

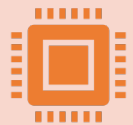
# System Modeling

Santanu Dash

# System modeling



The process of developing abstract models of a system, with each model presenting a different view or perspective of that system.



Representing a system based on graphical notations from the Unified Modeling Language (UML).



System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.



## **Models of the existing system are used to**

Help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses.

Lead requirements for the new system.



## **Models of the new system are used during requirements engineering to**

Help explain the proposed requirements to other system stakeholders.

Engineers use these models to discuss design proposals and to document the system for implementation.

# System perspectives

<b>External</b>	Model the context or environment of the system.
<b>Interaction</b>	Model the interactions between a system and its environment, or between the components of a system
<b>Structural</b>	Model the organization of a system or the structure of the data that is processed by the system
<b>Behavioral</b>	Model the dynamic behavior of the system and how it responds to events

# Context models



Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.

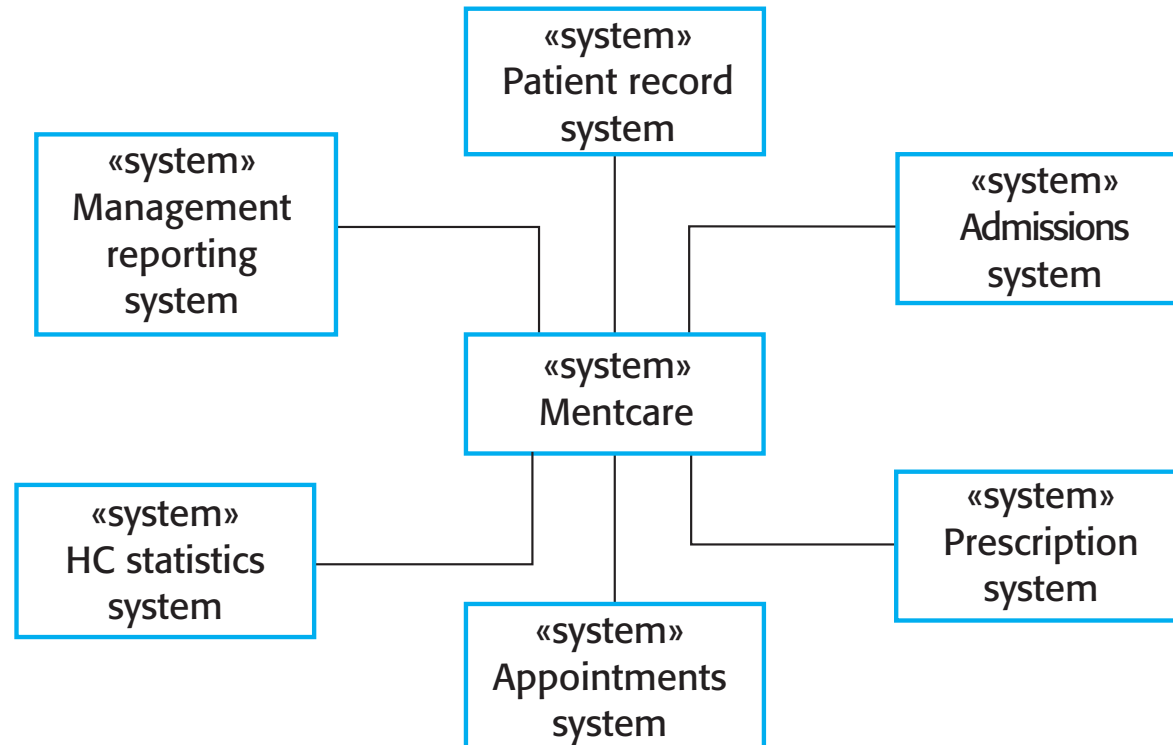


Social and organisational concerns may affect the decision on where to position system boundaries.



Architectural models show the system and its relationship with other systems.

# The context of the Mentcare system



# Interaction models



# Use case modeling



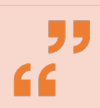
Use cases were developed originally to support requirements elicitation and now incorporated into the UML.



Each use case represents a discrete task that involves external interaction with a system.



Actors in a use case may be people or other systems.

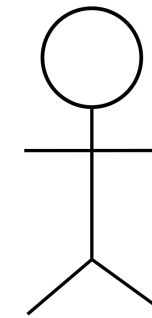


Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.



# Actors

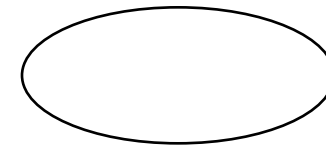
- A role played by an external entity that requires something (functionality) from the system
- To identify, think of the roles played by the users
  - Not limited to human operators/users
  - May include other systems, organisations, time, devices
- Actor is a “stick-man”
  - has no attributes
  - has no operations
  - has only one type of association



*Actor*

# Use cases

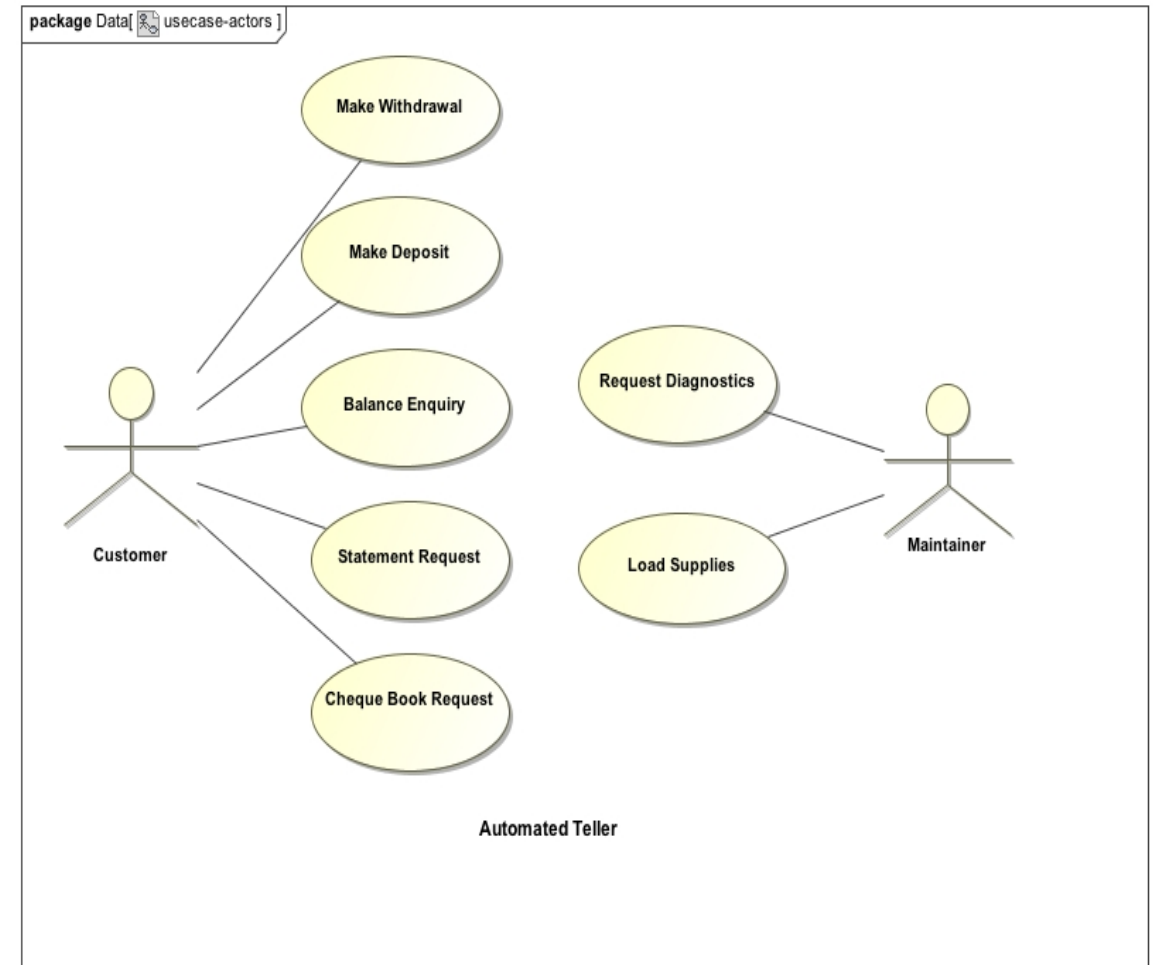
- A use case is a **specific behaviour required of the system**
- A system is described by a set of use cases
- UML permits use cases to have
  - attributes
  - operations
  - statecharts



*Use Case*

# Notation – associations

- Lines connecting actors to use cases are associations in UML
- But...
  - no name
  - no role phrases
  - no cardinality
- So, they simply denote that the actor communicates with the use case



# Documenting use cases

Actors	<b>ActorID</b> -- uniquely identify the actor (by a number) <b>Actor name</b> -- think of its role in the system <b>Description</b> -- short outline of involvement with the system
Use Cases	<b>Use case ID</b> -- provides a way to refer to it from other docs <b>Initiating actor</b> and <b>Use Case Name</b> <b>Purpose</b> -- outline activity / task <b>Pre-condition</b> -- any conditions that must hold before <b>Invariants</b> -- condition that must hold throughout <b>Minimal guarantee</b> -- conditions that must hold after
Scenarios	...

- For scenarios, need to document:
- **Scenario name**
- **Scenario description**
  - Describe the sequence of interactions that need to occur
    - Numbered sequence of succinct declarative statements
  - Include what is required to be done, not how it will be implemented
- **Pre-conditions**
  - Any significant conditions that hold before the scenario can be executed
- **Post-conditions**
  - Any significant conditions that hold after completion of the scenario

<b>Use Case ID</b>	ATM1
<b>Name</b>	Make Withdrawal
<b>Purpose</b>	To allow a bank customer to withdraw a specified amount of cash
<b>Initiating Actor</b>	Customer
<b>Precondition</b>	ATM is 'In Service' AND Cash Supply Balance > Min Withdrawal Amount
<b>Invariants</b>	Customer account shall never fall below agreed limit
<b>Minimal guarantee</b>	Customer activity is logged. Cash amount in the machine, after the transaction, equals the amount in machine beforehand less any dispensed to customer
<b>Requirements</b>	Banking Systems plc; ATM2000 SRS V1.0

## ATM Cash Withdrawal

<b>Scenario</b>	<b>Primary</b>
<b>Name</b>	Successful withdrawal
<b>Description</b>	1 customer inserts bank card and card details are verified 2 customer is prompted to enter PIN to authenticate customer 3 system displays a list of options 4 customer selects the 'Withdraw Cash' option 5 customer selects an amount of cash to withdraw 6 machine delivers the requested cash 7 customer selects 'Quit' from menu 8 customer activity is logged 9 machine ejects card
<b>Postcondition</b>	The customer account balance, daily withdrawal balance and the ATM Cash
<b>Notes</b>	Supply Balance are reduced by withdrawal amount

## Make Withdrawal: a secondary scenario

<b>Use Case ID</b>	ATM1
<b>Name</b>	Make Withdrawal
<b>Purpose</b>	To allow a bank customer to withdraw a specified amount of cash
<b>Initiating Actor</b>	Customer
<b>Precondition</b>	ATM is 'In Service' AND Cash Supply Balance > Min Withdrawal Amount
<b>Invariants</b>	Customer account shall never fall below agreed limit
<b>Minimal guarantee</b>	Customer activity is logged. Cash amount in the machine, after the transaction, equals the amount in machine beforehand less any dispensed to customer
<b>Requirements</b>	Banking Systems plc; ATM2000 SRS V1.0

<b>Scenario</b>	<b>Secondary 1</b>
<b>Name</b>	Invalid PIN, customer then enters PIN successfully
<b>Description</b>	<p>Use case proceeds as for primary scenario up to the point when customer prompted to enter PIN (<a href="#">step 2</a>)</p> <p>3 system informs user that the authentication process failed</p> <p>4 system displays options to 'quit' or 'enter PIN'</p> <p>5 customer selects the 'enter PIN' option</p> <p>6 system authenticates the customer</p> <p>7 use case then proceeds as in primary scenario (<a href="#">step 3</a>)</p>
<b>Postcondition</b>	The customer account balance, daily withdrawal balance and the ATM Cash Supply Balance are reduced by withdrawal amount
<b>Notes</b>	
<b>Scenarios TBD</b>	<p>Daily Limit Exceeded, customer enters lesser amount successfully</p> <p>Insufficient funds in account</p>

# Guidelines for documenting

## Concentrate initially on the primary scenario

- Start from the functionality the use case is to support
- At any decision point, follow most likely course of events
- Be consistent regarding the level of detail

## Note significant points of departure informally

- These will be revisited as secondary scenarios

## No need to describe all secondary scenarios but note them

- Reference common steps from the primary scenario (do not repeat)
- Focus on the differences between the scenarios
- Can provide detailed description later as understanding of the system grows but list secondary scenarios now

## ADVANCED CONSTRUCTS IN USE CASE MODELLING

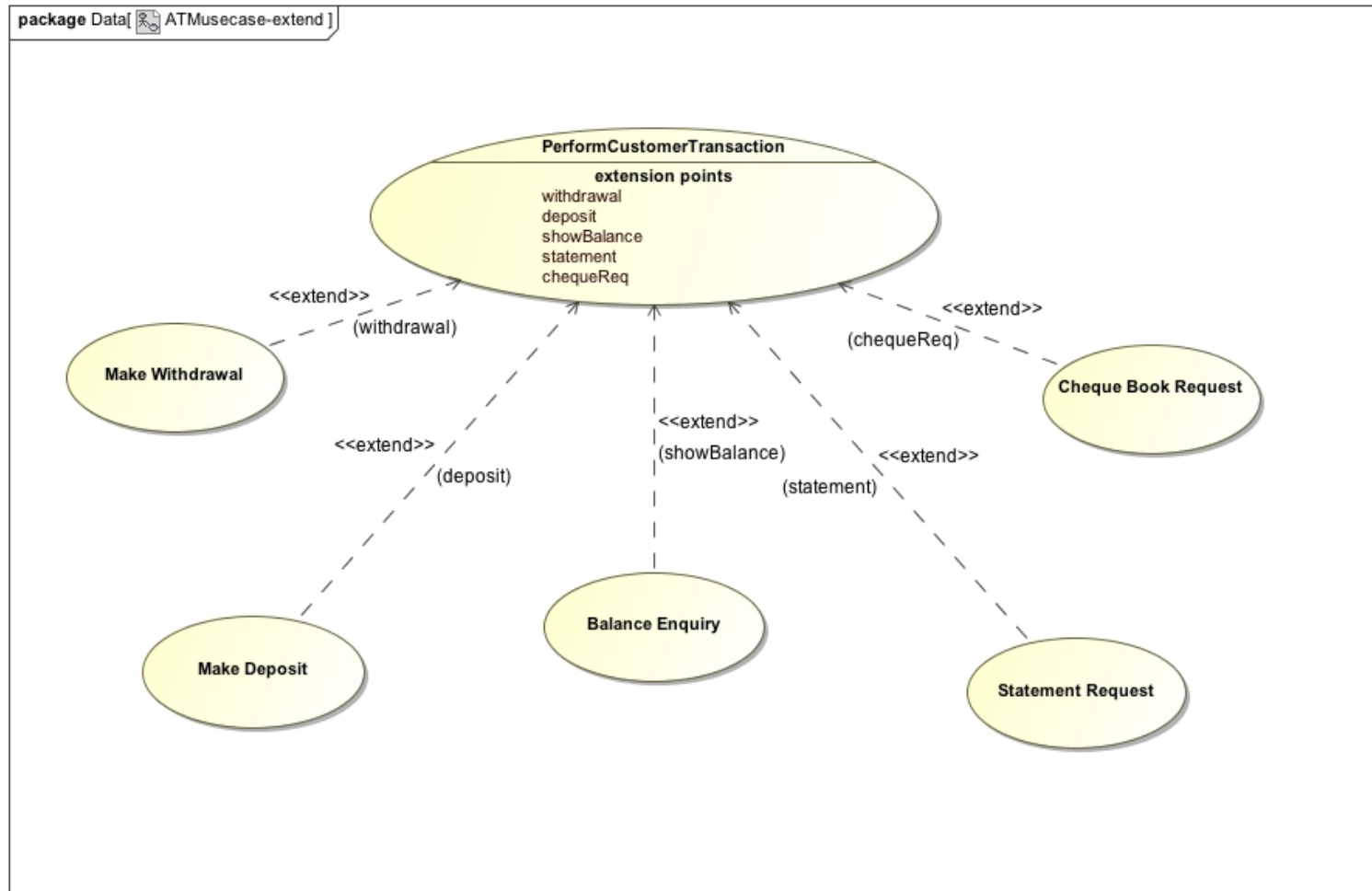
---



# Extend relationship

- A use-case may *extend* the behaviour of another use case
- Create a new use case that captures the common behaviour of the other use cases (generalization)
- The original use cases are then used to capture specialised behaviour (specialization)
  - No longer include the common behavior i.e. *no repetition!*
- Make Withdrawal and Make Deposit both require the customer to
  - insert a card, enter PIN number, and select option from menu before the main content of the use case can be performed
- This means some of the behaviour (see secondary scenarios, e.g., for Invalid PIN) would have to be repeated in many use cases.

