Software Engineering

Object Oriented Design - Basics

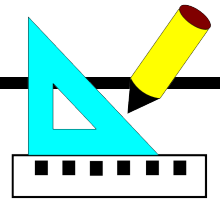
何明昕 HE Mingxin, Max

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

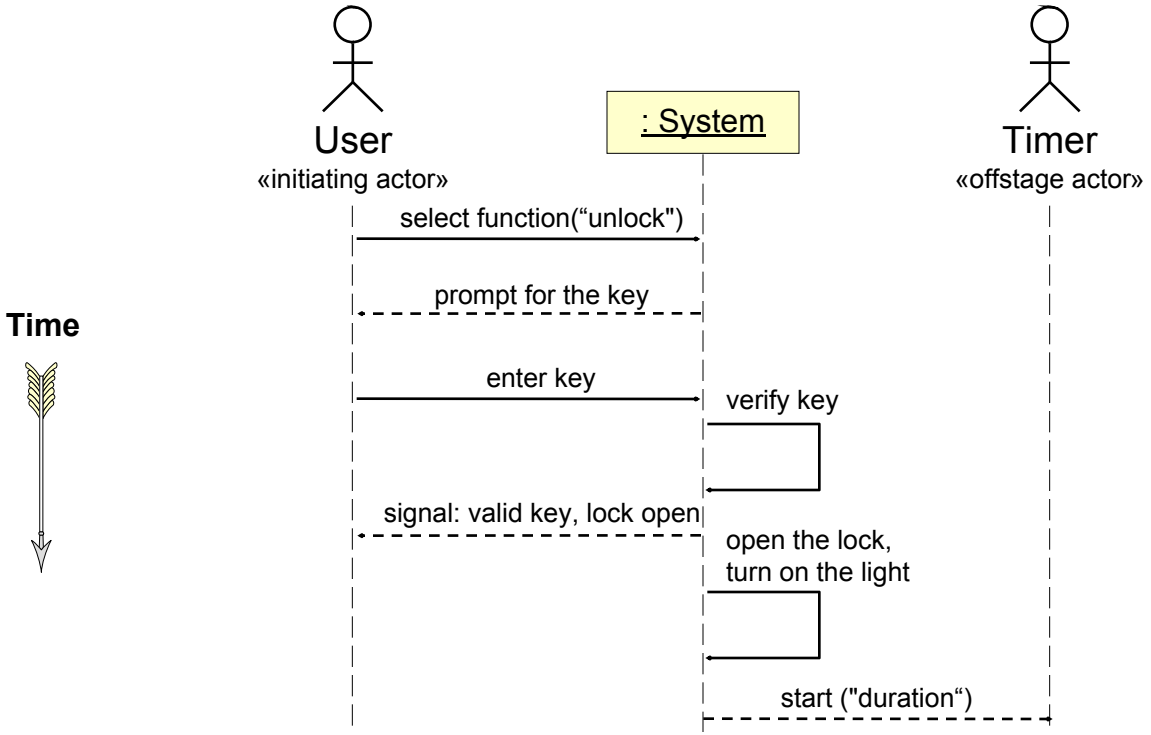
Topics



- Gluing the Modules from Analysis Stage
  - Assigning responsibilities for actions of use-case plans to modules
- Design Principles
  - Expert Doer
  - High Cohesion
  - Low Coupling
- Business Policies
- Class Diagram

