- When event detection needs to change or extend
- When new doer types need to be told what to do

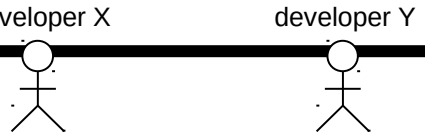
“After” == Responsibilities After Applying the Pub-Sub Pattern



Unrelated responsibilities of the Event Detector (now Publisher) are dissociated:

- When event detection needs to change or extend → change Publisher
- When new doer types need to be added → add an new Subscriber type
  (Subscribers need not be told what to do – they know what to do when a given event occurs!

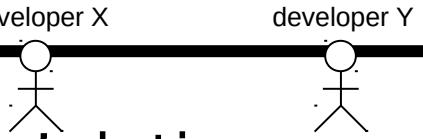
Labor Division (1)



- Before:
  Class A (and developer X) is affected by anything related to class B

Class	A	B
Who develops	developer X	developer Y
Who knows which methods of B to call, when, how	developer X	—
Who makes changes when classes of type B modified or new sub-types added	developer X	—

Labor Division (2)



- After: Separation of developers' duties
- Developer **Y** is completely responsible for class **B**

Class	A	B
Who develops	developer X	developer Y
Who knows which methods of B to call, when, how	—	developer Y
Who makes changes when classes of type B modified or new sub-types added	—	developer Y

Publisher-Subscriber Pattern

- Focused on detecting & dispatching events
  - The focus in on the “Publisher” object and the “environment” that it is observing for “events”
  - Example key events in safe home access case study:
    - Entered key is valid
    - Entered key is invalid
- Instead of making decisions & issuing orders to Doers/Subscribers
  - Notify doers/subscribers when the information/event of their interest becomes available, without knowing what for or how this information will be used
- Works in both directions:
  - Client-side events (user interaction by the initiating actor)
  - Server-side events (participating-actors/“environment” monitoring)

Publisher-Subscriber Pattern

PUBLISHER

- Controller (receives key-code)
  Key Checker (checks key validity, i.e., classifies: valid/invalid)
  - Publishes the classification result to “subscribers”

SUBSCRIBERS

- Lock Control
  Light Control
  Alarm Control
  - Do the work based on the received event

(Assumed, during initialization: Subscribers subscribe for Events
≡ dependency graph construction)

Pub-Sub Pattern

Reports incoming events to subscribers

Publisher

Knowing Responsibilities:
- Knows event source(s)
- Knows interested obj's (subscribers)

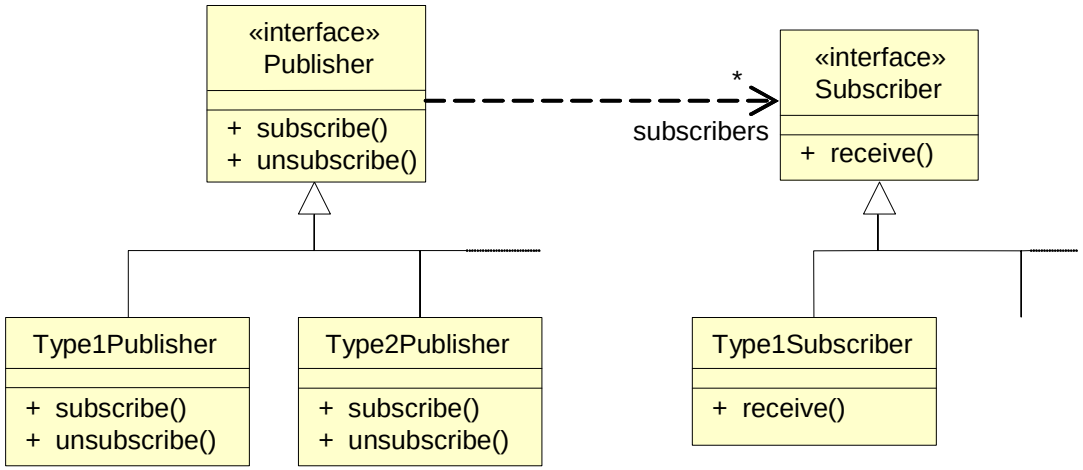
Doing Responsibilities:
- Registers/Unregisters subscribers
- Notifies the subscribers of events