# ATM example with <<extend>>



<b>Use Case ID</b>	ATM0
<b>Name</b>	Perform Customer Transaction
<b>Purpose</b>	To allow a bank customer to make banking transactions
<b>Initiating Actor</b>	Customer
<b>Precondition</b>	ATM is 'In Service'
<b>Invariants</b>	None
<b>Minimal guarantee</b>	Customer access is logged
<b>Requirements</b>	Banking Systems plc; ATM2000 SRS V1.0
<b>Scenario</b>	<b>Primary</b>
<b>Name</b>	Validated Session
<b>Description</b>	1 customer inserts bank card 2 customer is prompted to enter PIN to authenticate customer 3 on entry of correct PIN the system displays a list of options 4 log the fact customer has been authenticated <b>Extension Point 1</b> 5 customer selects 'Quit' option from the menu 6 customer activity is logged 7 ATM ejects card
<b>Postcondition</b>	None
<b>Notes</b>	
<b>Scenarios TBD</b>	Invalid PIN Daily Limit Exceeded Insufficient funds in account

A use case with  
<<extend>>

<b>Use Case ID</b>	ATM10
<b>Name</b>	Make Withdrawal
<b>Purpose</b>	To allow a bank customer to withdraw specific amount of cash
<b>Precondition</b>	The ATM0 use case is in progress AND the customer has selected the 'Cash Withdrawal' option AND the Cash Supply Balance > Min Withdrawal Balance
<b>Invariants</b>	None
<b>Minimal guarantee</b>	
<b>Requirements</b>	Banking Systems plc; ATM2000 SRS V1.0
<b>Scenario Name</b>	<b>Primary</b> Successful Withdrawal
<b>Description</b>	1 the system prompts the user to select an amount of cash 2 the ATM delivers the requested cash 3 returns to the ATM0 use case – menu selection (step 5)
<b>Postcondition</b>	The customer's account balance, daily withdrawal balance, and the ATM Cash Supply Balance are reduced by the withdrawal amount
<b>Notes</b>	
<b>Scenarios TBD</b>	Invalid PIN Daily Limit Exceeded Insufficient funds in account Insufficient funds in machine

An extending  
use case

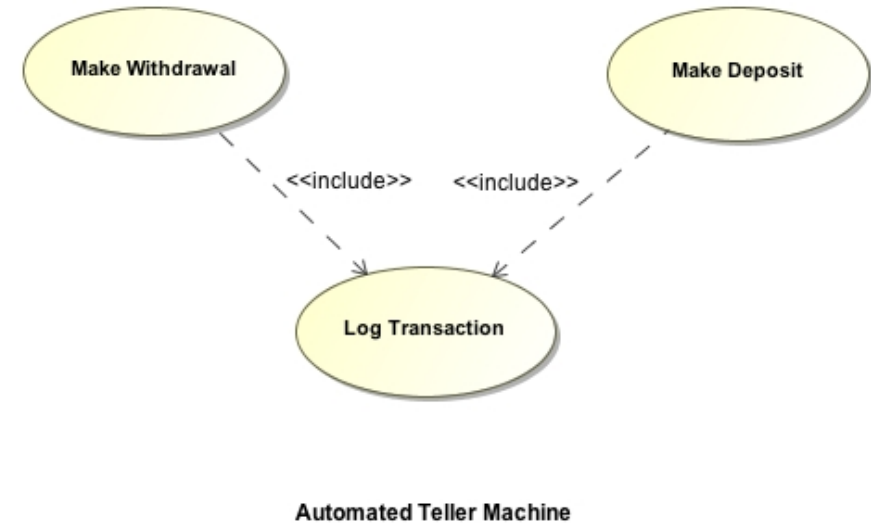
# Extension linking

- The extended use case must be complete in its own right -- no knowledge of what its extension does
- The extended use case names the point where it can be extended but does not specify which extending use case occurs
- In the extending use case, the Initiating actor is omitted
- The linking between extended and extending use cases happens at
  - the extension point (in extended use case spec)
  - the Precondition (in extending use case)
  - the Description / Steps (in extending use case)
- There is choice – the system performs one of several possible functions depending on some user-oriented selection mechanism
- A use case has non-trivial secondary scenarios
- We believe a use case will need to be extended in the future (to address changing requirements) and want to minimise the impact

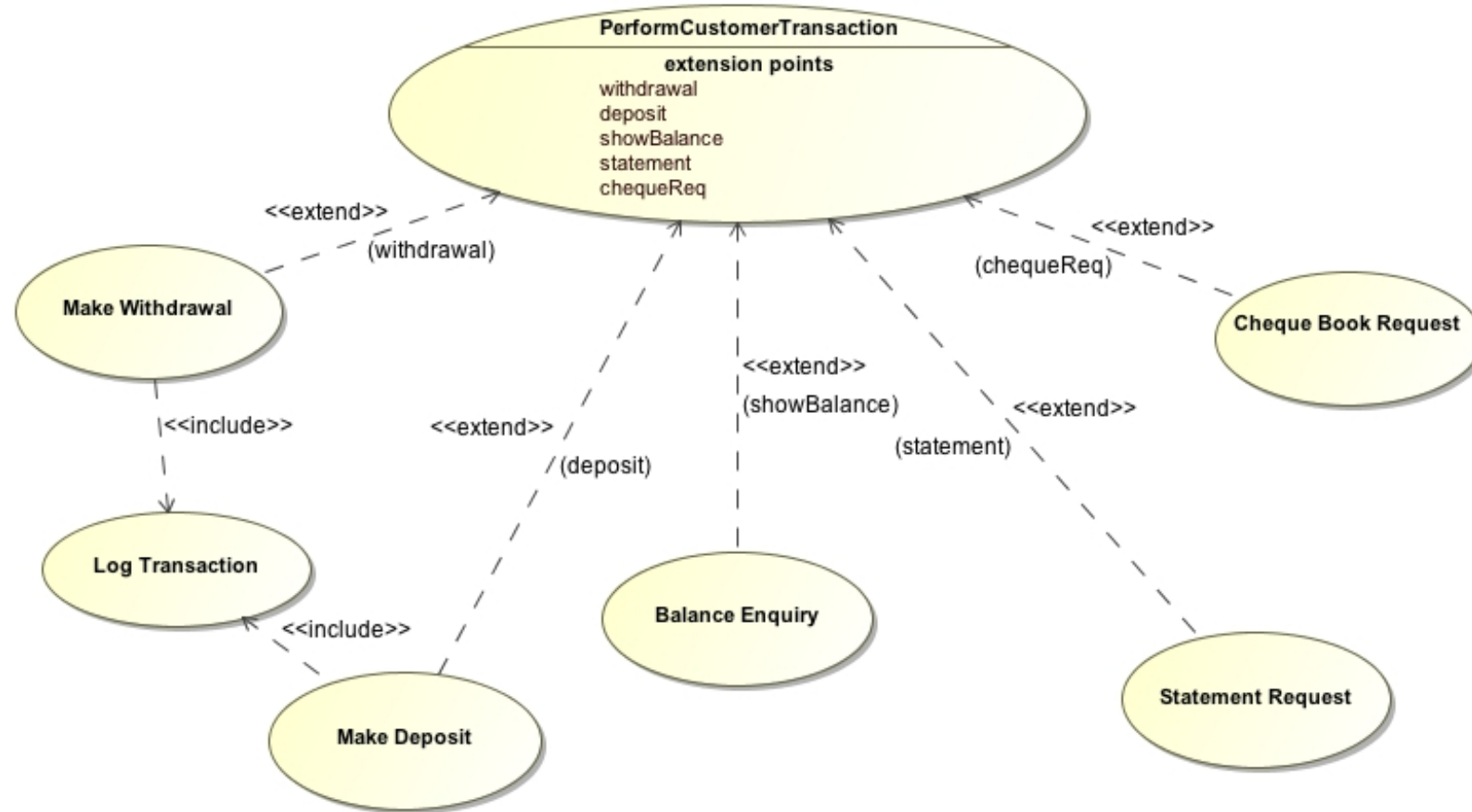
# Include relationship

- Simplify use case models by factoring out common behaviour patterns into use cases that act like shared subroutines
  - **e.g.** all customer transactions that affect the customer's account balance must be logged to separate non-volatile storage

This behaviour may be encapsulated in a common abstract use case Log Transaction and be invoked as needed by Make Withdrawal and Make Deposit



package Data[ ATMusecase-extend ]



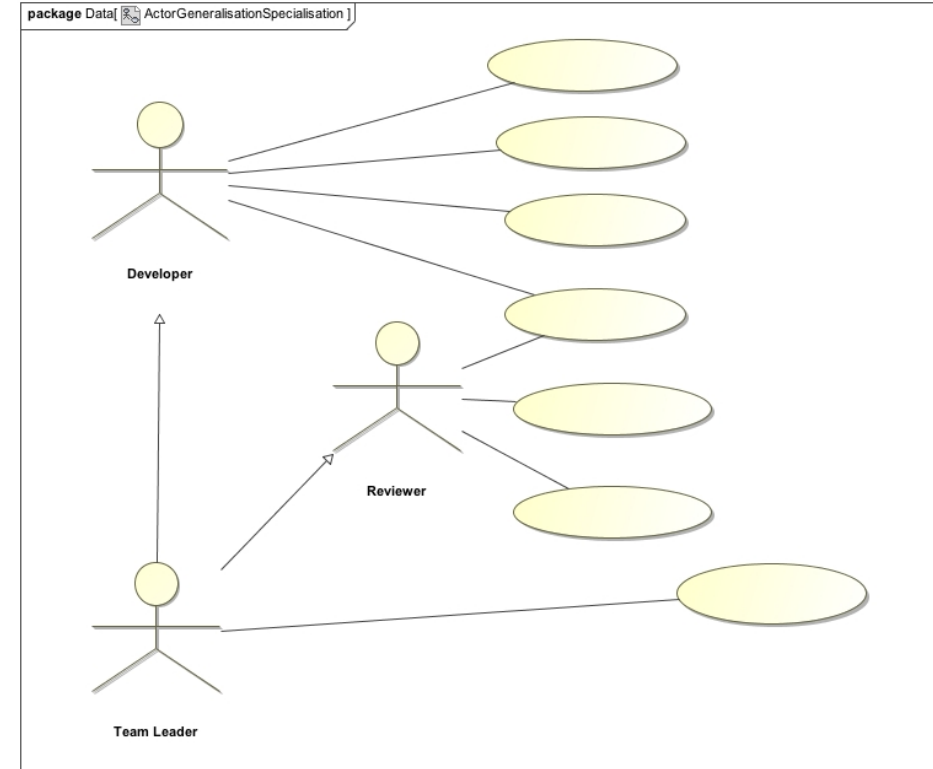
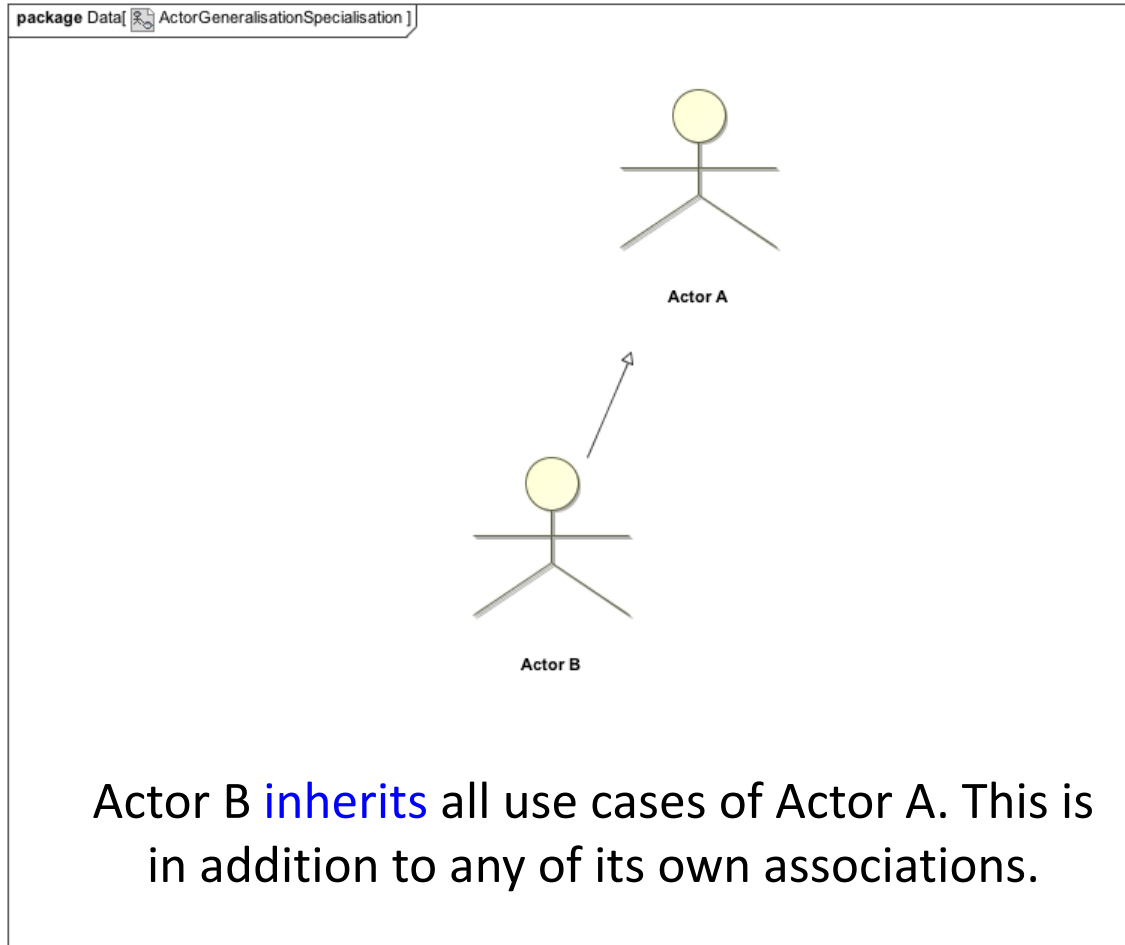
# Modeling Guidelines