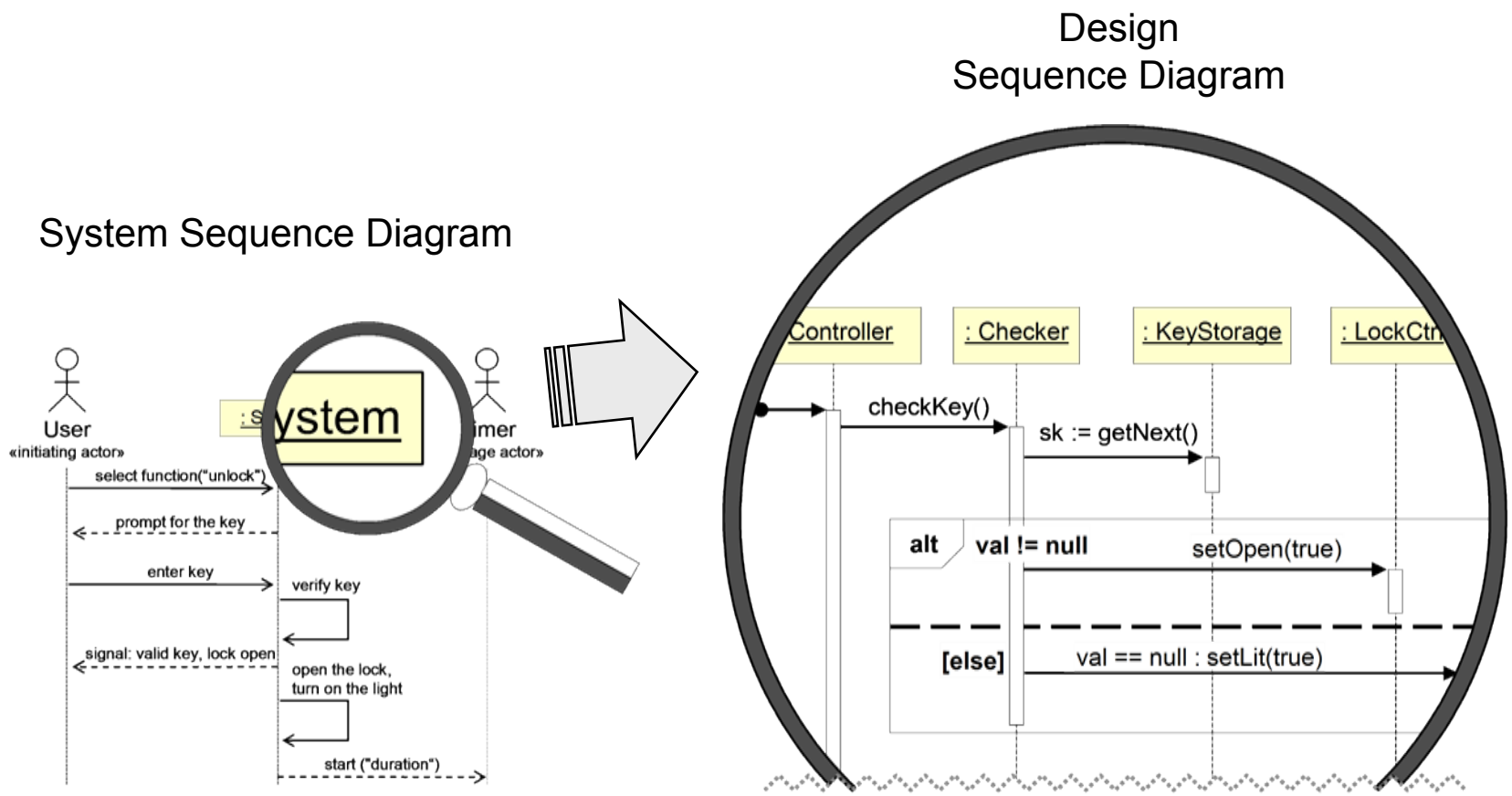
System Sequence Diagrams

We already worked with interaction diagrams: System Sequence Diagrams



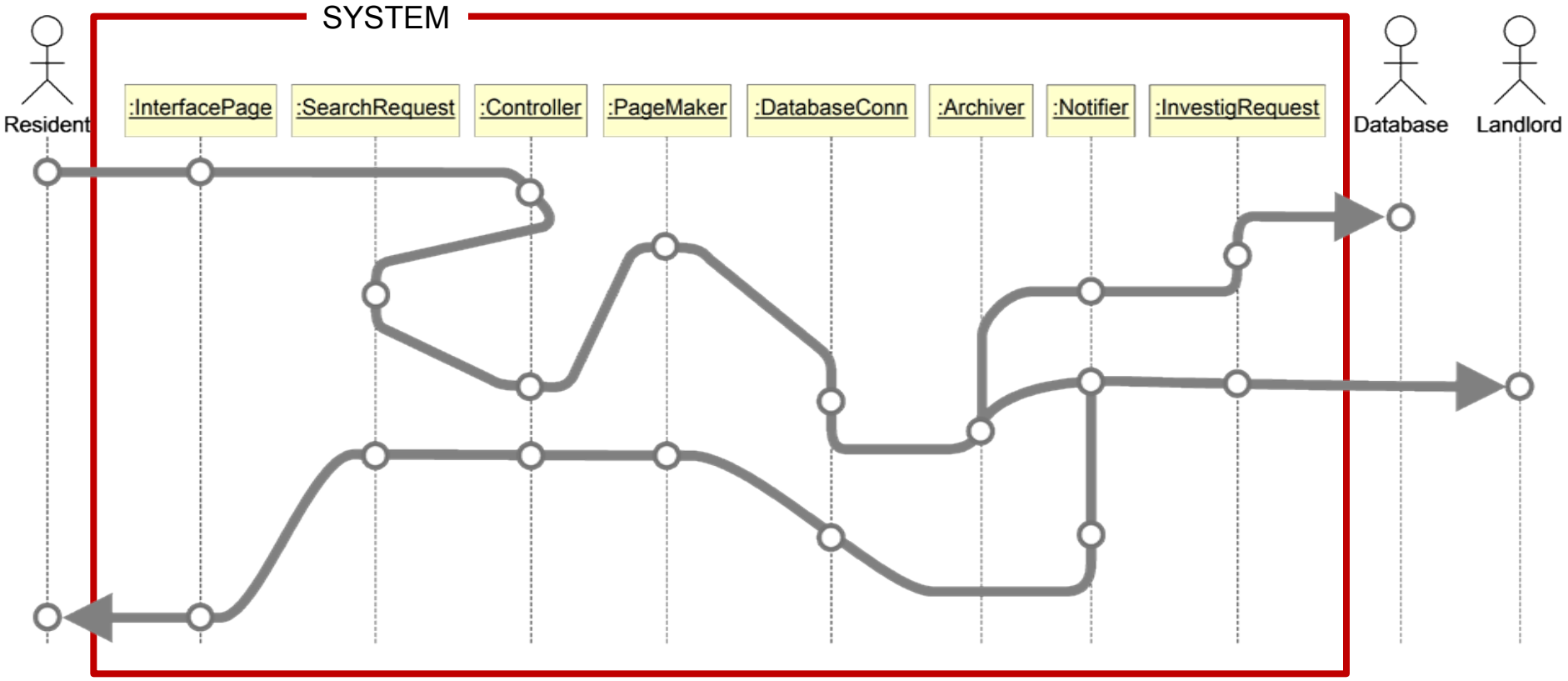
System Sequence Diagrams represent interactions between the **actors**
(the system is also an “actor”)

Design: Object Interactions



- System Sequence Diagrams represent interactions of **external actors**
- Module Sequence Diagrams represent interactions of **modules inside the system**

Metaphor for Software Design: "Connecting the Dots" within the System Box



We start System Sequence Diagrams (which show only actors and the system as a "black box") to design the internal behavior using concept modules from the Domain Model and modify or introduce new modules, as needed to make the given system function work.

Types of Object Responsibilities

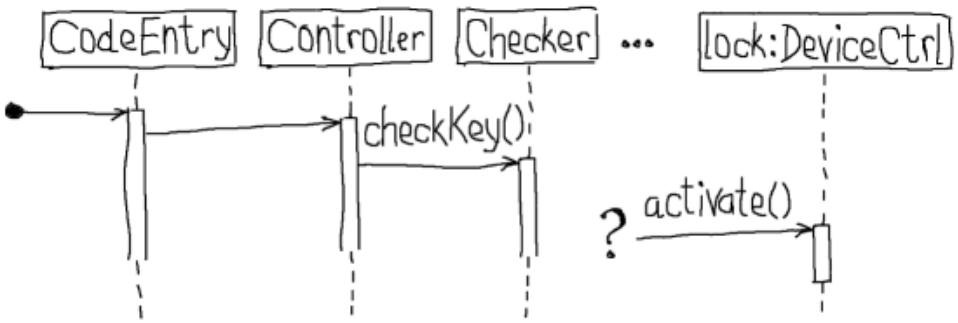
- **Knowing responsibility**: Memorizing data values, collections, or references to other objects, represented as attributes
- **Doing responsibility**: Performing computations, data processing, control of physical devices, etc., represented as methods
- **Delegation responsibility**: Delegating subtasks to object's **dependencies**, represented as sending messages (method invocation)

Example ...

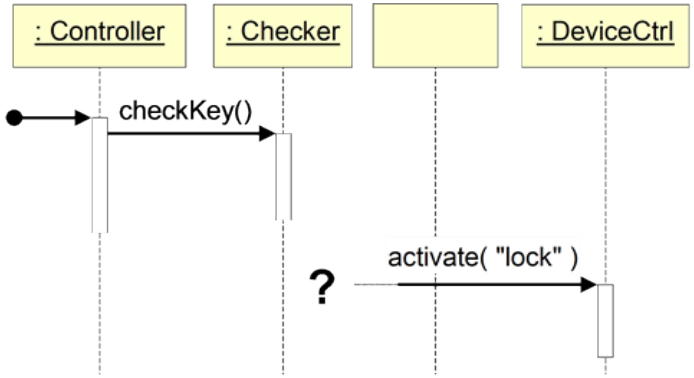
Delegation responsibilities identified for the system function “enter key”:

Responsibility Description
Send message to Key Checker to validate the key entered by the user.
Send message to a DeviceCtrl to disarm the lock device.
Send message to a DeviceCtrl to switch the light bulb on.
Send message to PhotoObserver to report whether daylight is sensed.
Send message to a DeviceCtrl to sound the alarm bell.

Assigning Responsibilities: Design Diagramming



(a) Hand-drawn during creation



(b) Computer-based UML specification of the design

- Two purposes of design diagrams:
  a) Communication tool, during the creative process ➔ use hand drawings and take an image by phone camera (don't waste time on polishing something until you're feel confident that you reached a good design)
  b) Specification tool for documentation ➔ use UML design tools to produce presentable and professional-looking diagrams

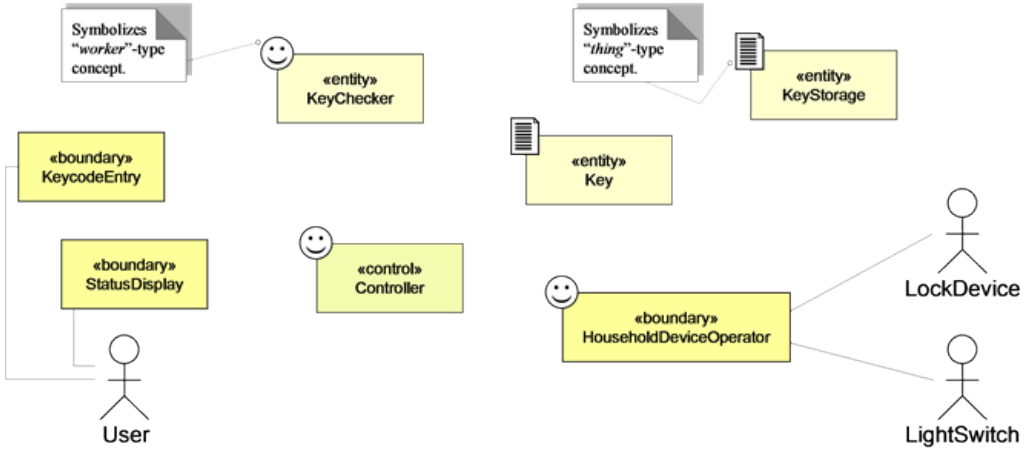
Gluing the Modules
("Connecting the Dots")

