ESE

exam preparation

Andrea Caracciolo
http://scg.unibe.ch/staff/caracciolo

UNIVERSITÄT
BERN

Exam

- 8th January, 2014 — ExWi A6 @ 11h00
- Register on KSL!
- Exam: 60% of final grade
- Language
  - Q: English
  - A: English (preferred); German (possible)

Material

- It covers the material of the lectures (inc. guest lectures).
- Suggested complementary material:
  - Sommerville, Software Engineering (7th-9th edition)

    (Google: Software Engineering Sommerville filetype:pdf)
- It combines simple knowledge questions with questions requiring thinking.
- You can **NOT** bring: books, slides, personal notes, electronic devices

Topics

- Terminology
- Software design/quality (principles & diagrams)
- Software Engineering Processes
- Software architecture (styles & properties)
- Testing (methods & techniques)

Recommendation

- Answer questions at the end of each lecture slides
- Use the book to complement material presented during the lecture

Topics

- **Terminology**
- Software design/quality (principles & diagrams)
- Software Engineering Processes
- Software architecture (styles & properties)
- Testing (methods & techniques)

Exercise

Terminology

1. define: architectural style

2. define: principle of encapsulation

3. Class vs. Object

4. Fault tolerance vs. Fault avoidance

5. define: Req. Consistency; Completeness; correctness

Exercise

Terminology

1. define: architectural style

An architectural style defines a ***family of systems*** *in terms of a pattern of* ***structural organization****. More specifically, an architectural style defines a vocabulary of components and connector* ***types****, and a set of constraints on how they can be combined.*

Exercise

Terminology

1. define: architectural style
2. define: principle of encapsulation

Keep behaviour together with any related information

Exercise

Terminology

1. define: architectural style

2. define: principle of encapsulation

3. Class vs. Object

class: *a class is a template to create objects, it defines the state and methods of its instances, mainly exists at compile time*

object: *has state and operations, an object is an instance of a class, it exists at runtime*

Exercise

Terminology

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its components.
Fault avoidance seeks to prevent faults from being introduced into the software.

4. Fault tolerance vs. Fault avoidance

5. define: Req. Consistency; Completeness; correctness

Exercise

Terminology

Consistency: Are there any *requirements conflicts*?
Completness: Are *all functions* required by the customer included?
Correctness: Are the requirements correct?

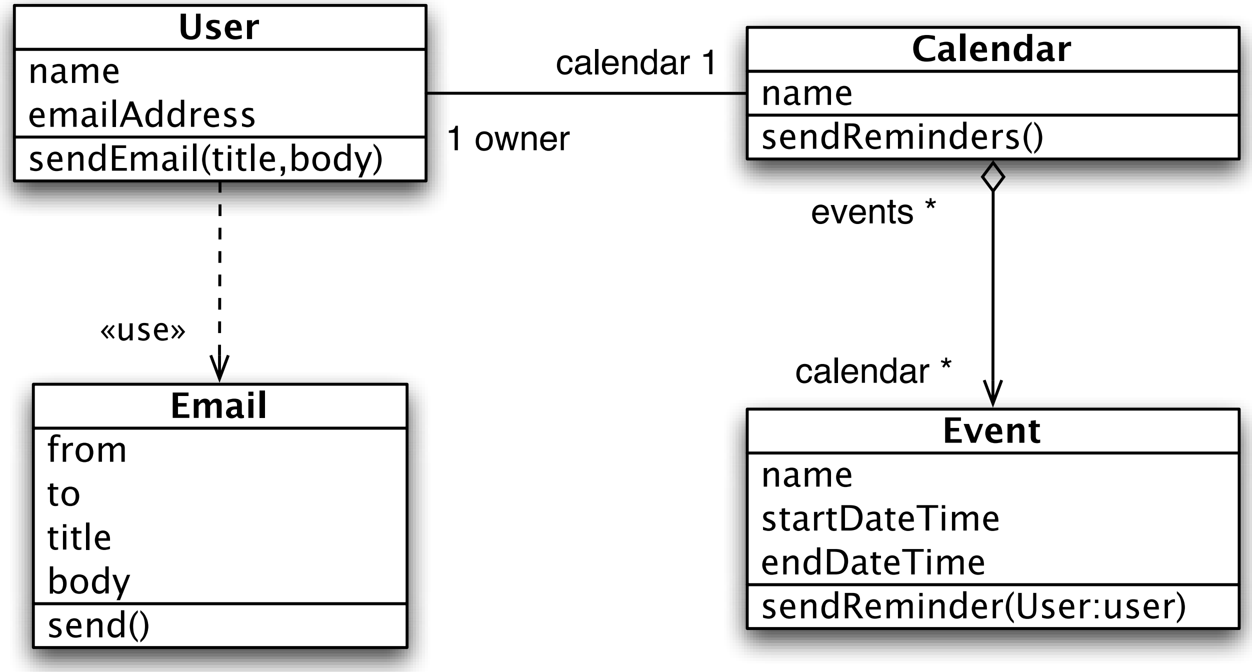
5. define: Req. Consistency; Completeness; correctness