---

- The included use case is ignorant of which use cases it is used by
- The including use case does not stand alone -- needs the included use case behaviour in order to be considered complete
- Contrast with <<extend>>...
- Rule of thumb:
  - Apply <<include>> when the common behaviour comes late, in the end
  - Apply <<extend>> when there is common behaviour early on that later on splits into various other more specialised behaviours



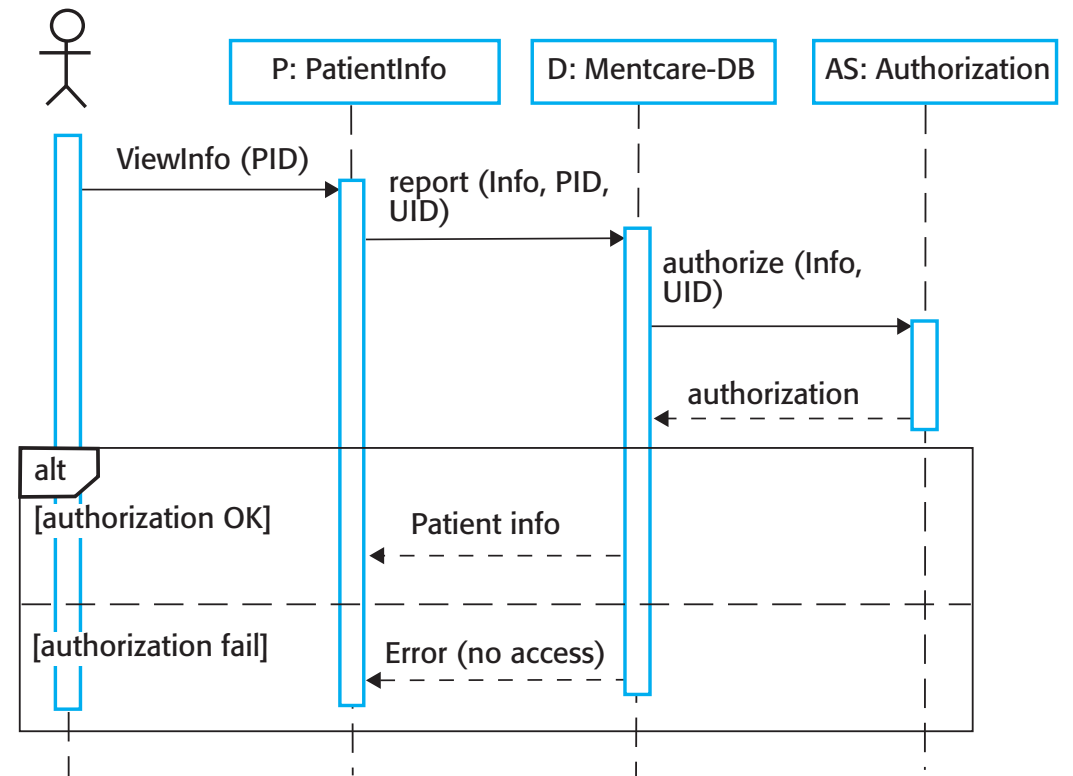
# Actor inheritance



# Sequence diagrams

- Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.

Medical Receptionist



# Structural models



# Structural models



Structural models of software display the **organization of a system in terms of the components** that make up that system and their relationships.



Structural models may be static, which show the structure of the system design, or dynamic, which show the organization of the system when it is executing.



You create structural models of a system when you are discussing and designing the system architecture.

# Class diagrams



Class diagrams are used in an object-oriented system model to show the classes in a system and their associations.



An object class can be thought of as a general definition of one kind of system object.



An association is a link between classes that indicates that there is some relationship between these classes.

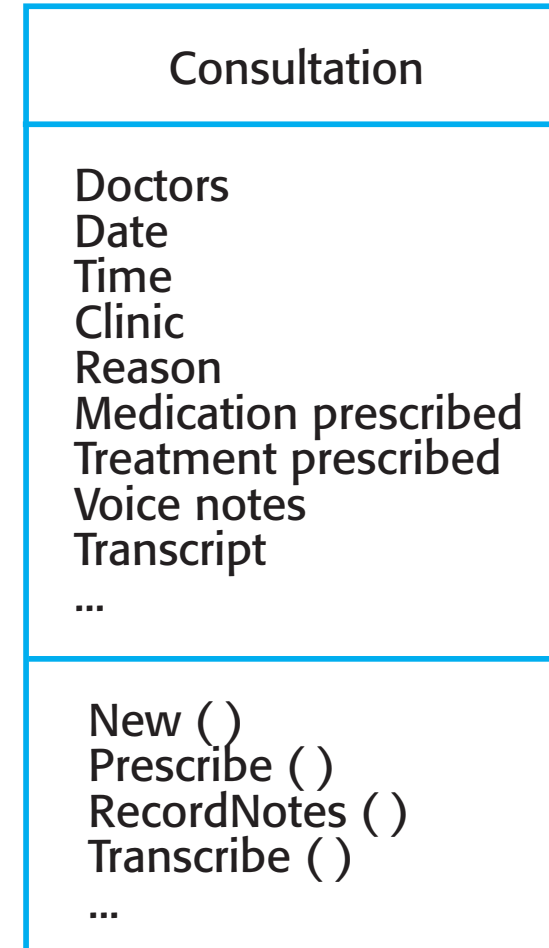
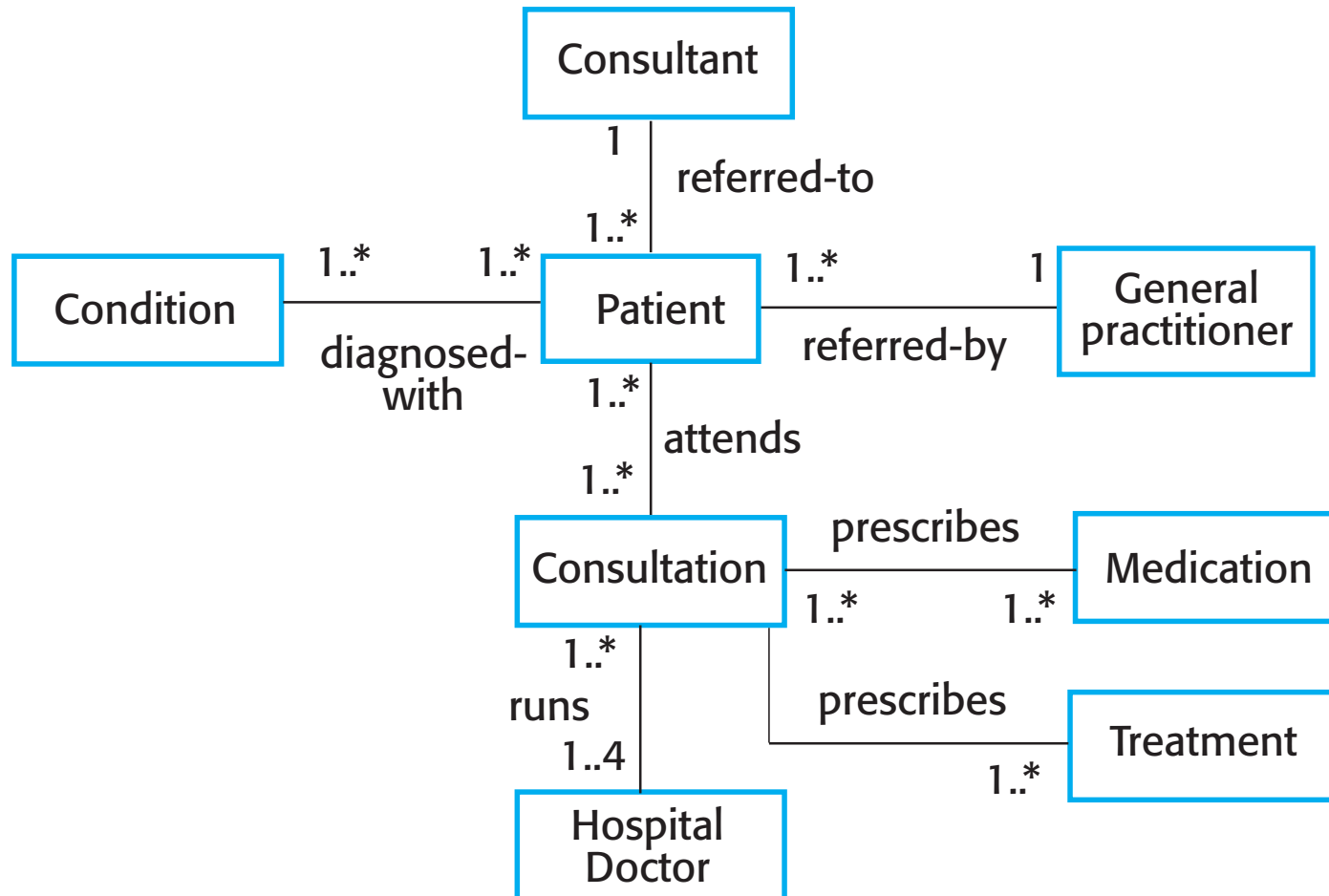


Objects typically represent something in the real world, such as a patient, a prescription, doctor, etc.

# UML classes and association

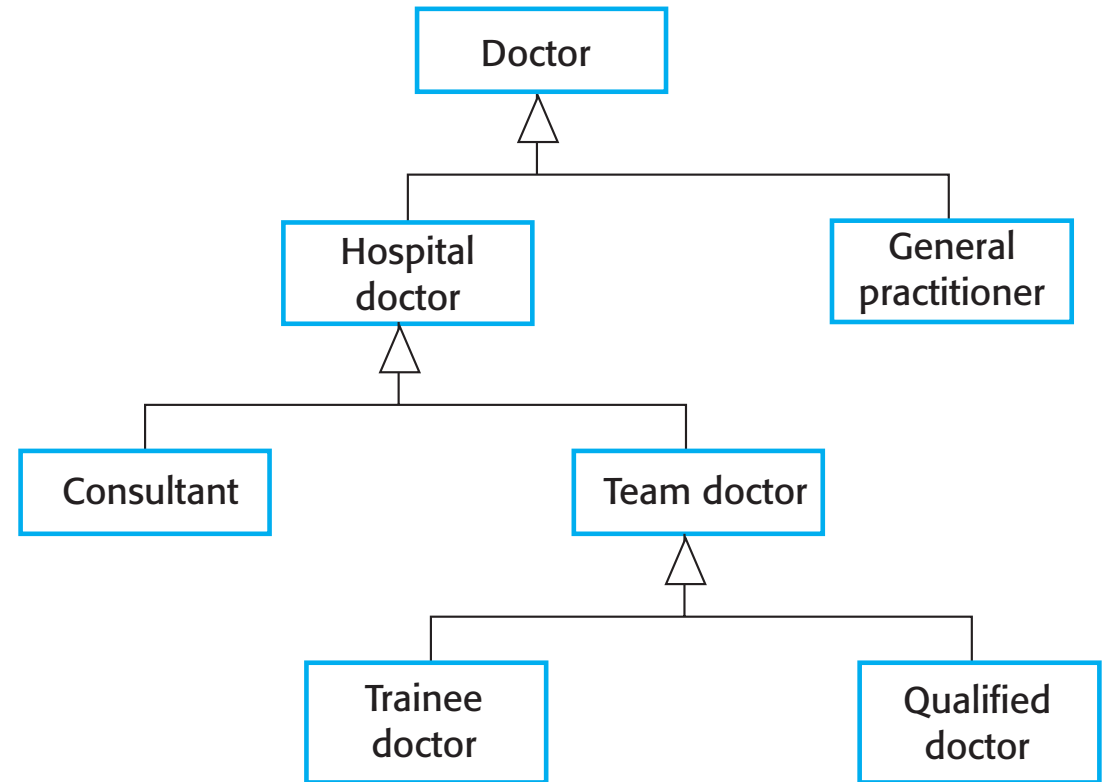


# Classes and associations



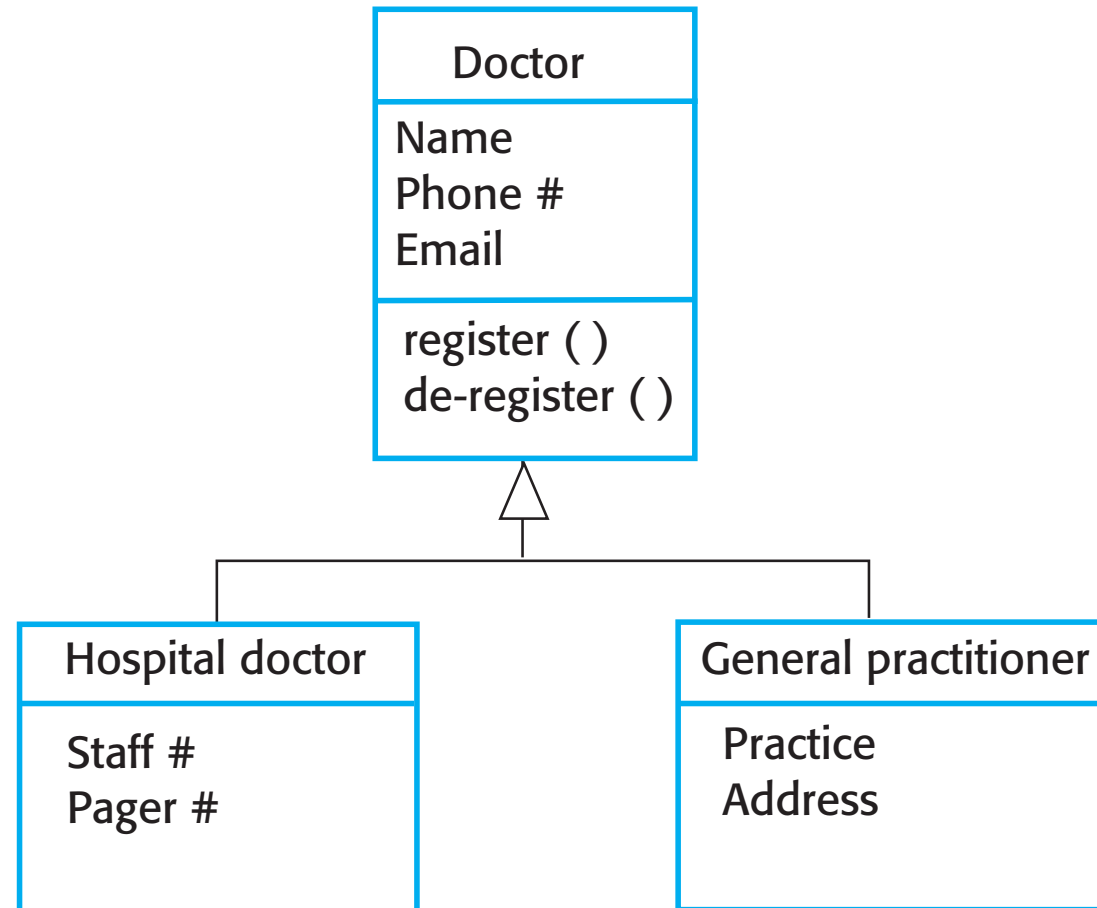
# Generalization

- We **place these entities in more general classes** (animals, cars, houses, etc.) and learn the characteristics of these classes.
- In object-oriented languages, generalization is implemented using the class inheritance mechanisms built into the language.
- The attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- The lower-level classes are subclasses inherit the attributes and operations from their superclasses.
  - These lower-level classes then add more specific attributes and operations.