Starting Points:

Use Case UC-1: **Unlock** (plan of action):

1. User enters the key code
2. System verifies that the key is valid
3. System signals the key validity
4. System controls:
   (a) LockDevice to disarm the lock
   (b) LightSwitch to turn the light on

Domain Model from UC-1 (concept modules):



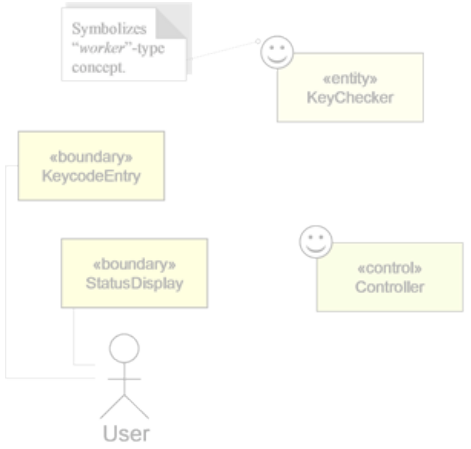
Gluing the Modules by Plan Walkthrough

Starting Points:

Use Case UC-1: **Unlock** (plan of action):

1. User enters the key code
2. System verifies that the key is valid
3. System signals the key validity
4. System controls:
   (a) LockDevice to disarm the lock
   (b) LightSwitch to turn the light on

Domain Model from UC-1 (concept modules):



Scenario Walkthrough:

for mapping a Use Case plan-of-action to the Domain Model

Q: who handles this data?

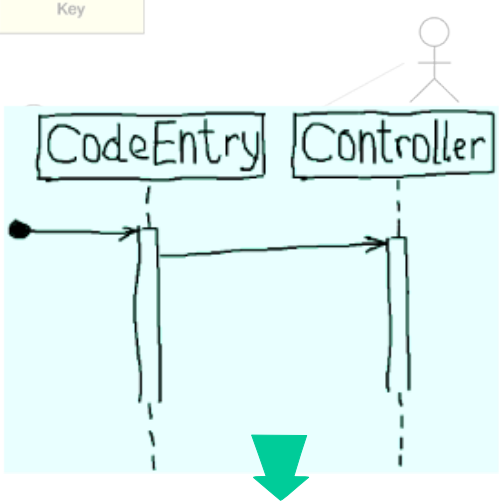
→ Interface objects and Controller

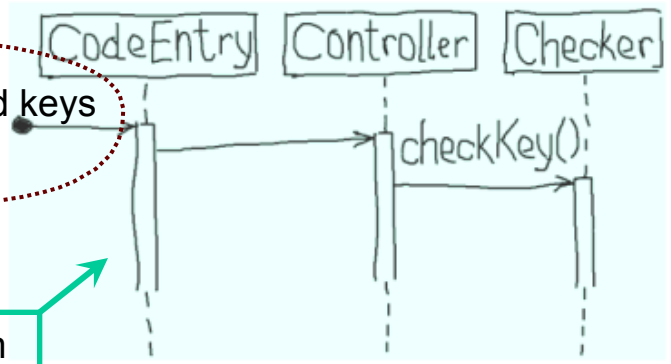
send message: checkKey(k)

Q: who performs the verification? Based on what data?

→ Key Checker, based on entered key-code and stored valid keys

return value





Design Sequence Diagram

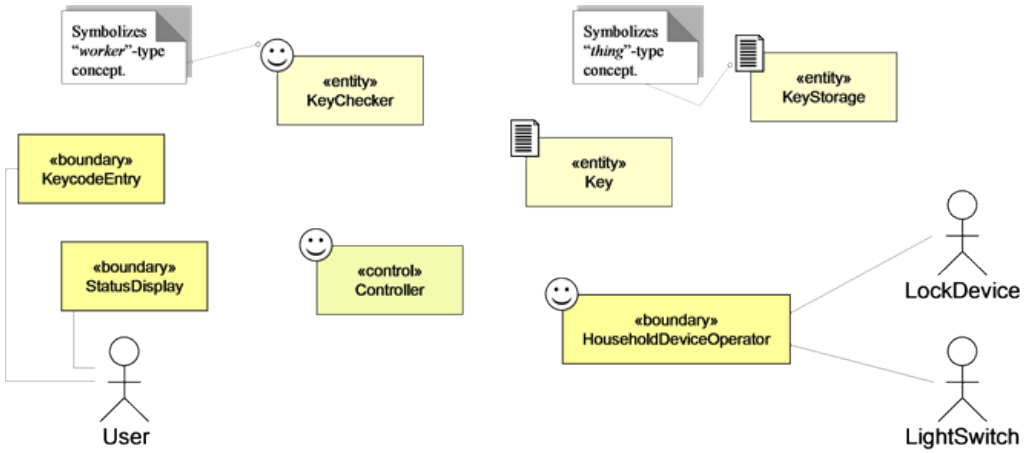
Mapping Actions to Modules

Starting Points:

Use Case UC-1: **Unlock** (plan of action):

1. User enters the key code
2. System verifies that the key is valid
3. System signals the key validity
4. System controls:
   (a) LockDevice to disarm the lock
   (b) LightSwitch to turn the light on

Domain Model from UC-1 (concept modules):



Scenario Walkthrough:

for mapping a Use Case plan-of-action to the Domain Model

Q: who handles this data?

→ Interface objects and Controller

message: checkKey(k)

Q: who performs the verification? Based on what data?

return value

→ Key Checker, based on entered key-code and stored valid keys

message: ???

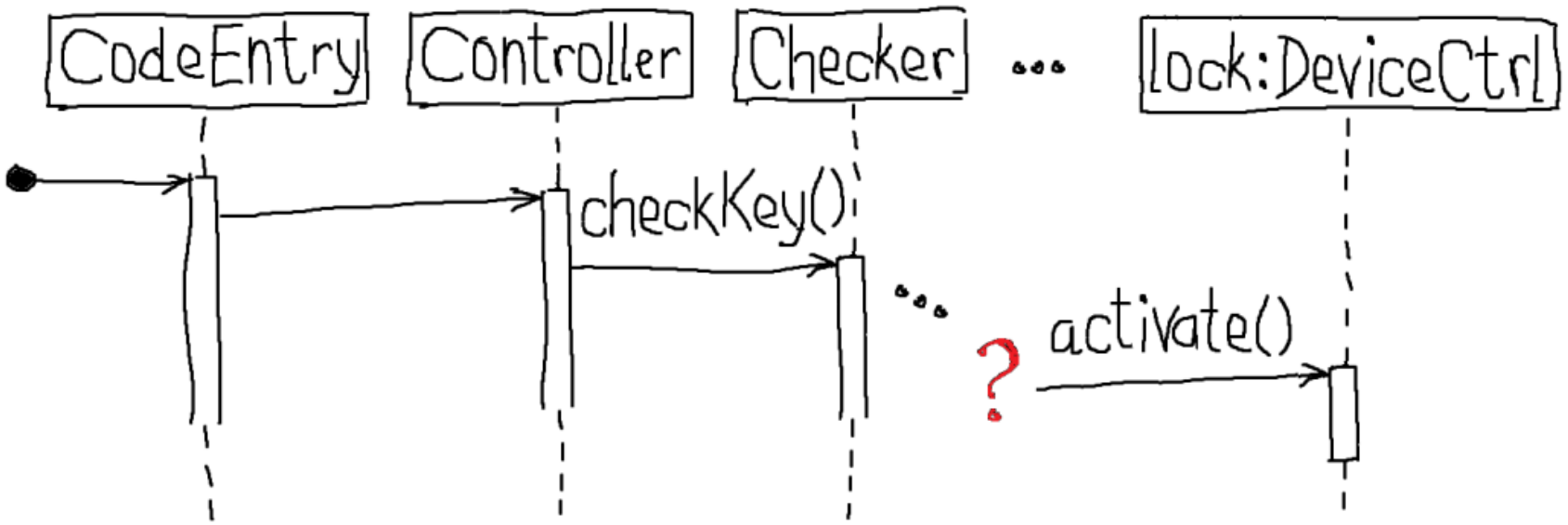
Q: who signals? Based on what data?