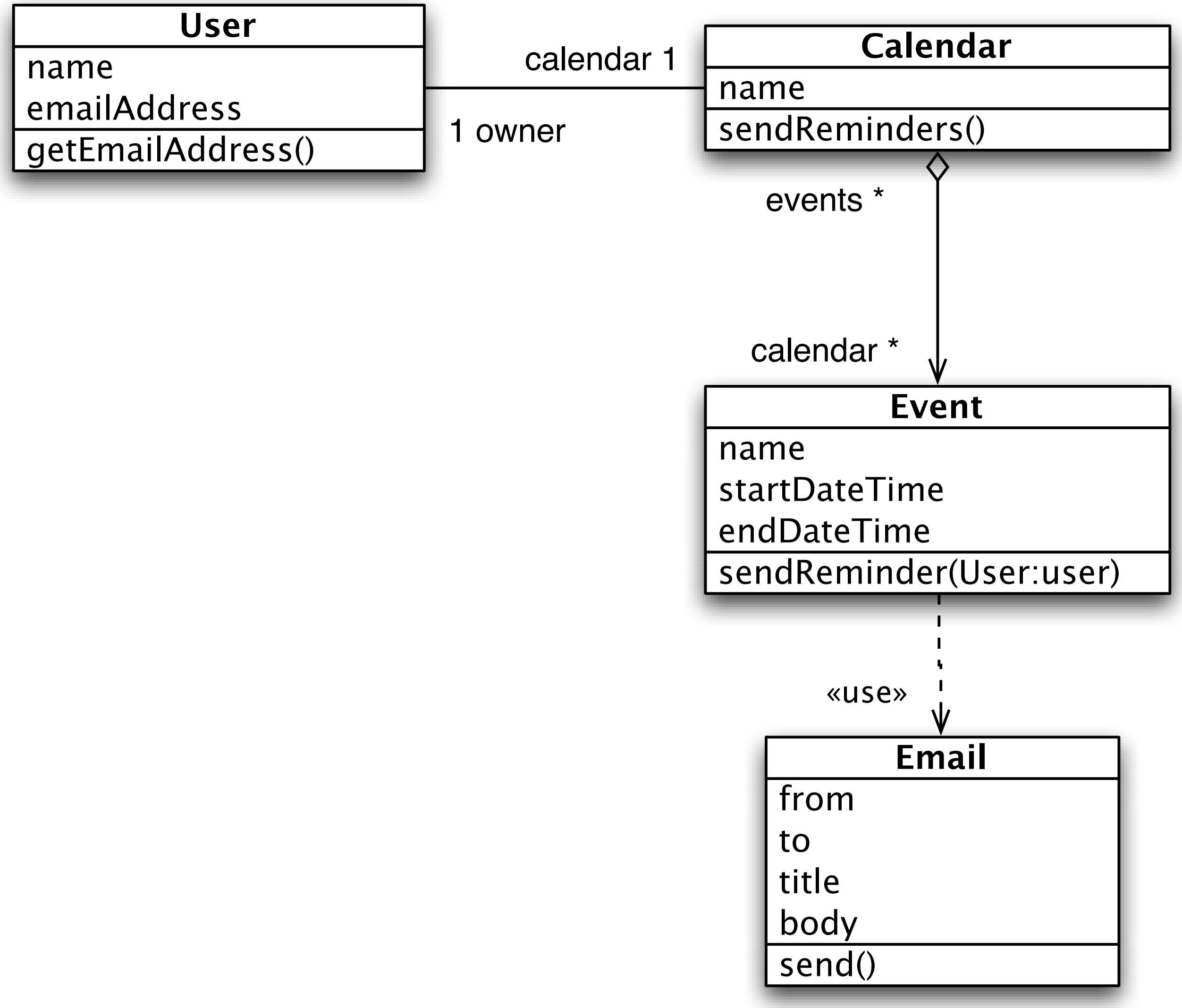
Topics

- Terminology
- **Software design/quality** (principles & diagrams)
- Software Engineering Processes
- Software architecture (styles & properties)
- Testing (methods & techniques)