# A generalization hierarchy with added detail



# Object class aggregation models

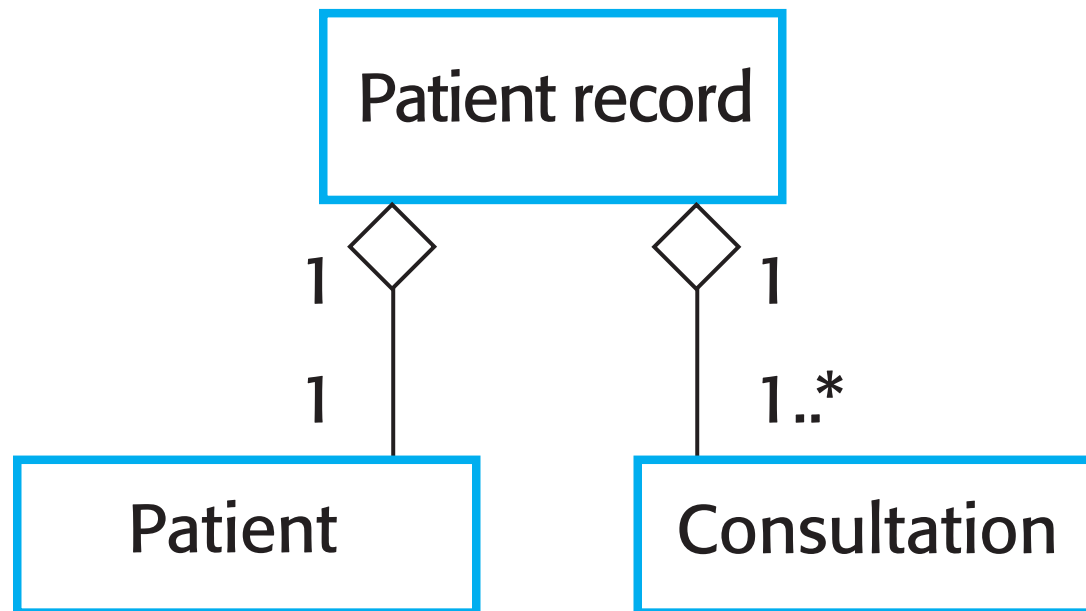


An aggregation model shows how classes that are collections are composed of other classes.



Aggregation models are similar to the part-of relationship in semantic data models.

# The aggregation association



# UML Sequence Diagrams



# Objectives

---

- To present the UML sequence diagram notation
- To illustrate uses of sequence diagrams
- To present heuristics for making good sequence diagrams
- To discuss when to use sequence diagrams

# Topics

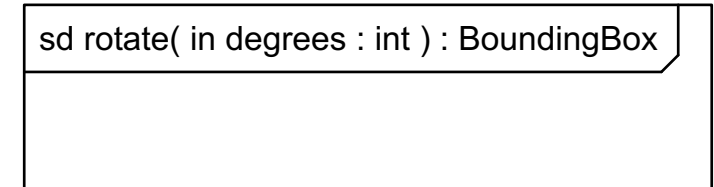
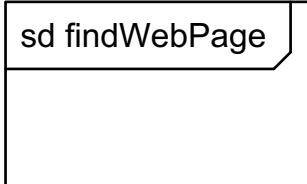
---

- UML interaction diagrams
- Frames, lifelines, and messages
- Combined fragments
- Sequence diagram heuristics
- Using sequence diagrams

# Sequence Diagram Frames

*Frame*—a rectangle with a pentagon in the upper left-hand corner called the *name compartment*.

- **sd** *interactionIdentifier*
- *interactionIdentifier* is either a simple name or an operation specification as in a class diagram



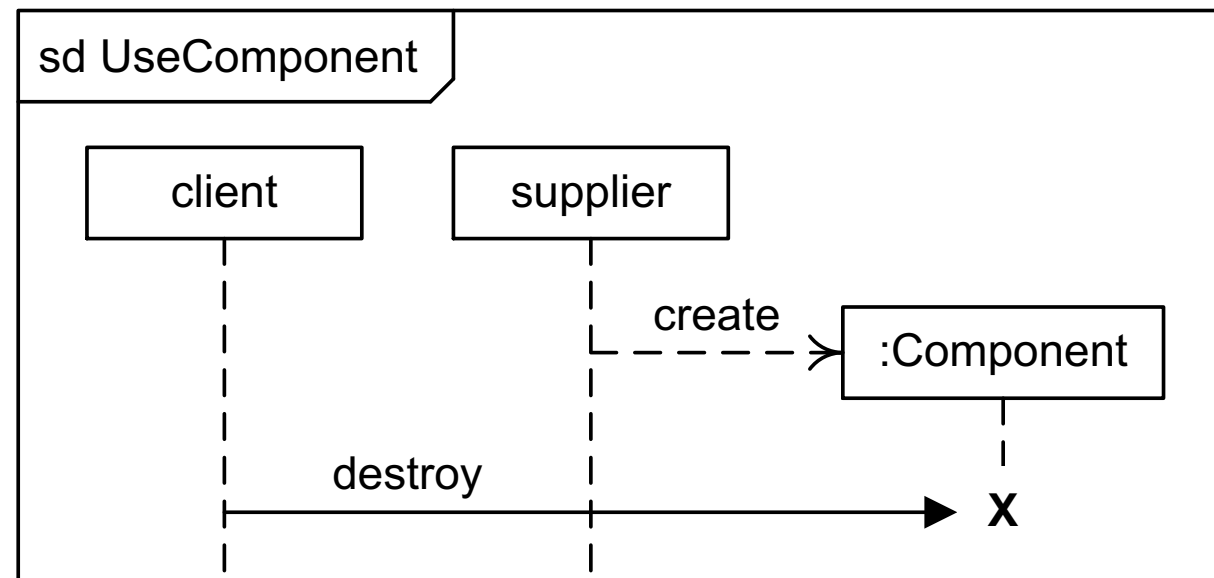
# Lifelines

---

- Participating individuals are arrayed across the diagram as *lifelines*:
  - Rectangle containing an identifier
  - Dashed line extending down the page
- The vertical dimension represents time; the dashed line shows the period when an individual exists.



# Lifelines Example



# Lifeline Creation and Destruction

---

- A new object appears at the point it is created.
  - Not clear from UML specification
- A destroyed object has a truncated lifeline ending in an **X**.
- Persisting objects have lifelines that run the length of the diagram.

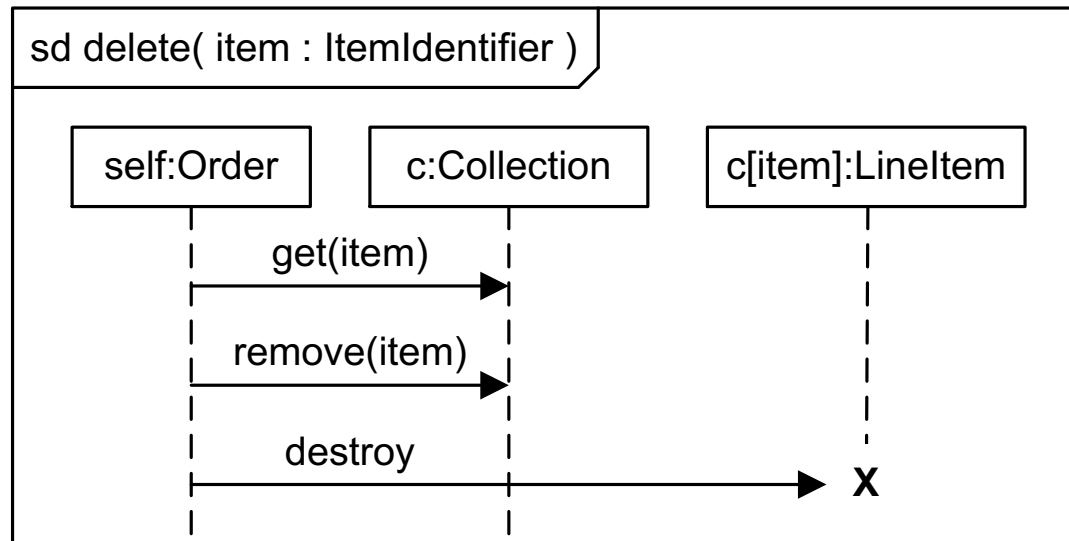
# Lifeline Identifier Format

- *name*—simple name or “self”; optional
- *selector*—expression picking out an individual from a collection
  - Format not specified in UML
  - Optional; if omitted, so are the brackets
- *typeName*—Type of the individual
  - Format not specified in UML
  - Optional; if omitted, so is the colon
- Either *name*, *typeName*, or both must appear

***name[ selector ] : typeName***

# Self

Used when the interaction depicted is  
“owned” by one of the interacting individuals

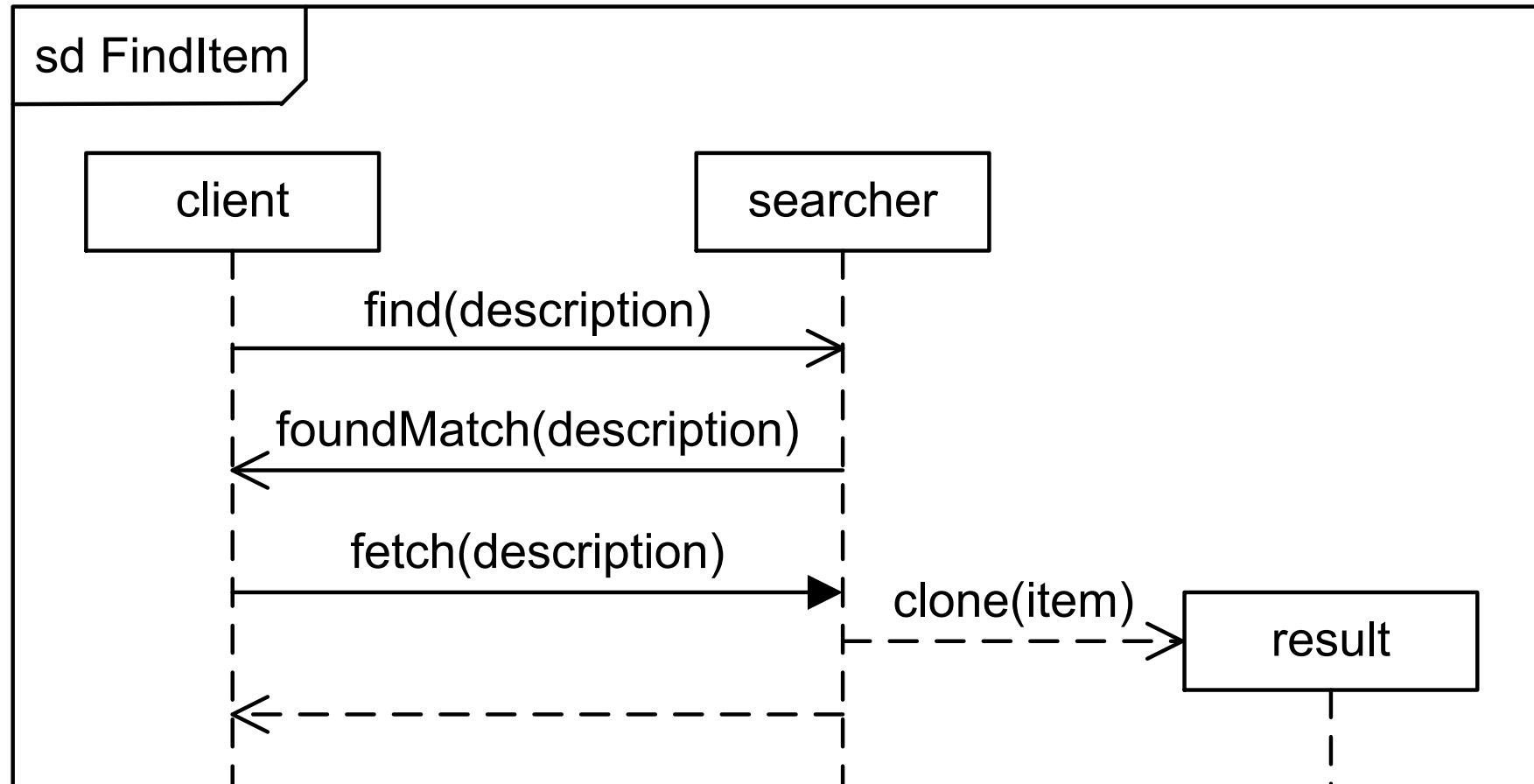


# Messages and Message Arrows



- *Synchronous*—The sender suspends execution until the message is complete
- *Asynchronous*—The sender continues execution after sending the message
- *Synchronous message return or instance creation*

# Message Arrow Example

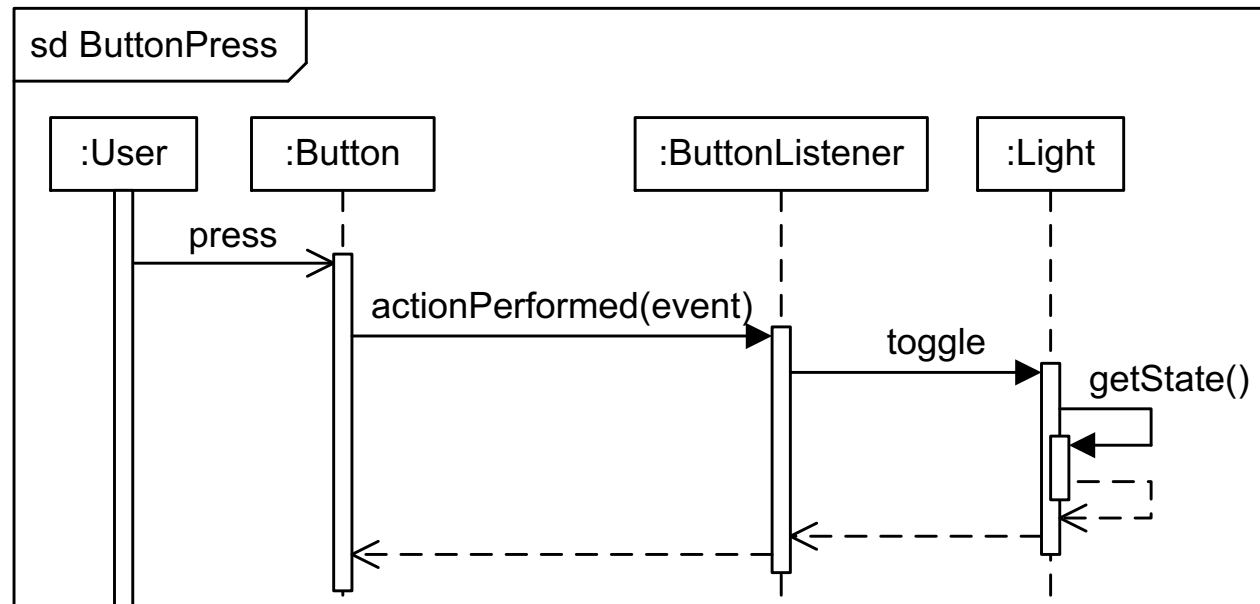


# Execution Occurrences

---

- An operation is **executing** when some process is running its code.
- An operation is **suspended** when it sends a synchronous message and is waiting for it to return.
- An operation is **active** when it is executing or suspended.
- The period when an object is active can be shown using an *execution occurrence*.
  - Thin rectangle over lifeline dashed line