→ Controller and Interface objects, based on key verification; because they are «boundary»

Q: who signals? Based on what data?

→ Controller or Key checker **???**, based on key verification

Sequence Diagram (in progress)



Q: who performs the verification? Based on what data?

return value

Key Checker, based on entered key-code and stored valid keys

message: ???

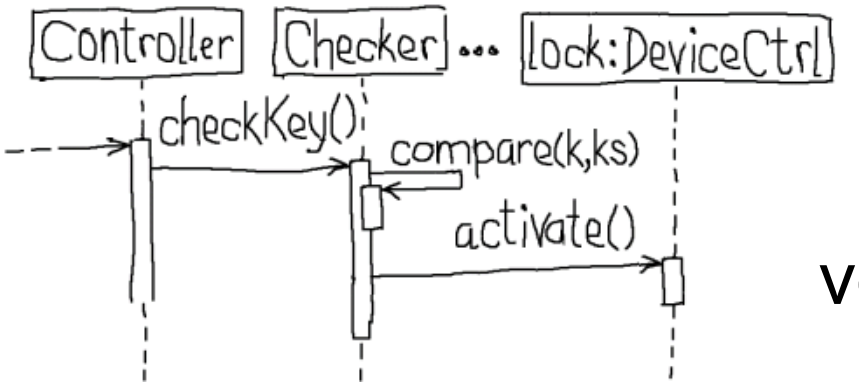
Q: who signals? Based on what data?

Controller and Interface objects, based on key verification; because they are «boundary»

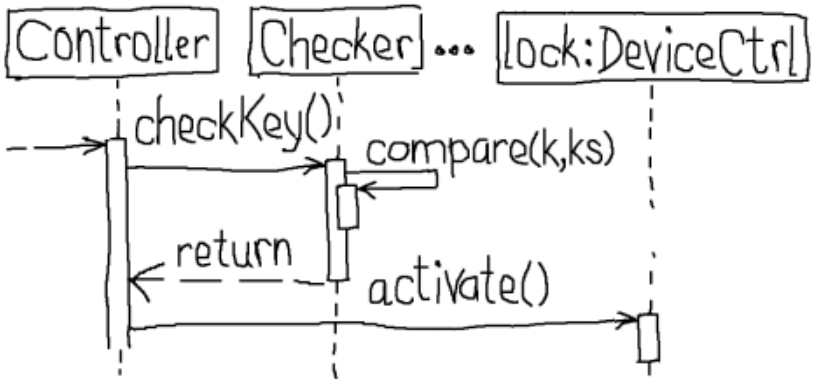
Q: who signals? Based on what data?

Controller or Key checker **???**, based on key verification

Alternative Designs:



versus

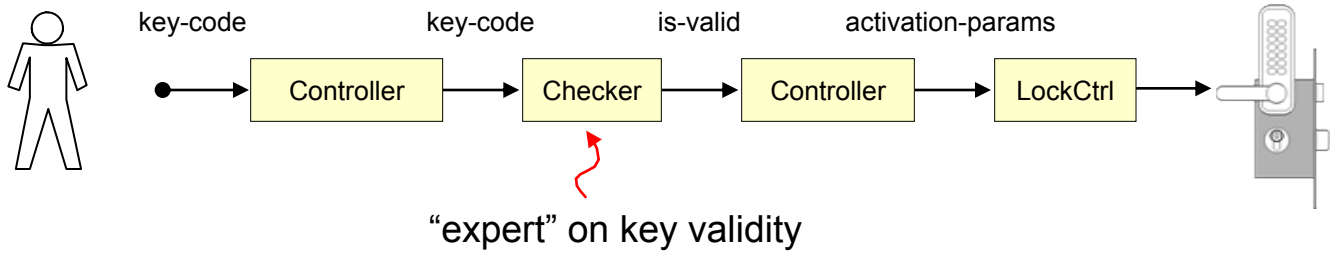


- Which design is better?
- How to evaluate the "goodness" of a design?

How Data Travels

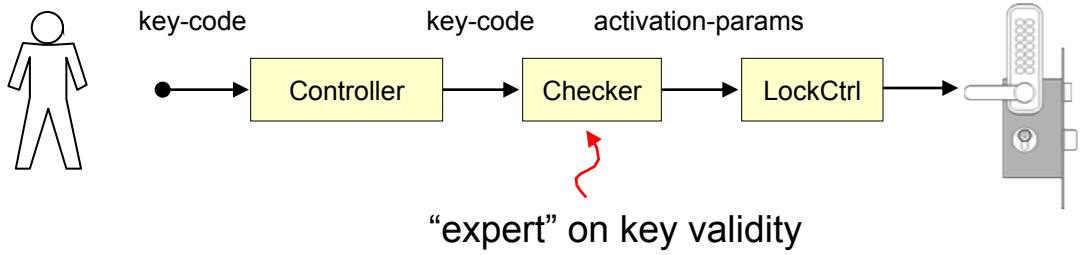
Option A:

"expert" (Key Checker) passes the information (key validity) to another object (Controller) which uses it to perform some work (activate the lock device)



Option B:

"expert" (Key Checker) directly uses the information (key validity) to perform some work (activate the lock device)

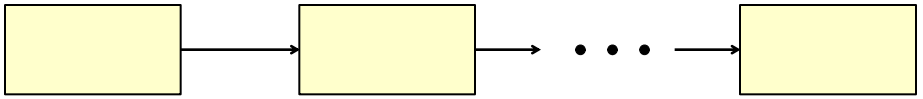


Advantage:
Shorter communication chain

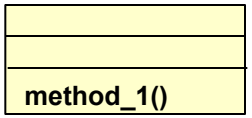
Drawback:
Extra responsibility for Checker

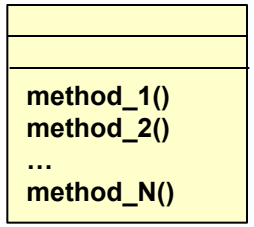
Characteristics of Good Designs

- Short communication chains between the objects

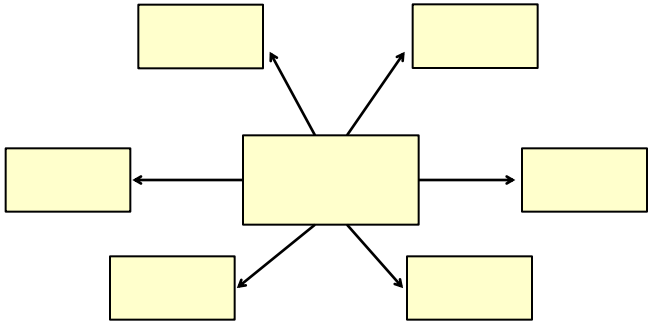


- Balanced workload across the objects
  - divide-and-conquer during analysis should divide the system into manageable modules





- Low degree of connectivity (associations) among the objects
  - A system with “hub” modules is more prone to failures