Subscriber

Knowing Responsibilities:
- Knows event types of interest
- Knows publisher(s)

Doing Responsibilities:
- Registers/Unregisters with publishers
- Processes received event notifications

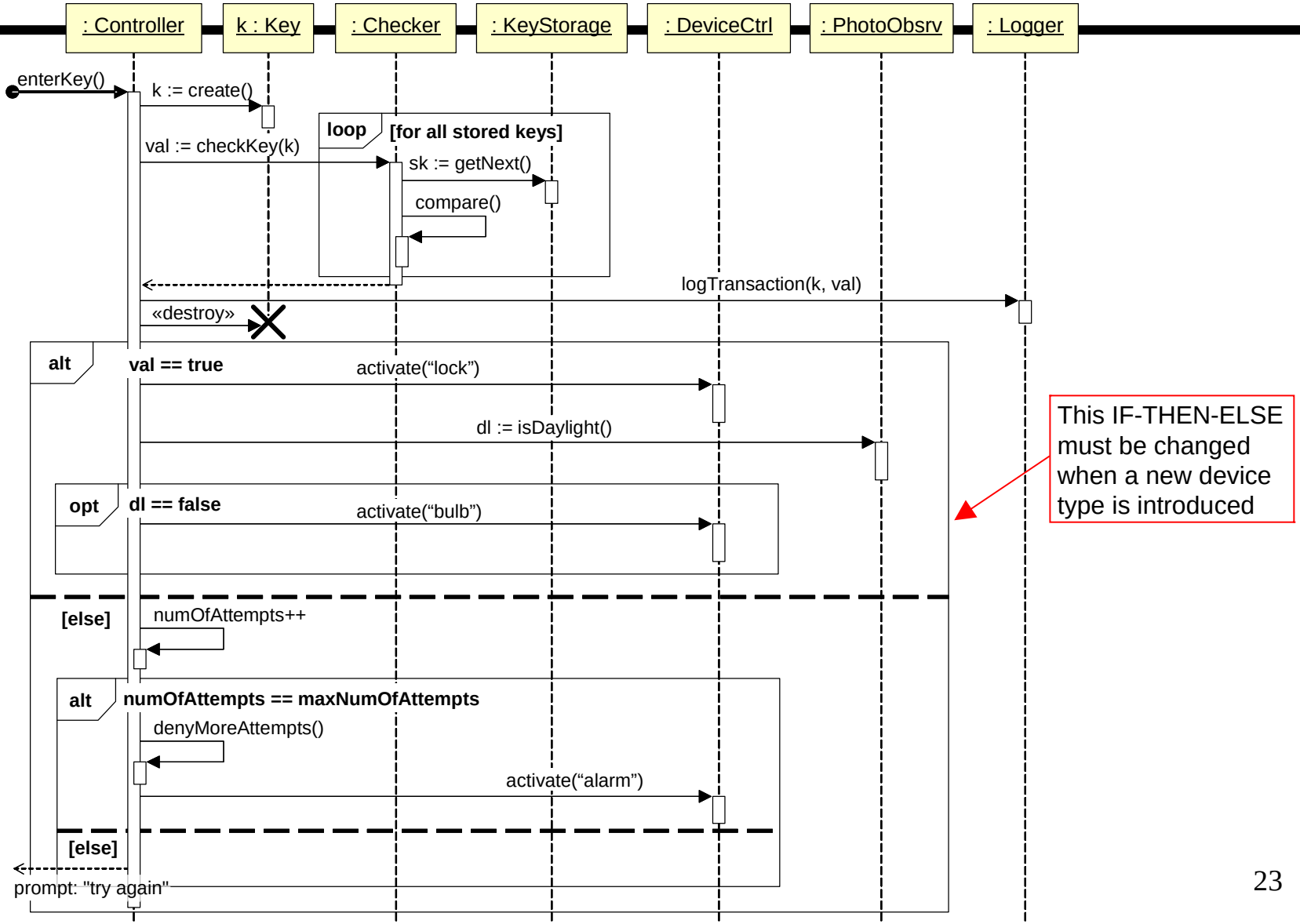
(a)