Software design/quality

Your are writing a calendar application with a web framework. The system has model classes `User`, `Event`, `Calendar`, and `Email`. Users can receive a reminder a few minutes before an event starts. Below are two possible design with UML:

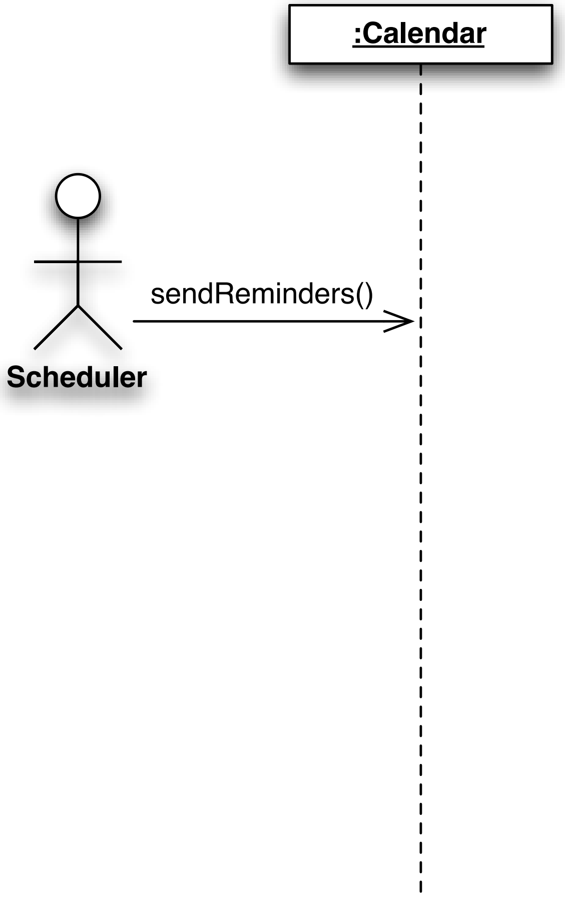


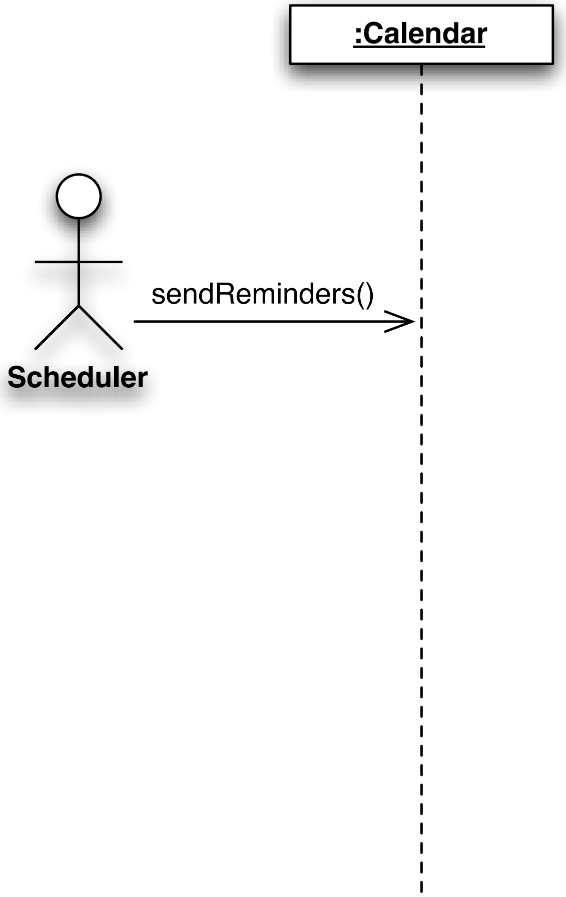


Exercise

Software design/quality

1. Complete the UML sequence diagrams below to show how a reminder is sent with each design. 2 points





SD fragments

- **alt**: Alternative multiple fragments; only the one whose condition is true will execute (Figure 4.4).