Some Design Principles

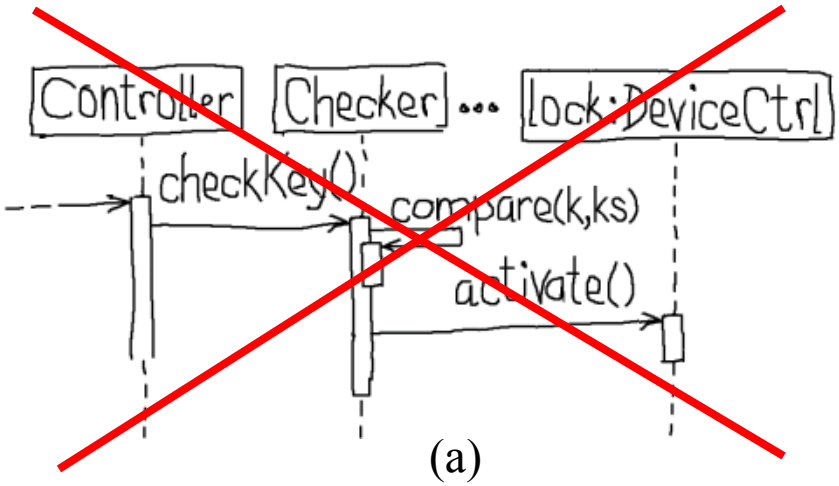
- **Expert Doer Principle**: module that knows should communicate information to those that need it
  - How to recognize a violation: If a method call passes many arguments
- **High Cohesion Principle**: module should not take on too many computation responsibilities
  - How to recognize a violation: If a class has many loosely or not-related attributes and methods
  - Single Responsibility Principle (next lecture)
- **Low Coupling Principle**: module should not delegate responsibilities in many tiny parts
  - How to recognize a violation: many outgoing links
  - Better solution: Hierarchical delegation by limiting the number of dependencies for each delegate and letting the delegates further split the complex resopnsibilities

There are many more … (next slides)

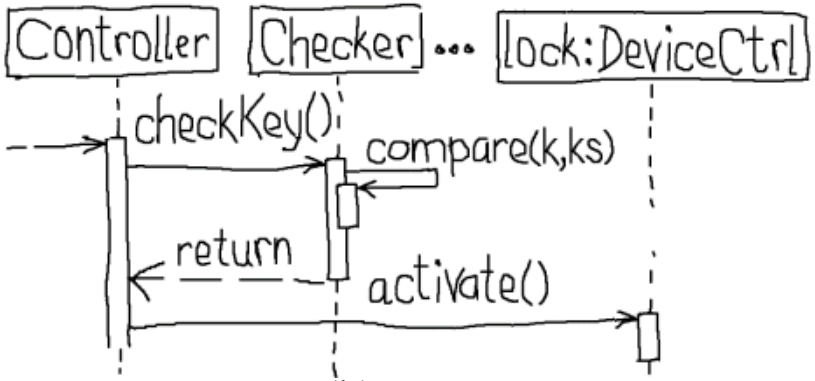
Design Involves Compromises

- Any nontrivial design is a **compromise** between the desired and the possible
- Design principles may contradict each other:
  - Shortening the communication chain (*expert doer*) tends to concentrate responsibilities to fewer objects (*low cohesion*)
  - Minimizing the number of responsibilities per object (*high cohesion*) tends to increase the number of dependencies (*strong coupling*)
- It is critical for others to know how the designer resolved each compromise/tradeoff, so others can evaluate the reasoning
  - **document** the reasons for choosing the particular tradeoffs and compromises
  - code alone cannot provide this information
    - code shows the product but not the process of reasoning

Design: Assigning Responsibilities



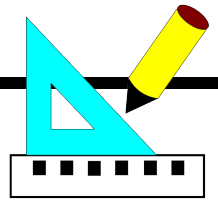
(a)



(b)

- Although the Checker is the first to acquire the information about the key validity, we decide to assign the responsibility to activate the DeviceCtrl to the Controller
- This is because the Controller would need to know key-validity information anyway—to inform the user about the outcome of the key validity checking
- In this way, we maintain the Checker focused on its specialty (key validation) and avoid assigning unrelated responsibilities to it

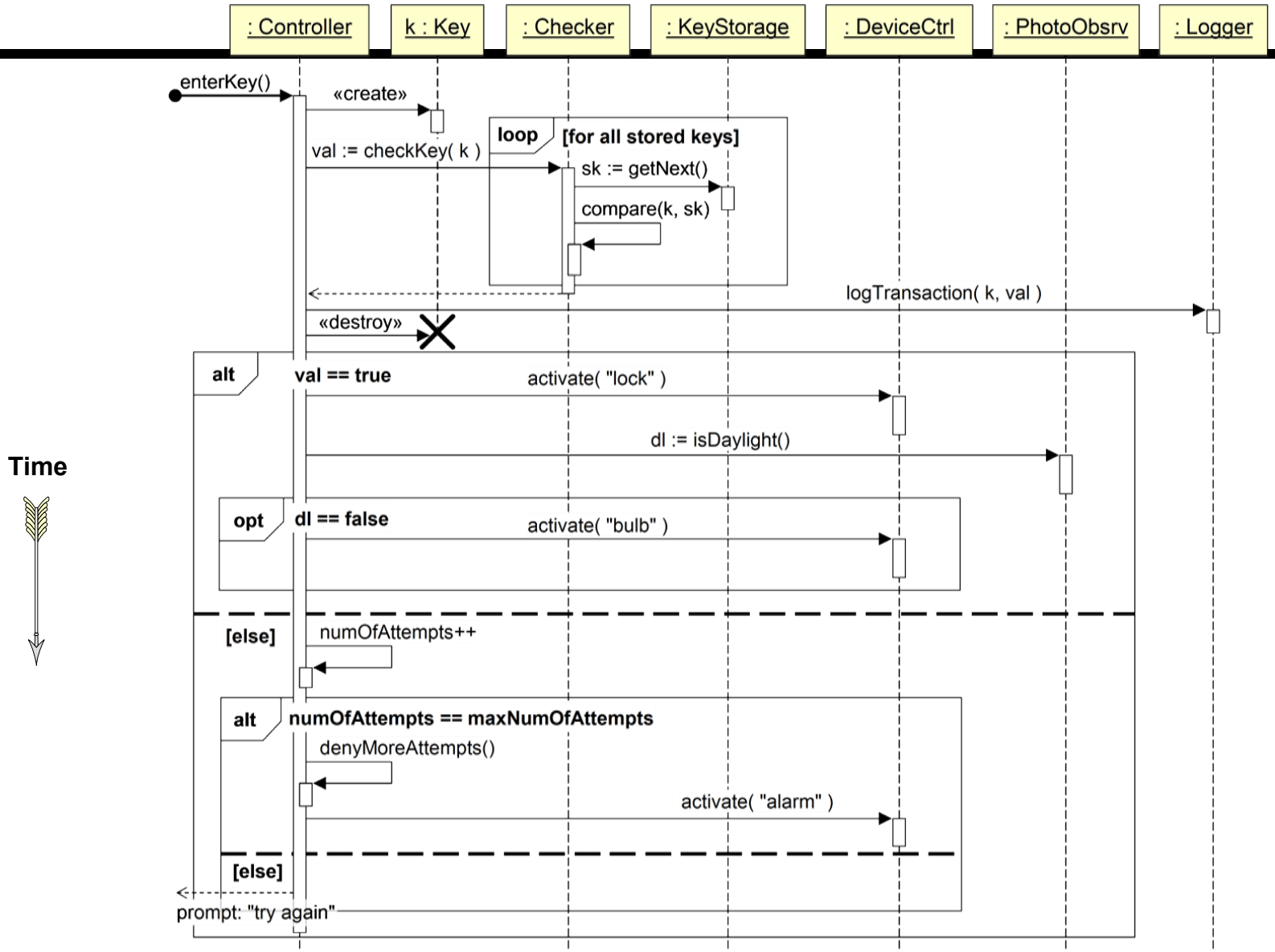
Responsibility-Driven Design



1. Identify the responsibilities
   - start with a *use case* plan-of-action and the *domain model*
   - some may be missed at first and identified during implementation of the design
2. For each responsibility, identify *candidate modules* or *objects* to assign to
   – if the choice appears to be unique then move to the next responsibility
3. Consider the merits and tradeoffs of each alternative by applying the *design principles*
   – select what you consider the "optimal" choice
4. Document the *reasoning process* by which you arrived to each responsibility assignment
   – design involves tradeoffs—let others know how you resolved them
   – preserve not only the final chosen design but also all the alternatives you considered and their perceived merits (*process*, instead of *product* only!)