(b)

From Chapter 2

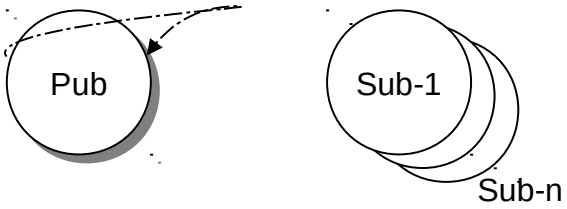
Unlock Use Case



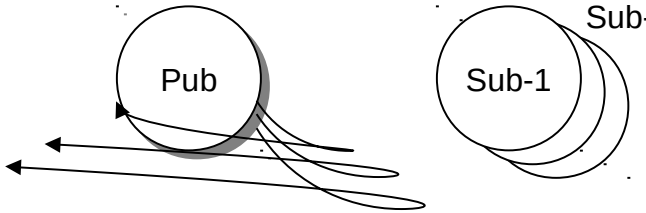
Refactoring to Publisher-Subscriber

Conditional logic is decided here, at design time, instead of run time

1. Subscribe for appropriate events



Event type	Subscriber
keyIsValid	LockCtrl, LightCtrl
keyIsInvalid	AlarmCtrl
itIsDarkInside	LightCtrl
. . .	



No need to consider the "appropriateness" of calling the "servers"

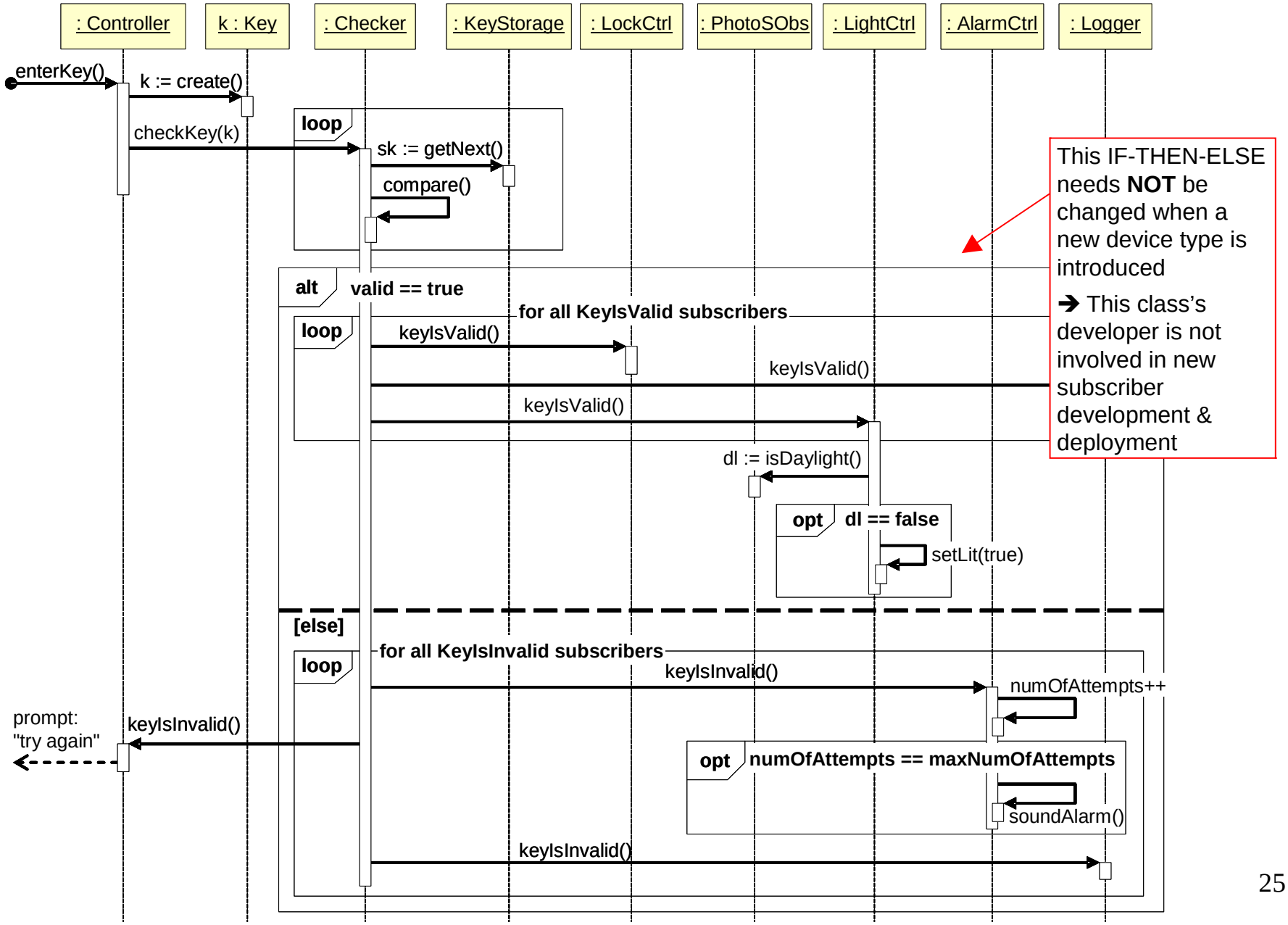
2. When event occurs:
   (a) Detect occurrence: keyIsValid / keyIsInvalid
   (b) Notify only the subscribers for the detected event class