# Execution Occurrence Example



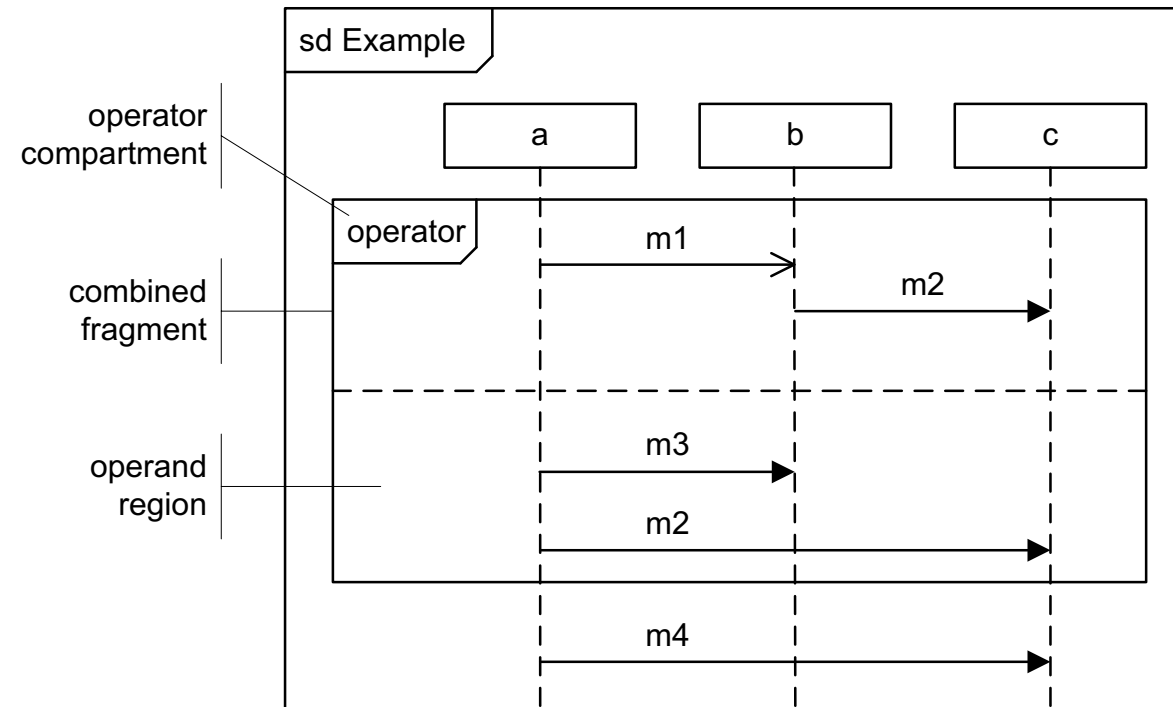


# Combined Fragments

---

- A combined fragment is a marked part of an interaction specification that shows
  - Branching,
  - Loops,
  - Concurrent execution,
  - And so forth.
- It is surrounded by a rectangular frame.
  - Pentagonal operation compartment
  - Dashed horizontal line forming regions holding operands

# Combined Fragment Layout

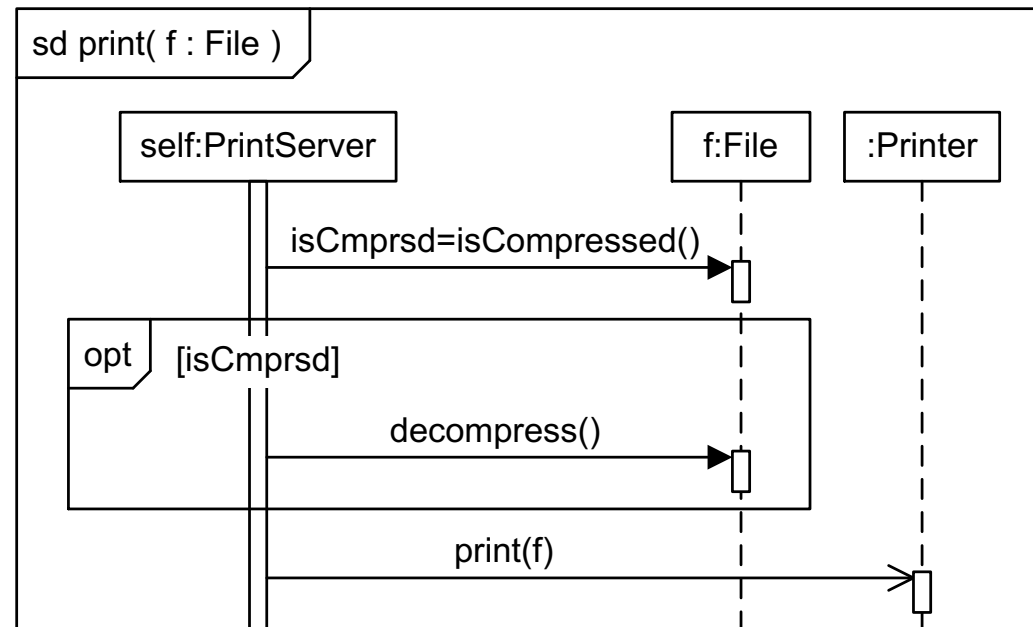


# Optional Fragment

---

- A portion of an interaction that may be done
  - Equivalent to a conditional statement
  - Operator is the keyword `opt`
  - Only a single operand with a guard
- A *guard* is a Boolean expression in square brackets in a format not specified by UML.
  - `[else]` is a special guard true if every guard in a fragment is false.

# Optional Fragment Example

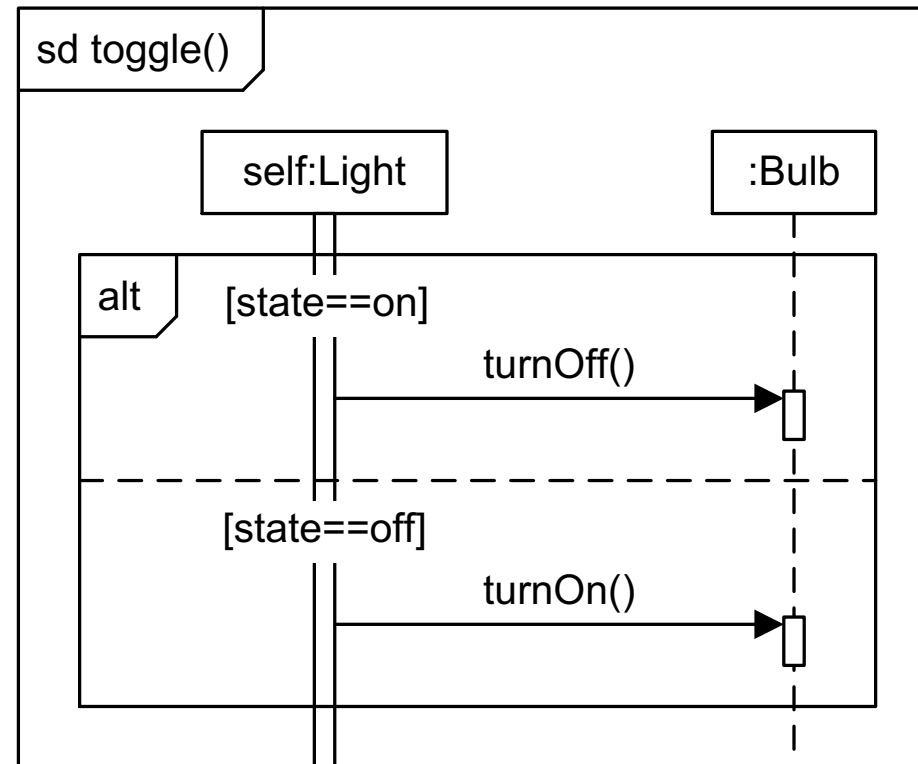


# Alternative Fragment

---

- A combined fragment with one or more guarded operands whose guards are mutually exclusive
  - Equivalent to a case or switch statement
  - Operator is the keyword alt

# Alternative Fragment Example

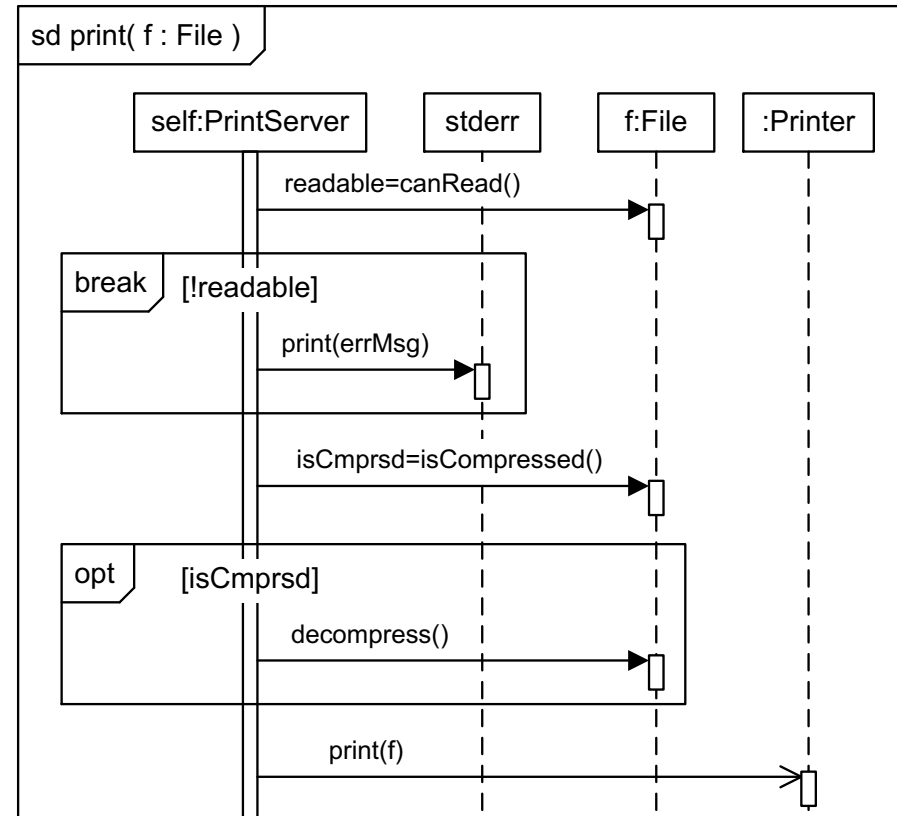


# Break Fragment

---

- A combined fragment with an operand performed in place of the remainder of an enclosing operand or diagram if the guard is true
  - Similar to a break statement
  - Operator is the keyword break

# Break Fragment Example





# Loop Fragment

---

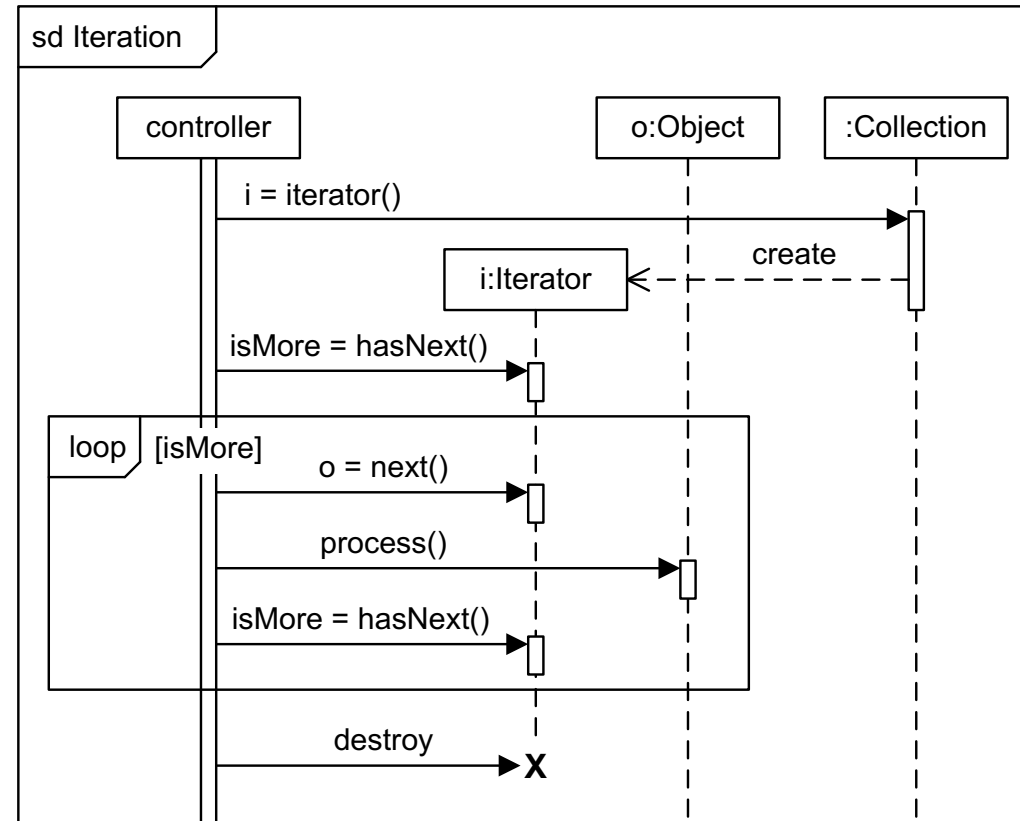
- Single loop body operand that may have a guard
- Operator has the form `loop( min, max )` where
  - Parameters are optional; if omitted, so are the parentheses
  - *min* is a non-negative integer
  - *max* is a non-negative integer at least as large as *min* or \*; *max* is optional; if omitted, so is the comma

# Loop Fragment Execution Rules

---

- The loop body is performed at least *min* times and at most *max* times.
- If the loop body has been performed at least *min* times but less than *max* times, it is performed only if the guard is true.
- If *max* is \*, the upper iteration bound is unlimited.
- If *min* is specified but *max* is not, then *min*=*max*.
- If the loop has no parameters, then *min*=0 and *max* is unlimited.
- The default value of the guard is true.

# Loop Fragment Example



# Design Patterns

Santanu Dash

# Software Development

---

- Software developers spend a significant amount of time
  - Writing code; and
  - Understanding code
- How can we ease this process for them?
  - Code should follow established norms wherever possible
  - Following norms normally means the code is
    - Easy to read
    - Easy to write
    - Easy to extend; and
    - Easy to maintain

# The Power of Patterns

---

*CN*    *NIB*    *MAP*    *PLE*

# The Power of Patterns

---


*CNN*

*IBM*

*APPLE*

# The Power of Patterns

*CN NIB MAP PLE*



*CNN IBM APPLE*

- There are many other potential “patterns”, but these somehow **succeeded** to become 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

---

- Patterns capture the existing best practices, rather than inventing untried procedures.
- Proven solution recipes from similar problems
- Patterns are used primarily to *improve existing designs or code by rearranging it according to a “pattern.”*
- Developer gains
  - *efficiency*, by avoiding a lengthy process of trials and errors in search of a solution, and
  - *predictability* because this solution is known to work for a given problem.