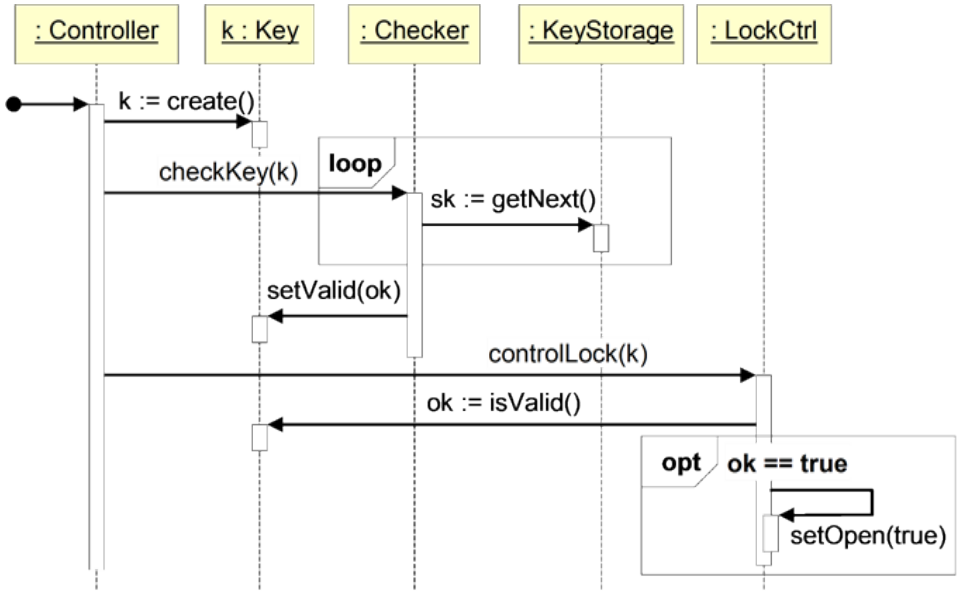
time to tidy up …

Unlock Use Case



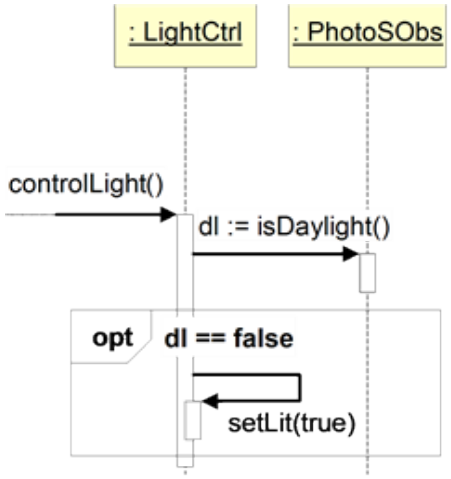
Unlock Seq. Diag. Variation 1

To avoid an impression that the above design is the only one possible...



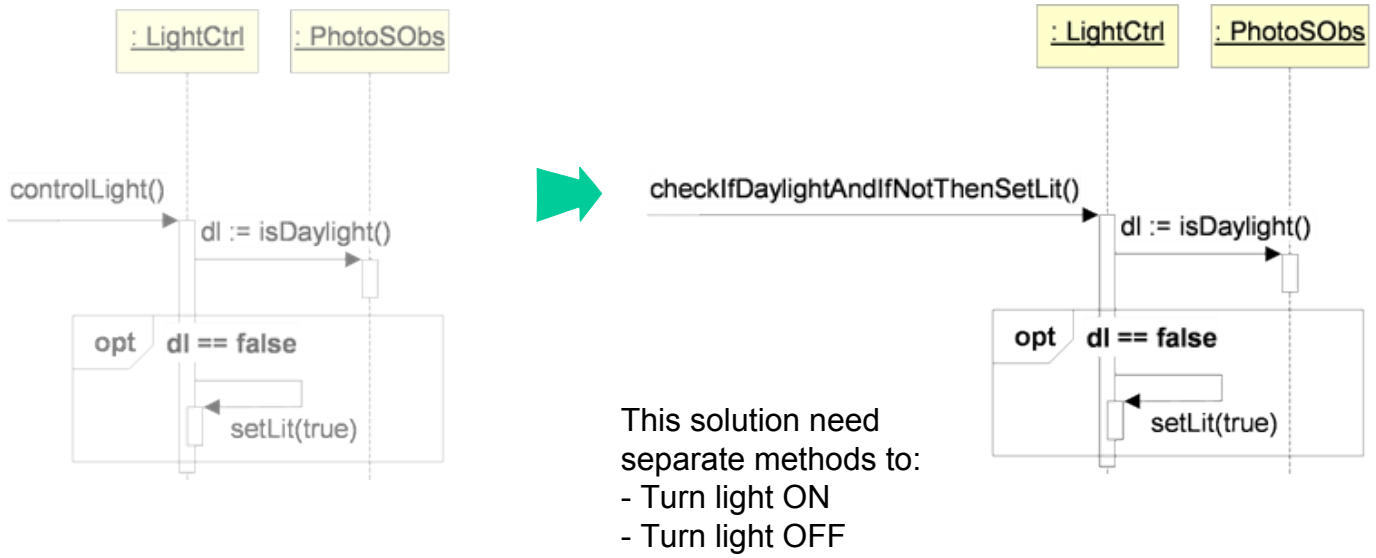
- Sets a Boolean attribute of the Key object: ok = true/false;
- Business logic (IF-THEN rule) relocated from Controller to LockCtrl
- May be useful if Controller and LockCtrl are in different memory spaces and their communication could be intercepted,
  so the Key should be encrypted