- **opt**: Optional; the fragment executes only if the supplied condition is true. Equivalent to an alt with only one trace

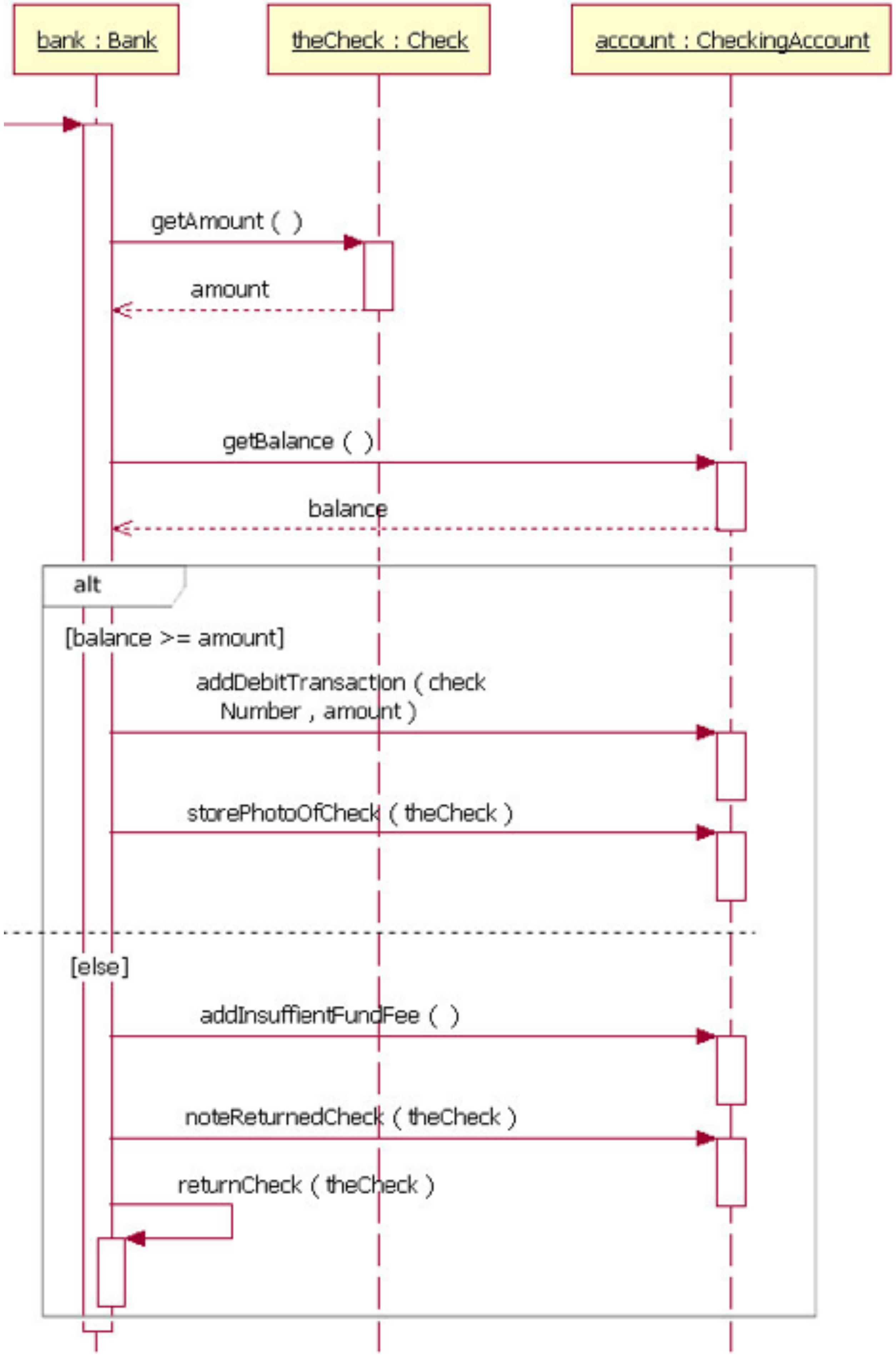
- **par**: Parallel; each fragment is run in parallel.