# Topics

---

- Publisher-Subscriber (a.k.a. Observer)
- Command Pattern
- Decorator Pattern
- State Pattern
- Proxy Pattern
  - Protection Proxy

# Software Changes: What they could be?

- Software change is needed to keep up with the reality
  - Change may be triggered by changing business environment, or
  - To refactor existing code
- **Change in-the-large:** What changes in real world are business rules → customer requests changes in software
- **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

# Software Changes: How should they be done?

---

- Change may have unintended consequences when they are not cohesive with existing software
  - This could be due to unrelated responsibilities
  - Change in code implementing one responsibility can unintentionally lead to faults in code for another responsibility
- What can the developer do?
  - Look for established ways in which the change can come about
  - Or, in other words, follow **design 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

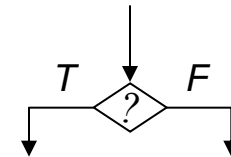
---

- 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

*Like many other design patterns*

- 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”



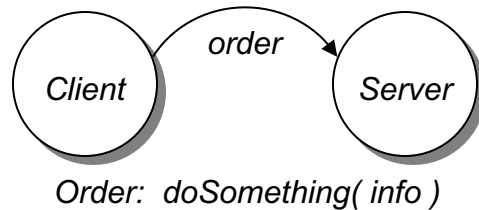


# Request- vs. Event-Based Communication

## 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:*



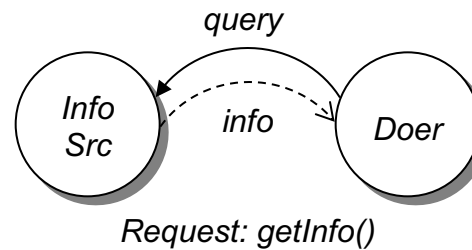
(a)

(tight coupling)

## Direct Communication

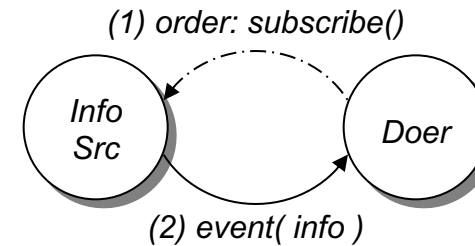
*Directly ordering the server what to do or demanding information from a source*

*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:*



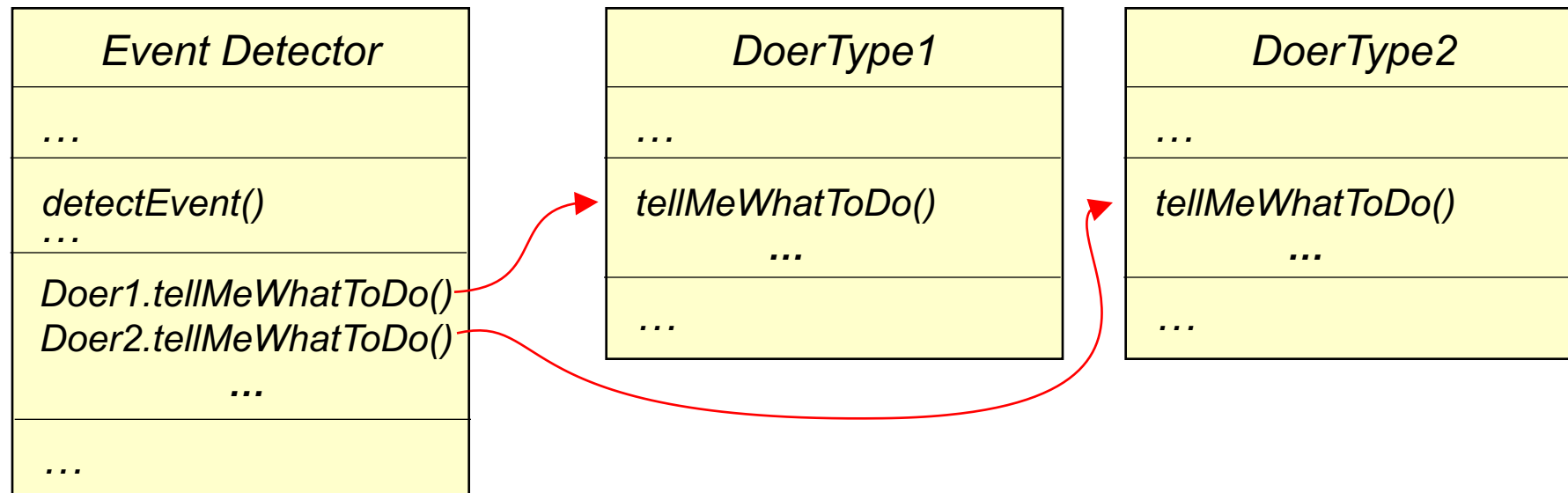
(c)

(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:

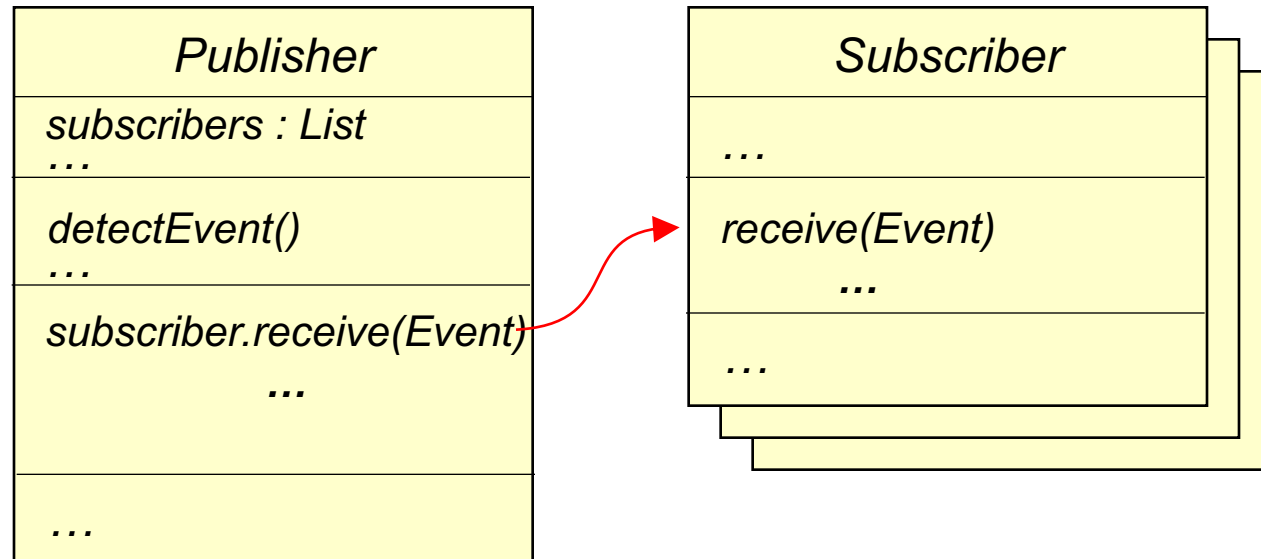
- Tell Doer-1 what to do
- Tell Doer-2 what to do

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!)*

# Labour Division (1)

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

developer X



developer Y



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	—

## Labour Division (2)

developer X



developer Y

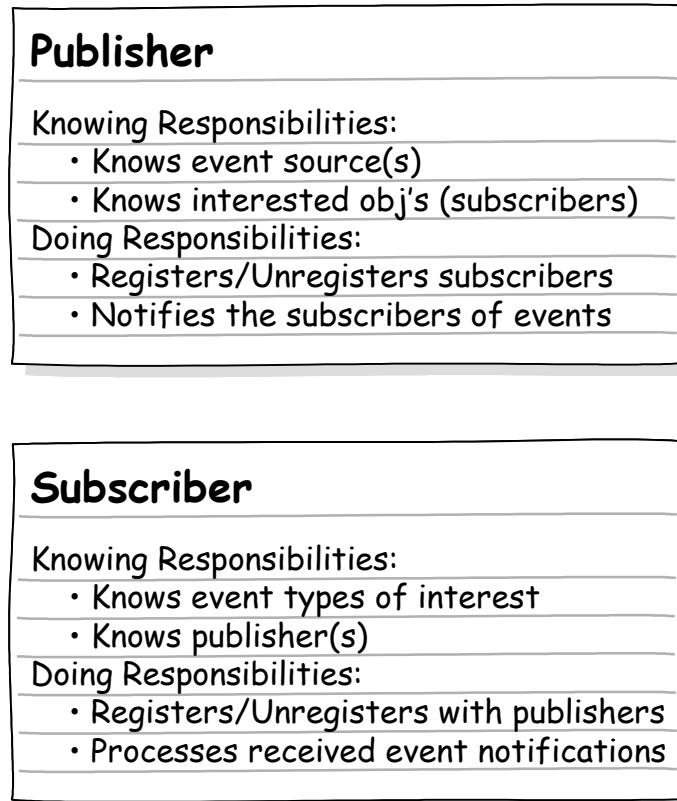


- After: Separation of developers' duties
- Developer **Y** is completely responsible for class **B**

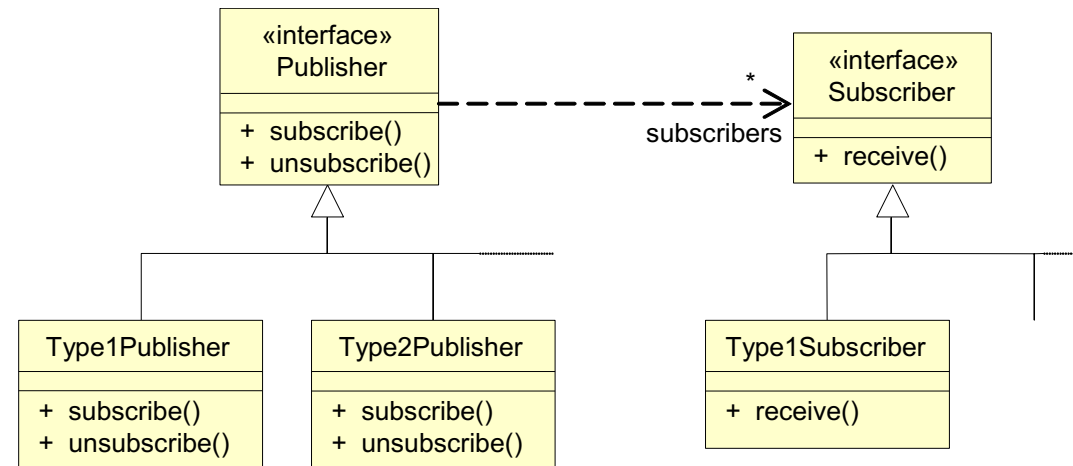
Class	A	B
Who develops	developer X	developer Y
Who knows which methods of B to call, when, how	—	developer Y
Who makes changes when classes of type B modified or new sub-types added	—	developer Y

# Pub-Sub Pattern

*Reports incoming events to subscribers*

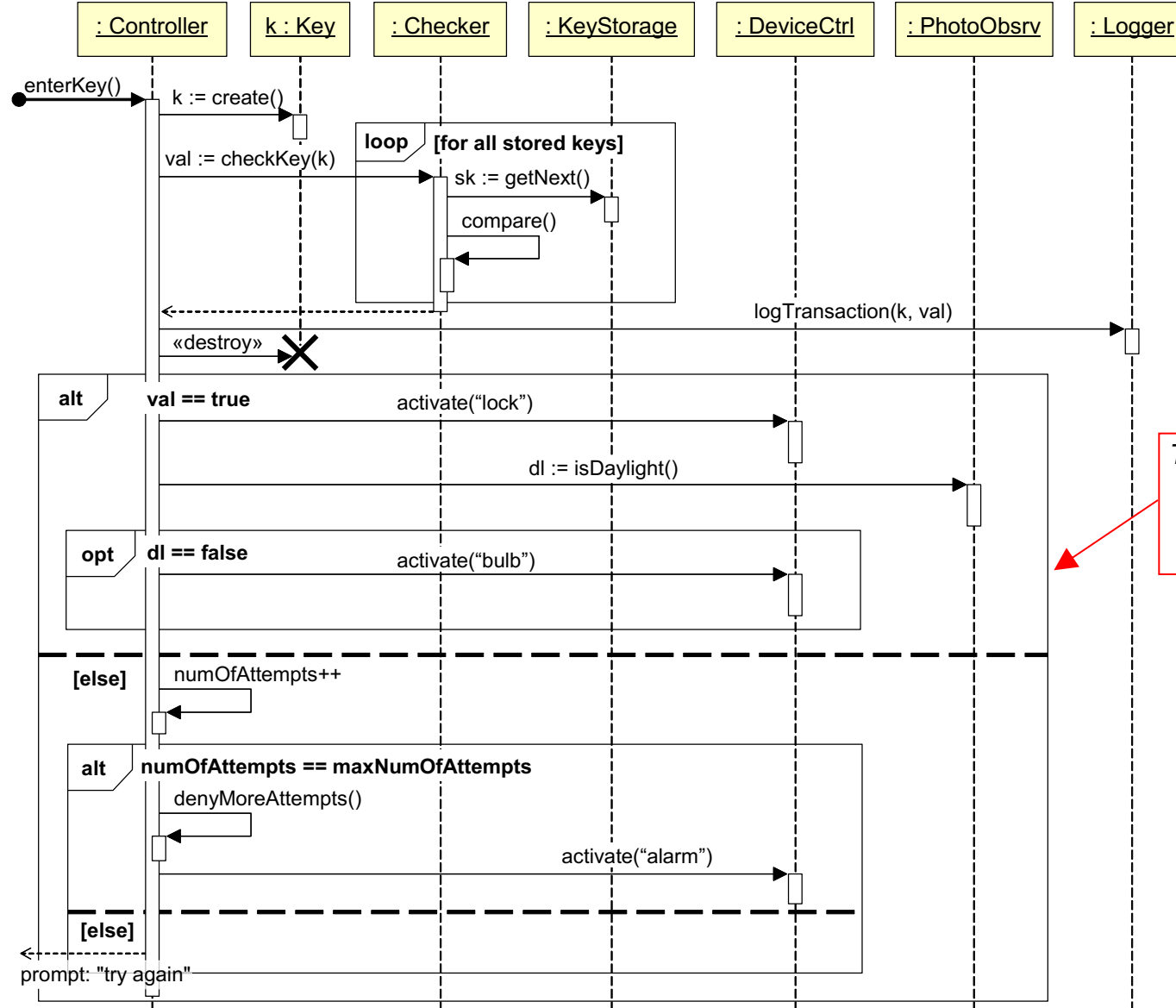


(a)



(b)

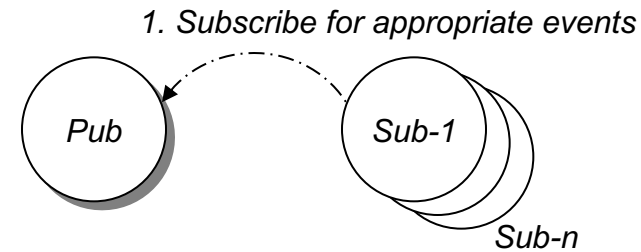
# Unlock Use Case



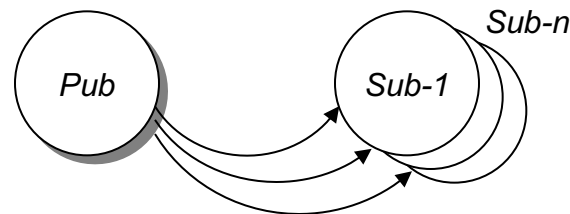
*This IF-THEN-ELSE  
must be changed  
when a new device  
type is introduced*

# Refactoring to Publisher-Subscriber

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



Event type	Subscriber
keysValid	LockCtrl, LightCtrl
keysInvalid	AlarmCtrl
itIsDarkInside	LightCtrl
.	
.	
.	