Unlock Seq. Diag. Variation 2a



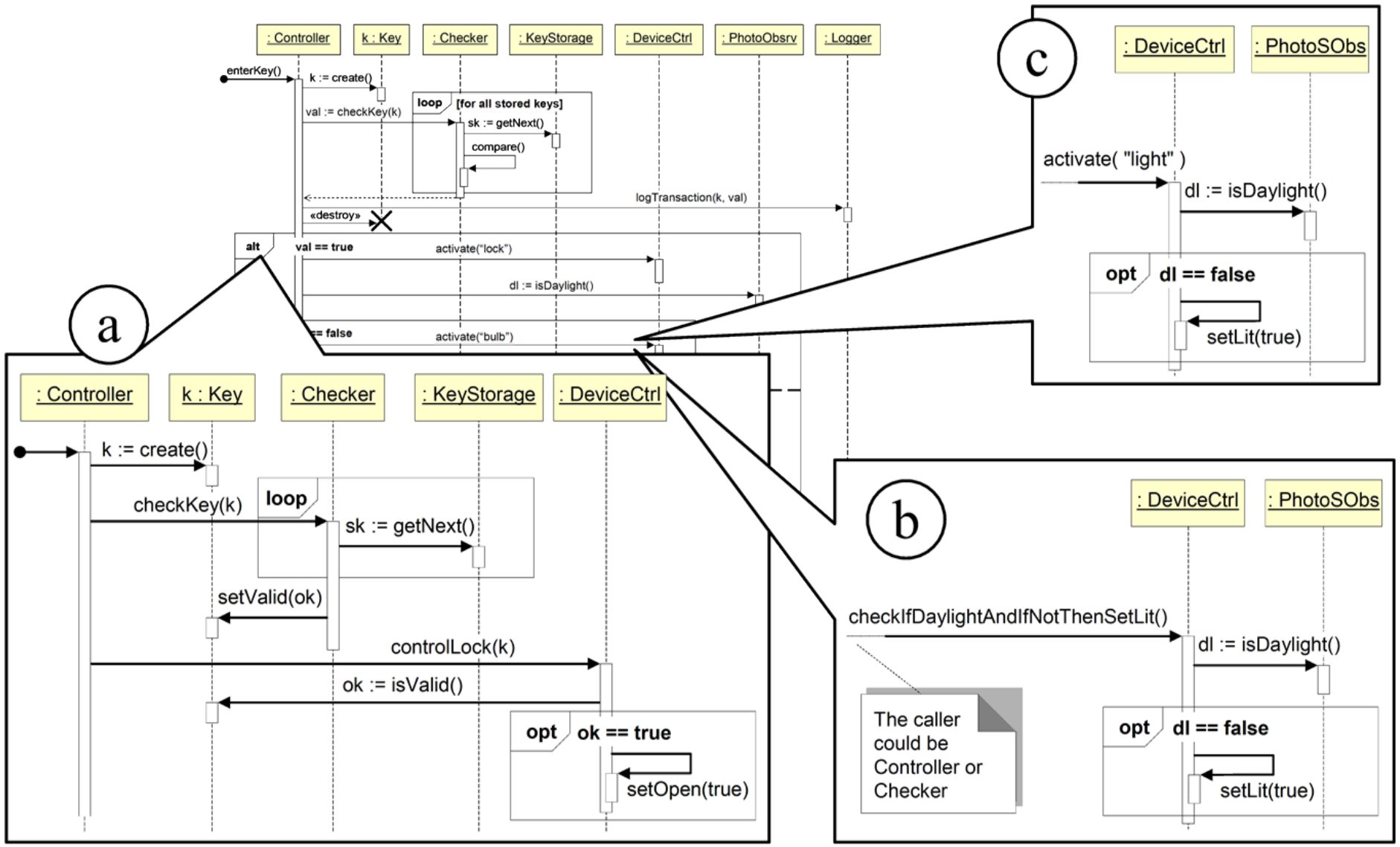
- Instead of the original solution where the Controller, gets involved in the operation of different devices, the Controller only sends the message to one of a group of objects associated with a device
- Controller contains high-level business logic/policies and should not get involved in device-control details
- Checking whether it is dark and the light is needed should be a responsibility of LightCtrl
- Similarly, if other devices require multi-step control, the Controller should not be involved
  - For example, MusicCtrl would find the appropriate playlist and activate the player
  - AlarmCtrl will determine the list of who needs to be notified and send text messages

Unlock Seq. Diag. Variation 2b



- It may seem helpful that checkIfDaylightAndIfNotThenSetLit() is named informatively (reveals the intention) …
- but the low-level knowledge of operating a particular device (lighting) is encoded in the name of the method …
- which, in turn, means that low-level knowledge ("mechanism") is imparted onto the caller (Controller) which should be concerned with high-level business policies
- Mixing low-level knowledge with high-level knowledge results in *rigid* and *non-reusable* designs

Summary of Some Design Variations



Are We Done w/ UC-1: Unlock ?

- Didn't check that the user is at the right door
  - Missing: Managing access rights
- Didn't distinguish critical and non-critical functions
  - For example, what if logTransaction() call to Logger does not return, e.g., no access to database (network outage) or disk-space full ?
  - Missing: Independent execution of non-critical functions
- Adding new household devices causes major design changes
- Controller has several *unrelated* reasons for future changes:
  1. Business policies are entangled with authentication mechanisms
  2. Device management
- Etc.

➔ this design will be revisited in future lectures!

Business Policies

policy:

IF key $\in$ ValidKeys THEN disarm lock and turn lights on

ELSE

mechanism:

increment failed-attempts-counter

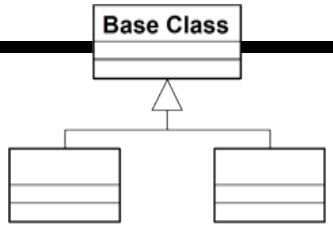
IF failed-attempts-counter equals maximum number allowed

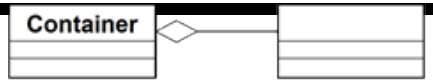
THEN block further attempts and raise alarm

Should be moved into a separate object:

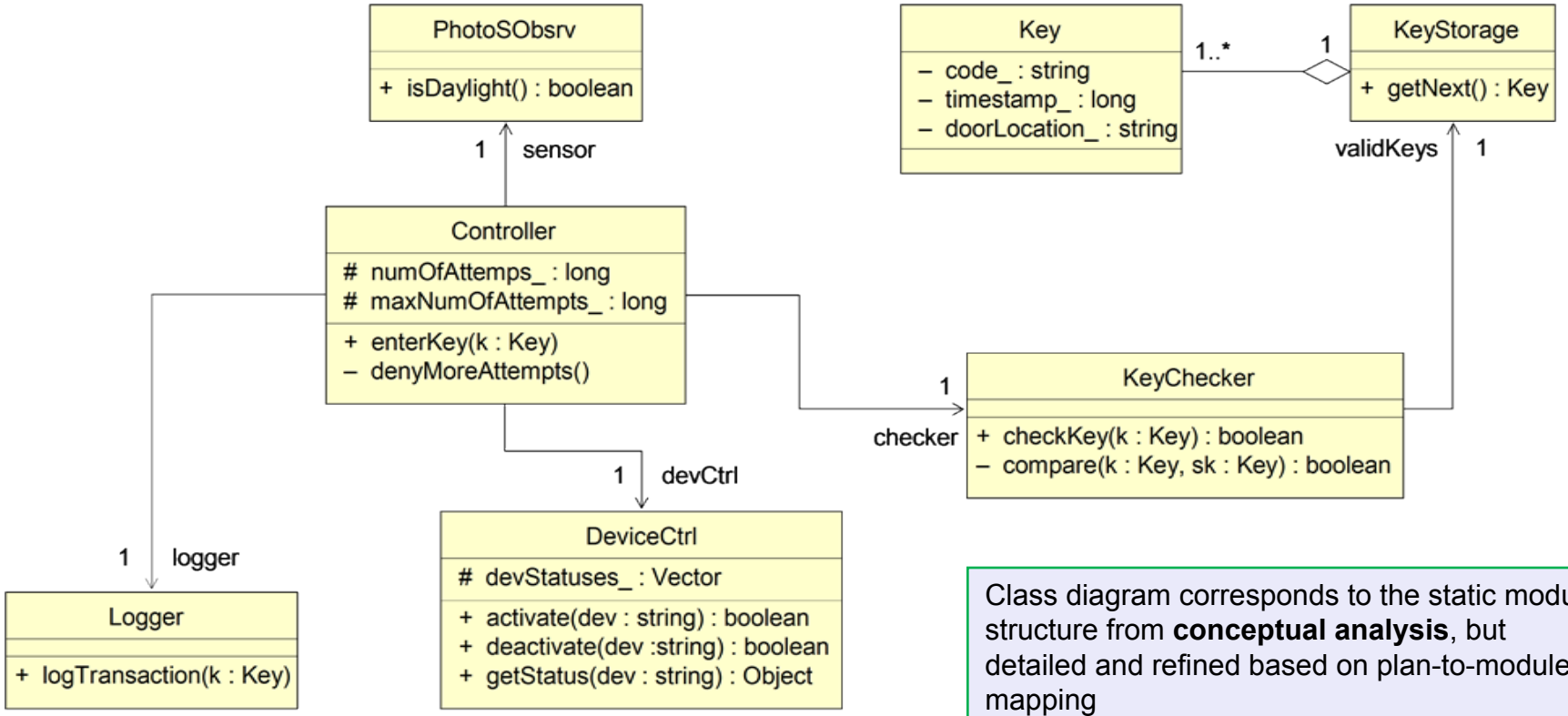
- Make them explicit part of the model
- Separate business policies from impl. Mechanism
- See **Dependency Inversion Principle** (next lecture)