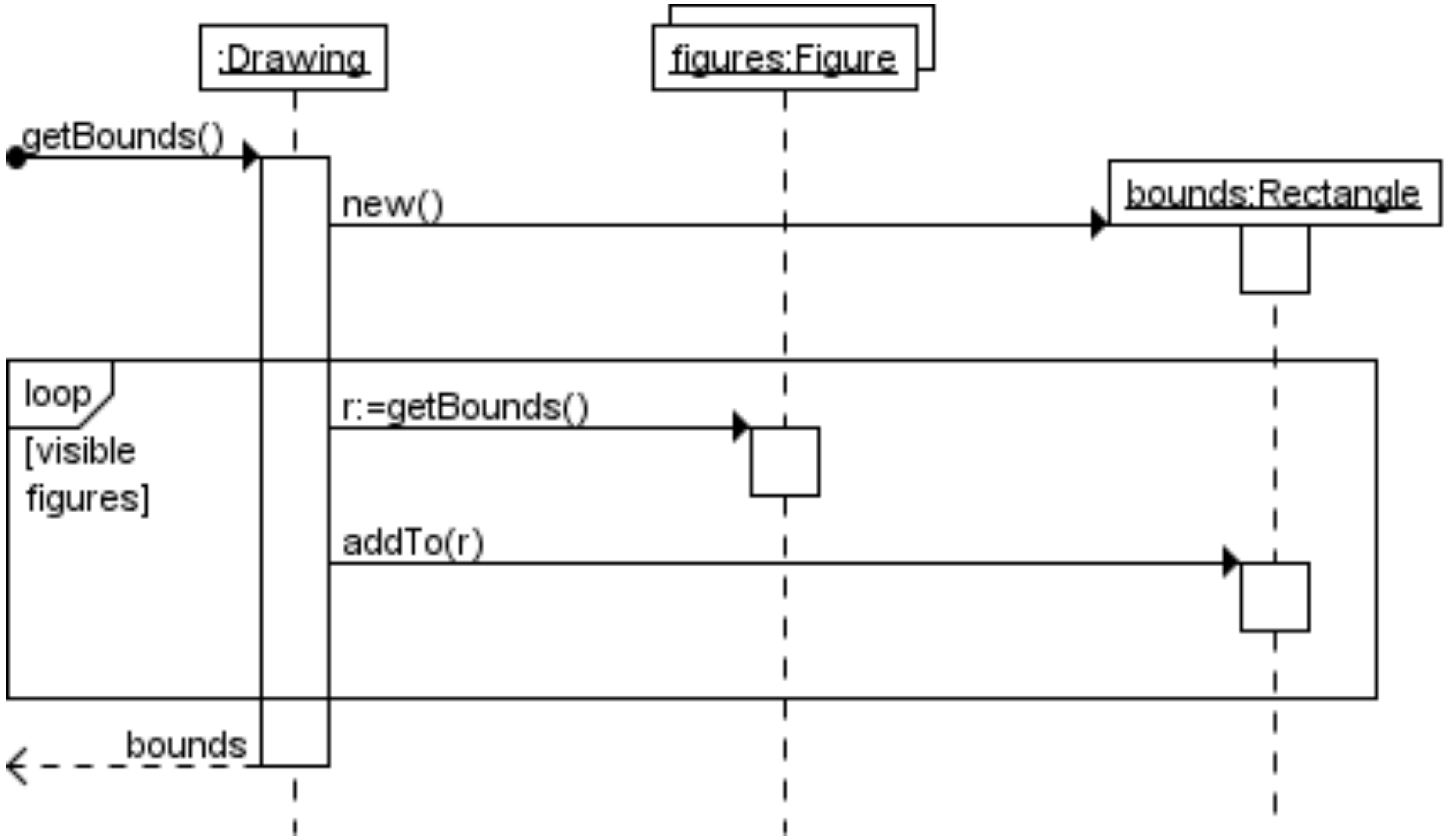
- **loop**: Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration

- **region**: Critical region; the fragment can have only one thread executing it at once.

- **neg**: Negative; the fragment shows an invalid interaction.