Design-time decisions are better than runtime decisions, because they can be easier checked if they "work" (before the product is developed & deployed)

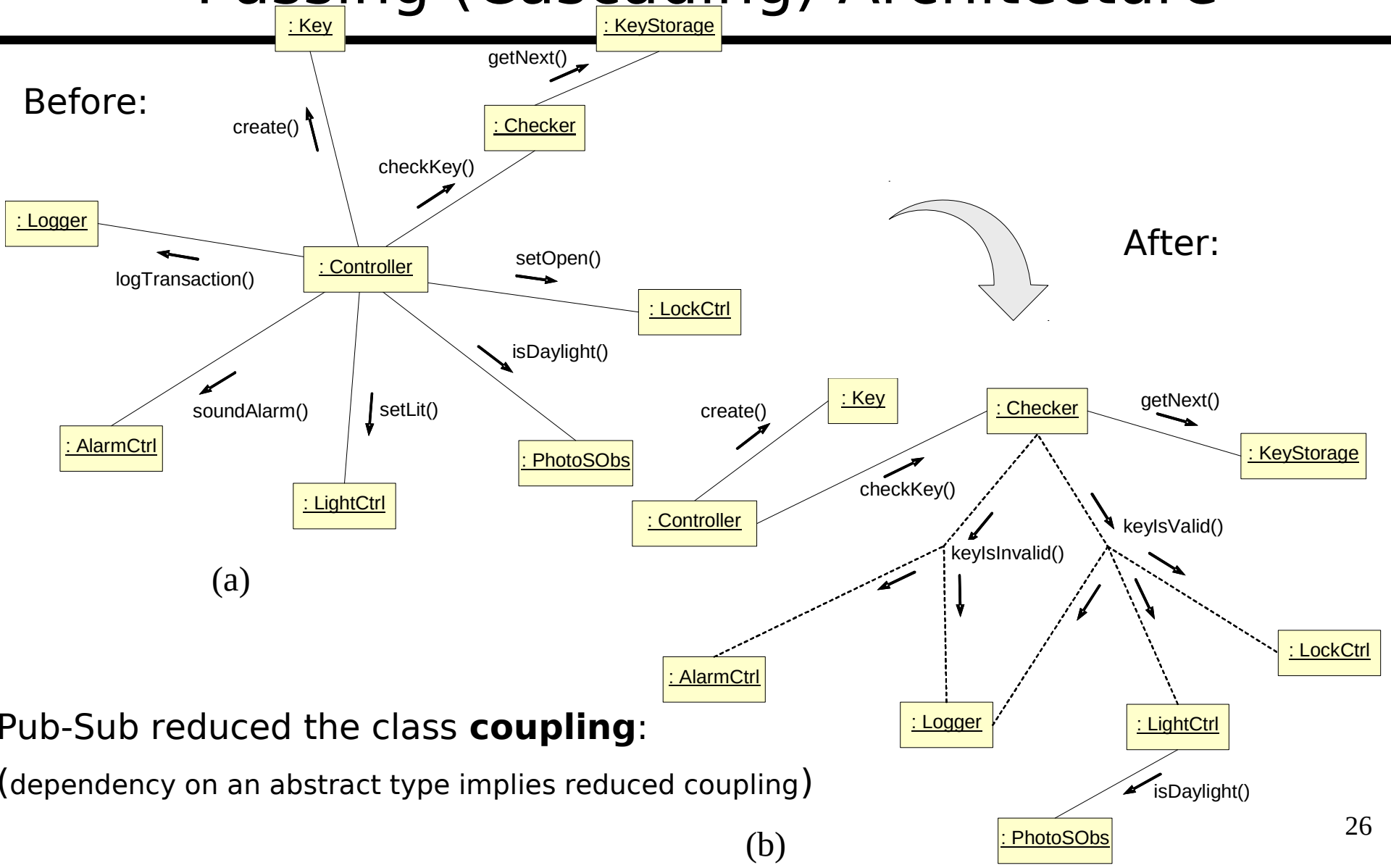
If a new device is added -- just write a new class; NO modification of the Publisher!