No need to consider the “appropriateness” of calling the “servers”

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

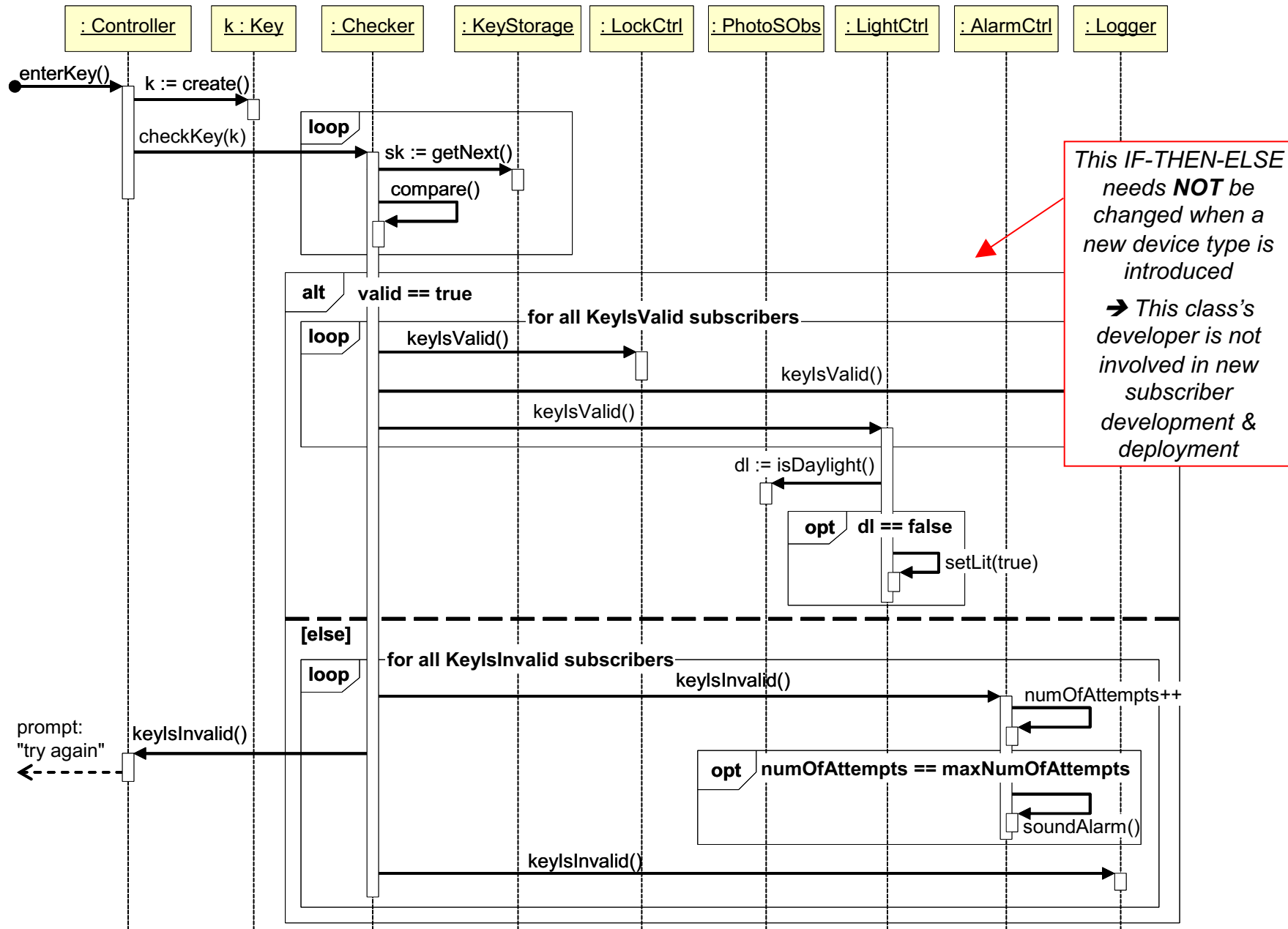
Design-time decisions are better than runtime decisions, because they can be easier checked if they “work” (before the product is developed & deployed)

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

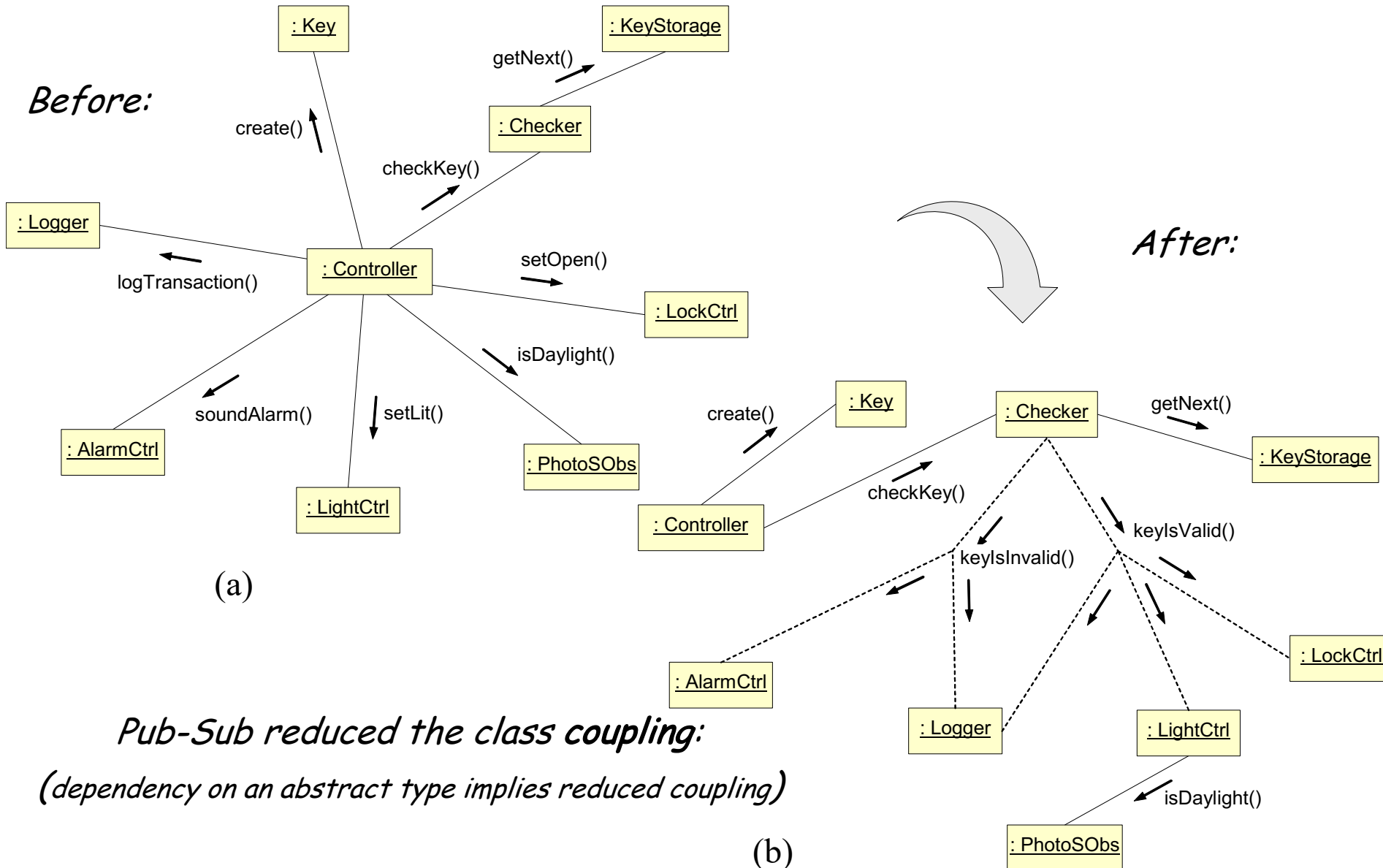
Complexity due to decision making cannot disappear;  
It is moved to the design stage, which is early and easier to handle.  
The designer's choice is then hard-coded, instead of checking runtime conditions.



# Pub-Sub: Unlock Use Case



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



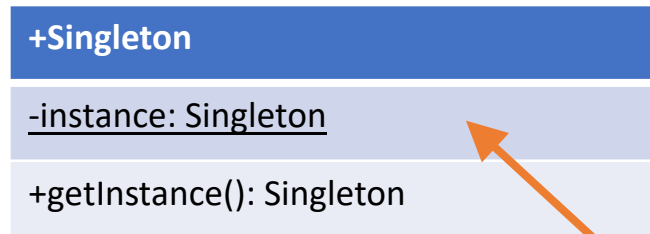
*Singleton*

# Singleton

---

- Purpose
  - To ensure that a class has only **one instance**
- Why?
  - Often used to represent system resources
  - For example, the console output or a printer, ...
- How?
  - Singleton class looks after its **own instance**
  - Only the class itself can instantiate an object

# Example Class Diagram



*Class: Singleton*

*Attribute: private instance of Singleton*

*Method: returns reference to instance*

*Underline shows "static"*

# Example Code

```
public class Singleton { // Comments omitted.
```

*Static property – belongs to the class*

```
    private static Singleton instance = new Singleton();
```

```
    private Singleton() {  
        // Do stuff.  
    }
```

*Private default constructor – prevents any other  
class from creating a Singleton object*

```
    public static Singleton getInstance() {  
        return Singleton.instance;  
    }
```

*Returns the single instance*

```
    //add methods as required  
}
```

# Singleton

---

- Strengths
  - Provides exactly **one instance** of a class
  - Single point of **access** and **control**
  - Can be written to allow subclasses

# Examples in Java

---

- Not strictly singletons (no access method):
  - System.in
  - System.out
  - System.err
- But all singleton instances
  - All are static properties of the System class
  - System class cannot be instantiated



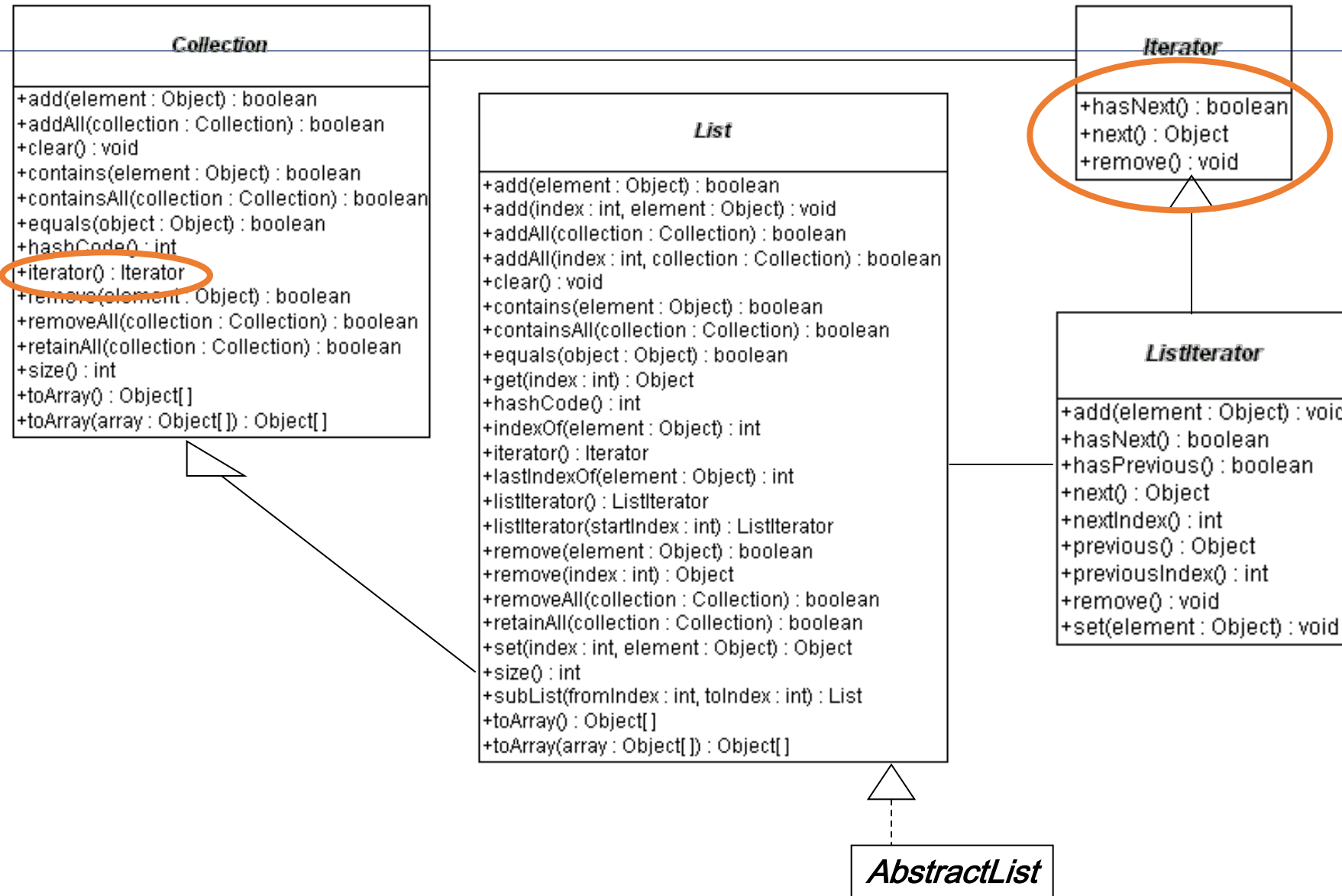
# *Iterator*

# Iterator

---

- Purpose
  - To allow **sequential access** to elements of an aggregate without exposing the implementation
  - Behavioural pattern
- Why?
  - Different implementations (array, list, ...)
  - To give a consistent way of accessing the elements
- How?
  - By providing an interface to a set of iterator methods
  - Traversal of the elements in the hands of the iterator

# Example Class Diagram



# Example Code

```
public class Basket { // Code and comments omitted.
```

```
    public void displayBasket() {
```

*Get the iterator from the aggregate (a set in this case)*

```
        Iterator<Item> iterator = basket.iterator();
```

```
        System.out.print("Basket data: ");
```

*Loop through each item using next() and hasNext()*

```
        // Displaying the tree set data.
```

```
        while (iterator.hasNext()) {
```

```
            System.out.print(iterator.next().toString() + " ");
```

```
        }
```

```
    }
```

```
}
```

# Iterator

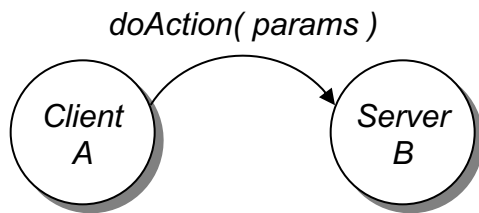
- Strengths
  - Simple and consistent access to a **collection**
  - Control is with the **iterator**, not the aggregate
  - Iterators can provide **immutable** access  
(just disable the `remove()` method. Note that any returned object is still mutable, though)
  - You can have **more than one iterator** on a collection at any one time
- Weaknesses
  - Iterators imply **no order** – this is up to the aggregate
  - Modifying the collection while iterating can be dangerous
    - **Concurrency anomalies**
- Java Collections Framework
  - All implement iterators
  - For each statement ( `for (type var : coll){//body of loop}` )

# Practical Issues

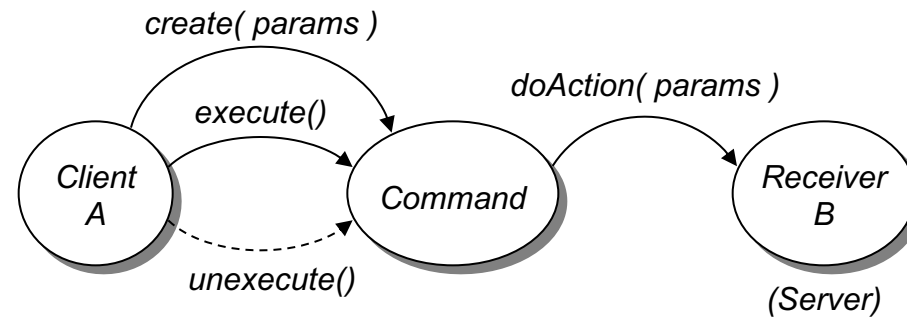
---

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!