- **ref**: Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.


bank : Bank
theCheck : Check
account : CheckingAccount
getAmount ( )
amount
getBalance ( )
balance
alt
[balance >= amount]
addDebitTransaction (check Number , amount)
storePhotoOfCheck (theCheck)
[else]
addInsuffientFundFee ( )
noteReturnedCheck (theCheck)
returnCheck (theCheck)

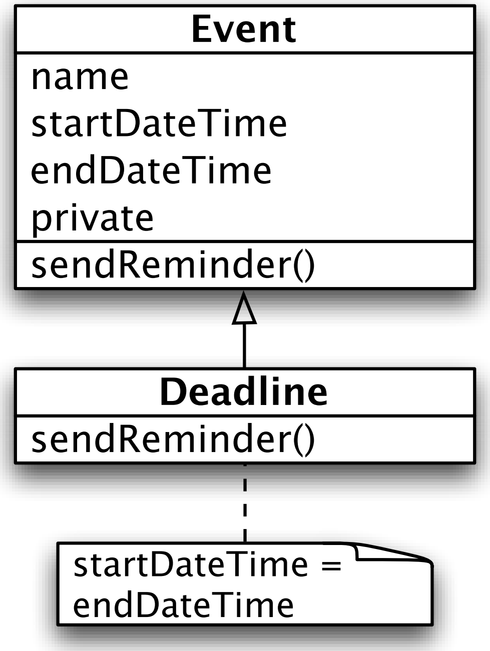


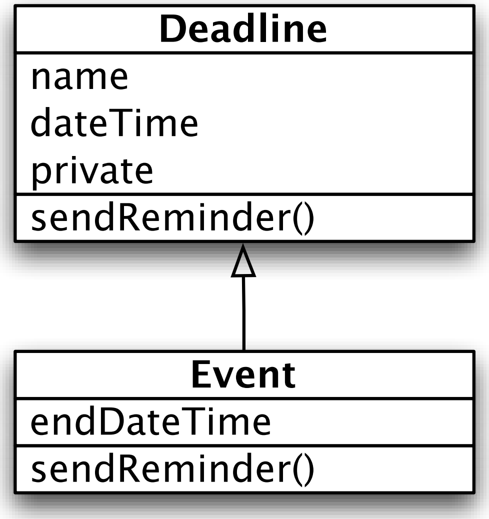
The iteration only applies to the visible figures of the drawing

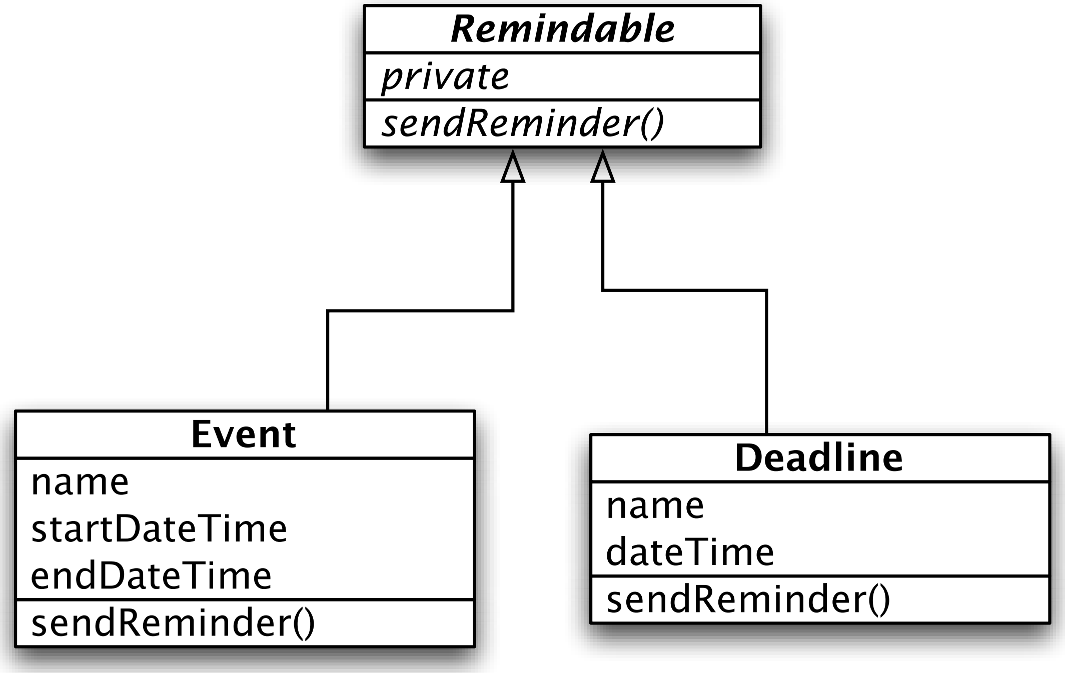
Exercise

Software design/quality

A new requirement is to support “deadlines”. A deadline is a point in time. Just as with events, a reminder can be sent a few minutes in advance a deadline is due. After the design meeting with your colleagues, you end up with three possible ways to model deadlines:

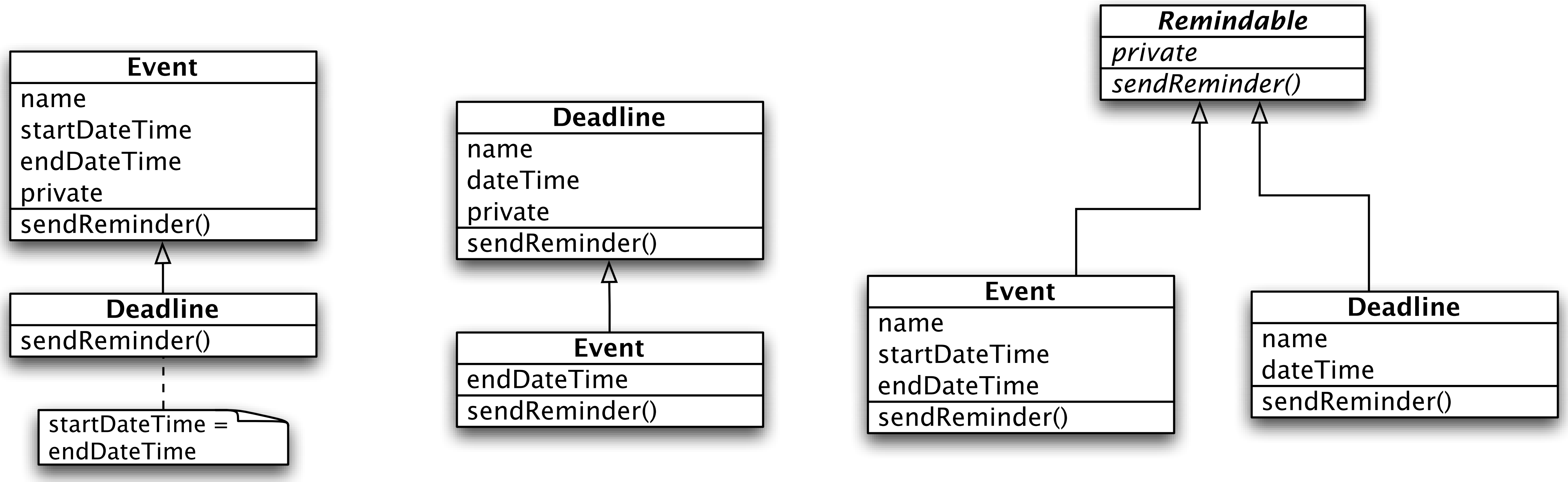






3. Which one would you pick? Why?

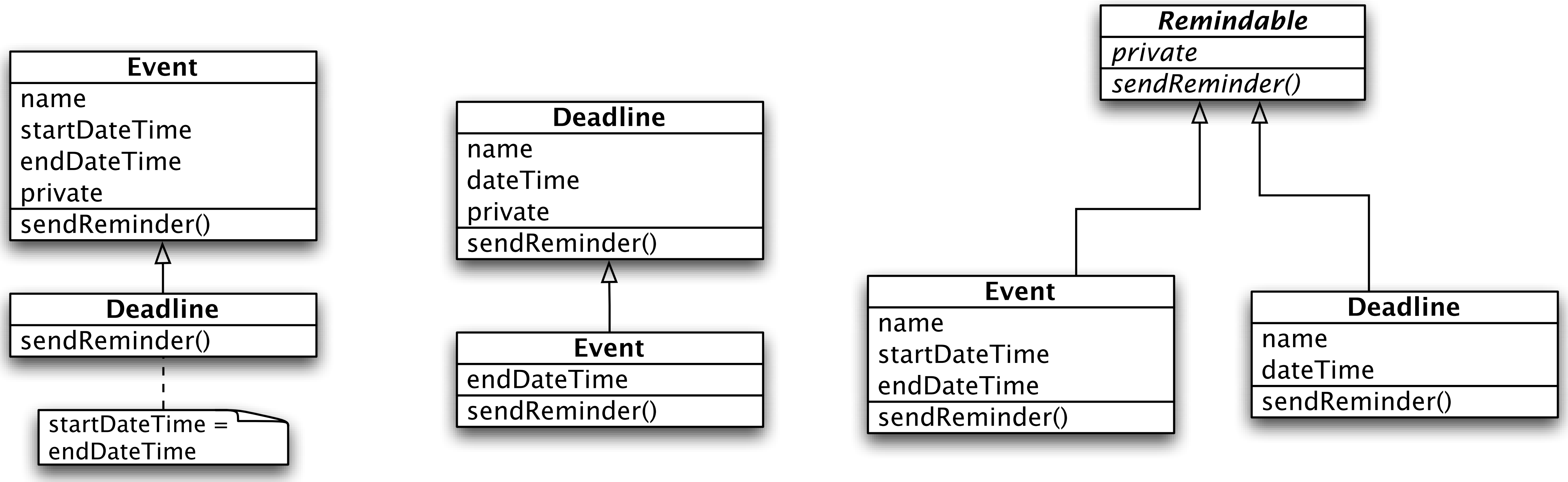
Hierarchies



Model a "kind-of" hierarchy:

> Subclasses should *support all inherited responsibilities*, and possibly more

Hierarchies



Event
name
startDateTime
endDateTime
private
sendReminder()

Deadline
sendReminder()

startDateTime =
endDateTime

Deadline
name
dateTime