Complexity due to decision making cannot disappear;
It is moved to the design stage, which is early and easier to handle.
The designer's choice is then hard-coded, instead of checking runtime conditions.

Pub-Sub: Unlock Use Case



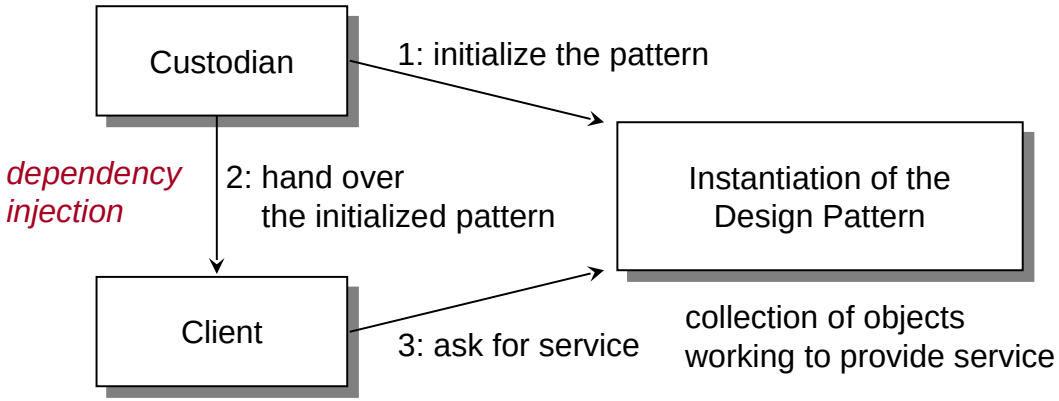
From Hub-and-Spokes (Star) to Token Passing (Cascading) Architecture



Pub-Sub reduced the class **coupling**:

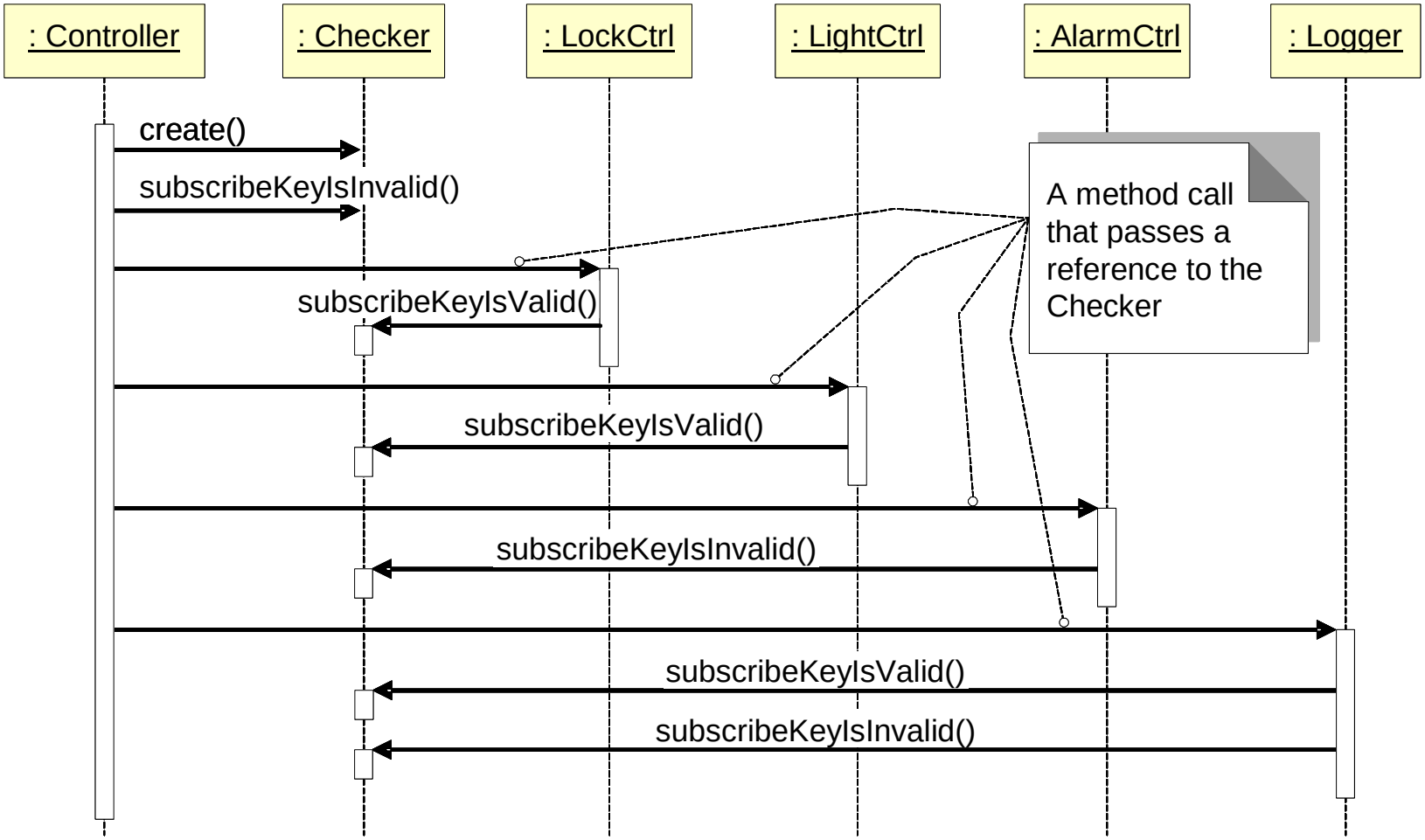
(dependency on an abstract type implies reduced coupling)

Design Patterns: Dependency Injection



Alternative names for Custodian are Assembler or Initializer

Pub-Sub: Initialization



Practical Issues

1. Do not design for patterns first
   - Reaching *any* kind of solution is the priority; solution optimization should be secondary
2. Refactor the initial solution to patterns
   - E.g., to reduce the complexity of the program's conditional logic

- Important to achieve a tradeoff between rapidly progressing towards the system completion versus perfecting the existing work
- Uncritical use of patterns may yield worse solutions!