# Command Pattern Motivation



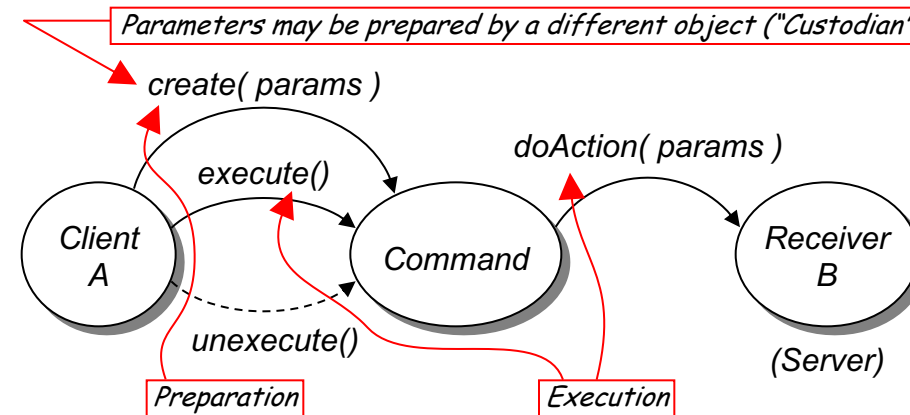
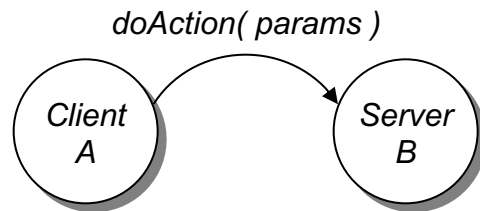
(a)



(b)

# Command Pattern Motivation

- **Motivation:** To separate parameter preparation from passing program control (decision on when to call)



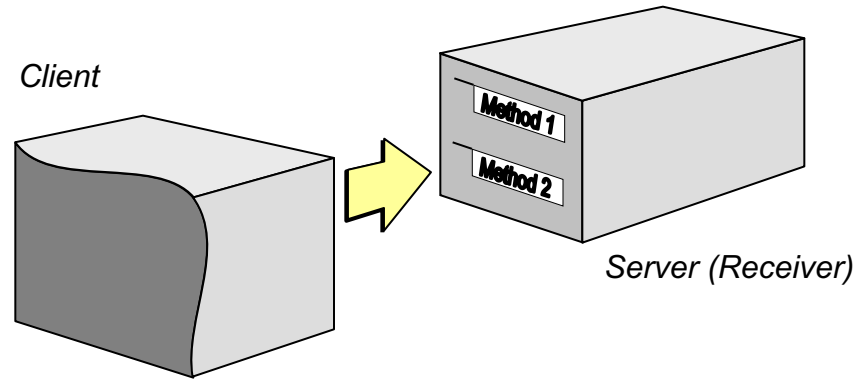
- **Reasons for separation:**

- Separate preparation of calling parameters (which may become available much before the execution time or may become available incrementally)
  - All calling parameters become localized in a Command object ("encapsulated")
  - Parameters may be prepared for the Client by a different object ("Custodian")
  - Client and Custodian objects' codes may evolve separately
    - I.e., different developers develop and maintain or upgrade these classes
- May prepare all Commands in a list (with different parameters or different Receivers) and simply iterate through the list to execute all
- For un-execute (roll-back) capability



# Command Pattern Motivation

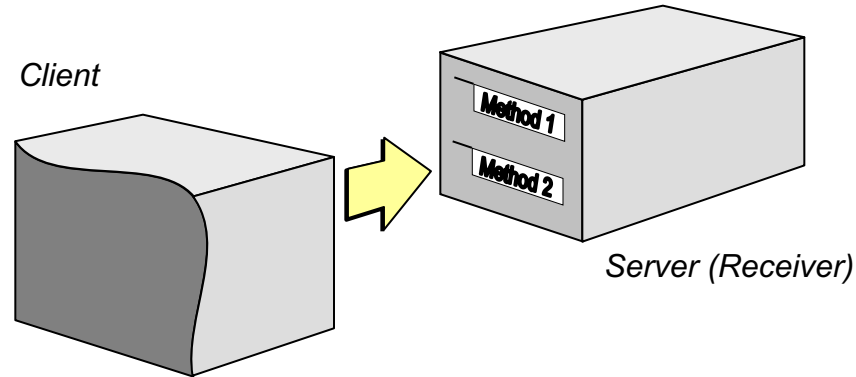
*Before:*



- Problem:  
Variable and evolving method signature
  - If Server code changes, Client code needs to change, too

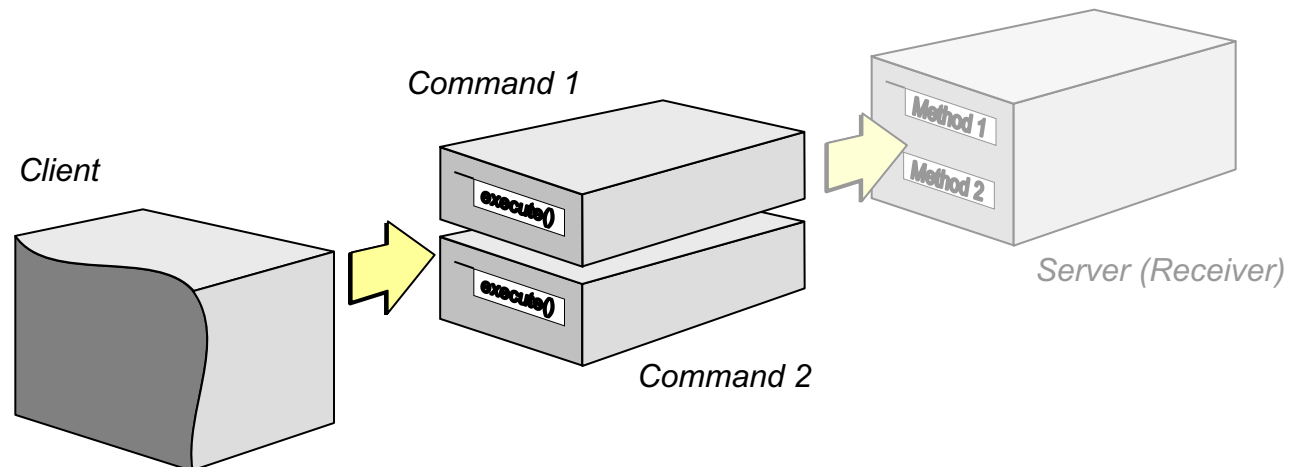
# Command Pattern Improvement

*Before:*



*The change towards the Cmd. pattern may not appear radical when viewed overall, but looking from the client's standpoint, the simplification is significant.*

*After:*



➔ *The interface to the Server object is much simpler.*

# Example – The Interface

```
//Command
public interface Command{
    public void execute();
}
```

```
//Concrete Command
public class LightOnCommand
implements Command{
    //reference to the light
    Light light;
    public LightOnCommand(Light light){
        this.light = light;
    }
    public void execute(){
        light.switchOn();
    }
}
```

```
//Concrete Command
public class LightOffCommand
implements Command{
    //reference to the light
    Light light;
    public LightOffCommand(Light
light){
        this.light = light;
    }
    public void execute(){
        light.switchOff();
    }
}
```

# Example – The Receiver and Invoker

```
//Receiver
public class Light{
    private boolean on;
    public void switchOn(){
        on = true;
    }
    public void switchOff(){
        on = false;
    }
}
```

```
//Invoker
public class RemoteControl{
    private Command command;
    public void setCommand(Command
command){
        this.command = command;
    }
    public void pressButton(){
        command.execute();
    }
}
```

# Example – The Client

```
//Client
public class Client{
    public static void main(String[] args) {
        RemoteControl control = new RemoteControl();
        Light light = new Light();
        Command lightsOn = new LightsOnCommand(light);
        Command lightsOff = new LightsOffCommand(light);
        //switch on
        control.setCommand(lightsOn);
        control.pressButton();
        //switch off
        control.setCommand(lightsOff);
        control.pressButton();
    }
}
```

# Command Pattern Improvement

---

- Command provides a uniform method signature (“interface”) to the Server
  - The Command interface never changes, so Server changes do not force Client changes
- Client only decides **when** to execute()
- Client evolution is decoupled from Server implementation and Command implementation
  - Client versus Server/Command can be responsibilities of different developers

# Command Pattern

## Command

### Knowing Responsibilities:

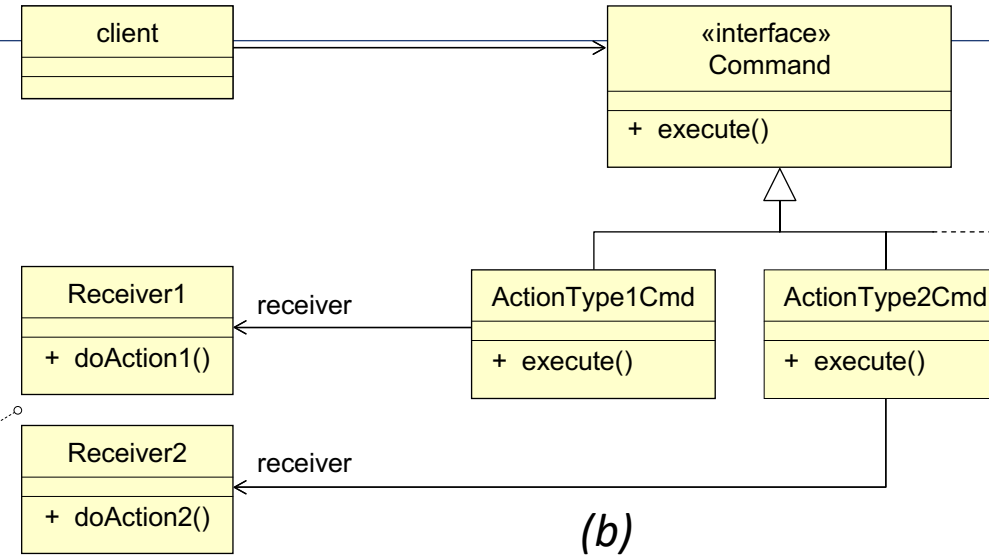
- Knows receiver of action request
- Optional: May know whether action is reversible

### Doing Responsibilities:

- Executes an action
- Optional: May undo an action if it is reversible

(a)

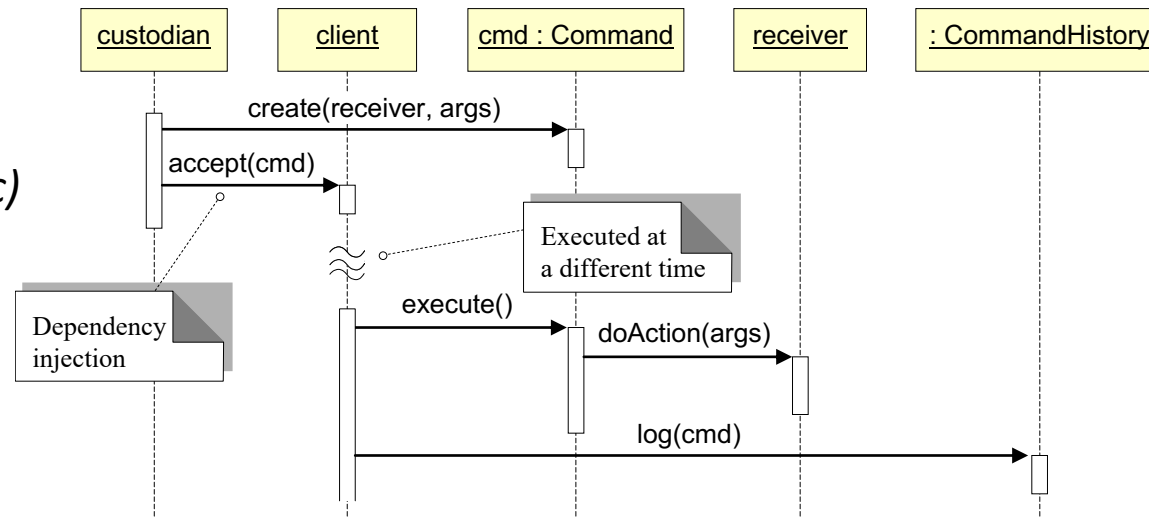
“Servers”



(b)

Forward  
execution  
(do)

(c)



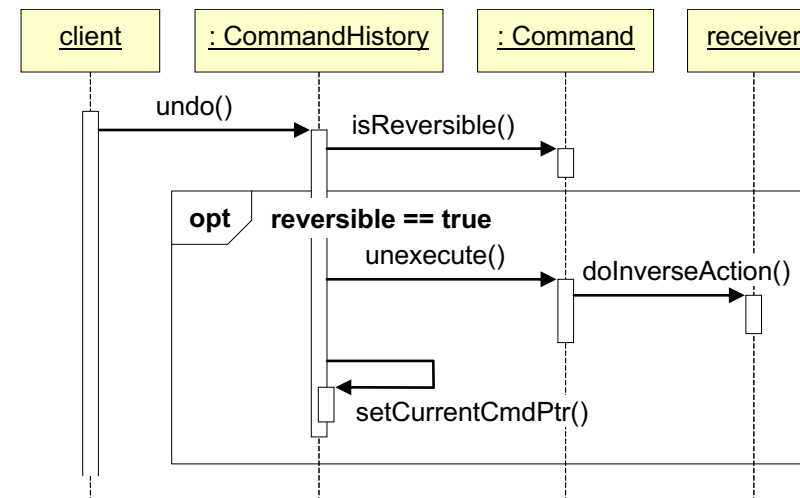
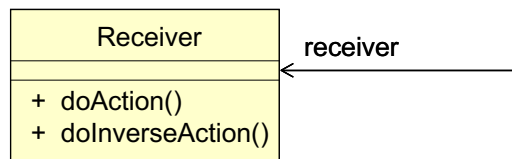
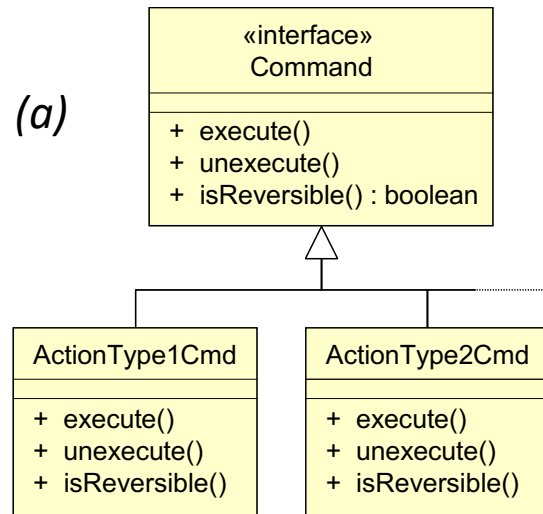
# Command Advantages

- Execution is usually called with other business logic
  - Now it is decoupled from parameter preparation, which can be done at another place (“staging area”), not interfering with business logic (“execution area”)
  - ➔ Business logic is decoupled from parameter preparation
    - Client and Custodian codes may evolve independently, by different developers
- NOTE:
  - The Command method `execute()` does not return a result
    - Through a Command, the client gives an order and does not expect to be answered back
  - If the result of a command execution is needed, it should be retrieved separately, after `execute()` returns



# Command Pattern Interaction

- Often commands can be **reversed/undone**
- Extended interface to check for reversibility and, if true, undo



(b)  
Reverse execution  
(undo)