Class Diagram





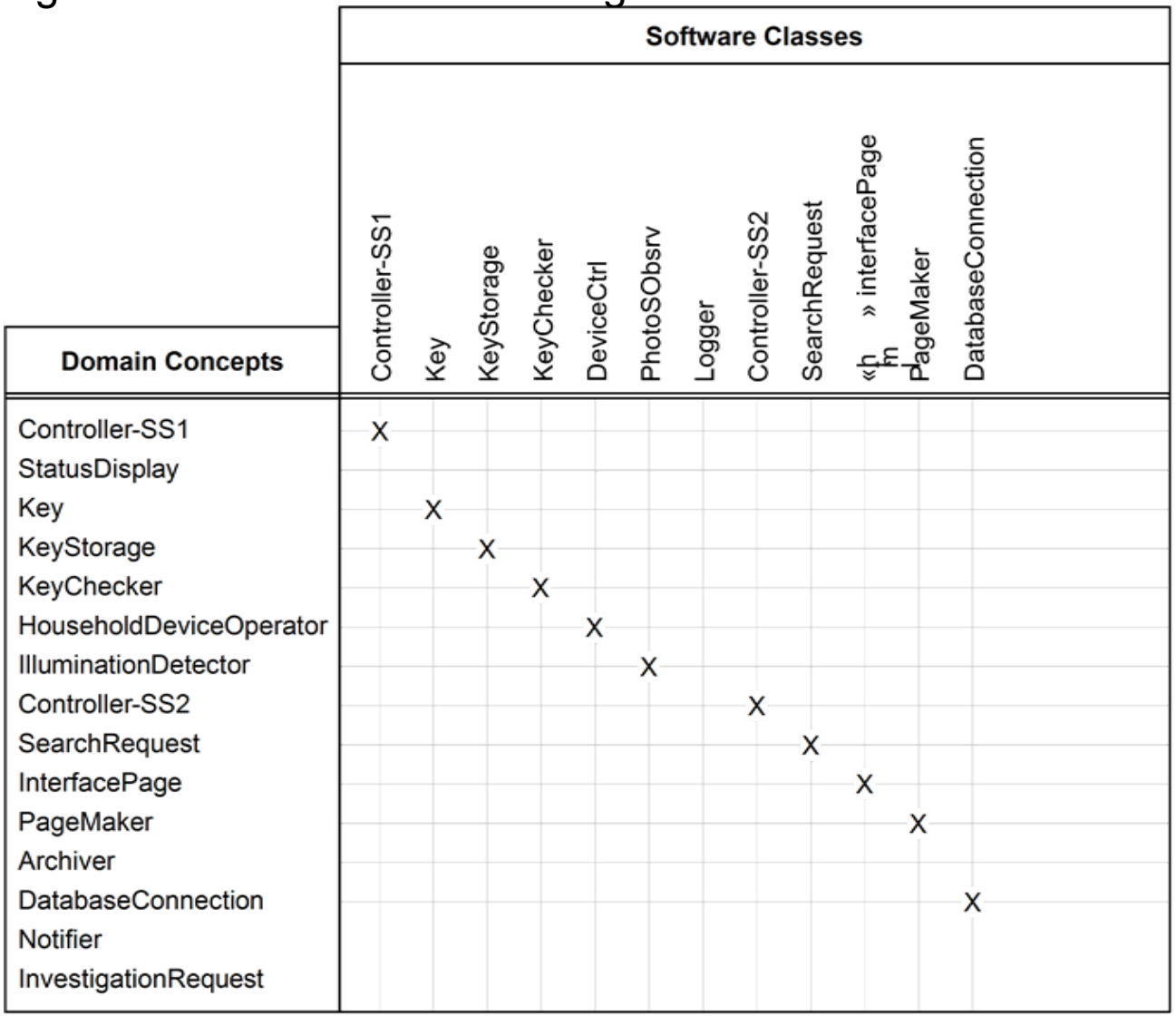
Class diagram should be derived by looking-up the sequence diagrams



Class diagram corresponds to the static module structure from **conceptual analysis**, but detailed and refined based on plan-to-module mapping

Traceability Matrix (3)

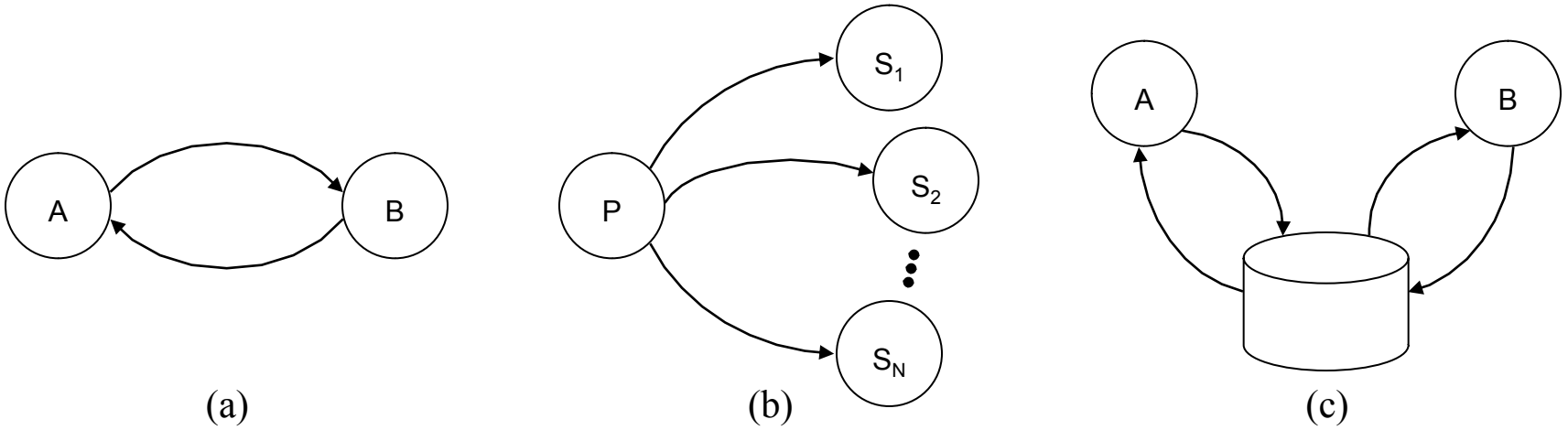
Mapping: Domain model to Class diagram



Domain Concepts	Controller-SS1	Key	KeyStorage	KeyChecker	DeviceCtrl	PhotoSObsrv	Logger	Controller-SS2	SearchRequest	« » interfacePage	PageMaker	DatabaseConnection
Controller-SS1	X											
StatusDisplay												
Key		X										
KeyStorage			X									
KeyChecker				X								
HouseholdDeviceOperator					X							
IlluminationDetector						X						
Controller-SS2								X				
SearchRequest									X			
InterfacePage										X		
PageMaker											X	
Archiver												
DatabaseConnection												X
Notifier												
InvestigationRequest												

For missing checks, explain whether the concept module was discarded or delayed until future iterations

Types of Object Communication



(a) (b) (c)