private
sendReminder()

Event
endDateTime
sendReminder()

Remindable
private
sendReminder()

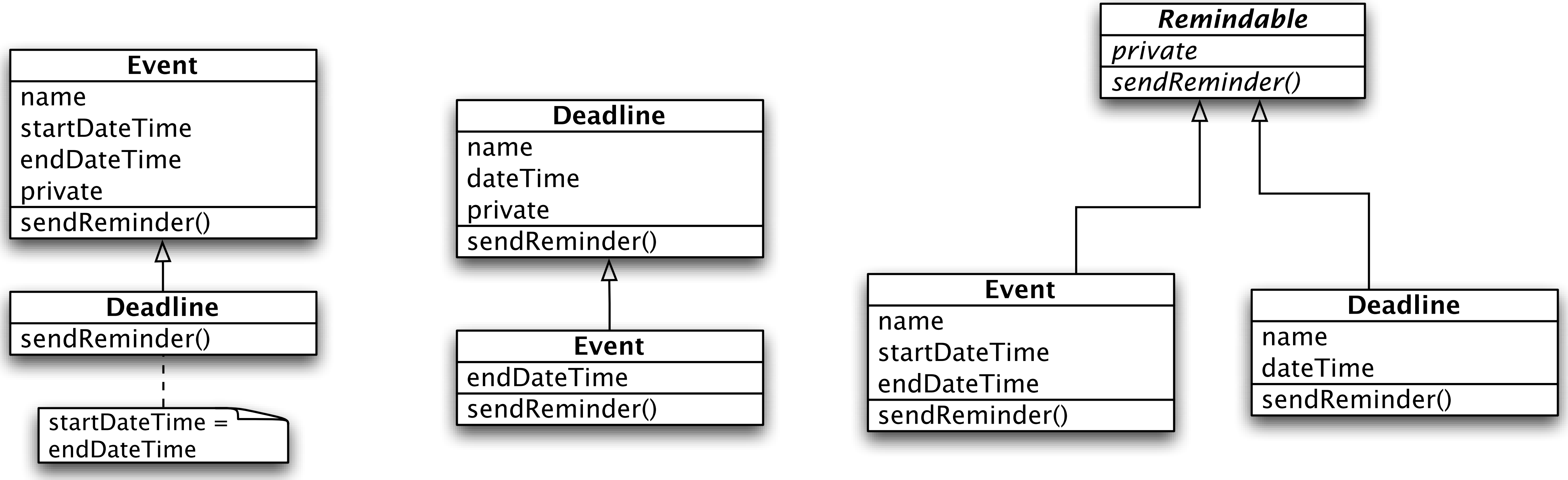
Event
name
startDateTime
endDateTime
sendReminder()

Deadline
name
dateTime
sendReminder()

Factor common responsibilities as high as possible:

- Classes that *share common responsibilities* should *inherit from a common abstract superclass*; introduce any that are missing

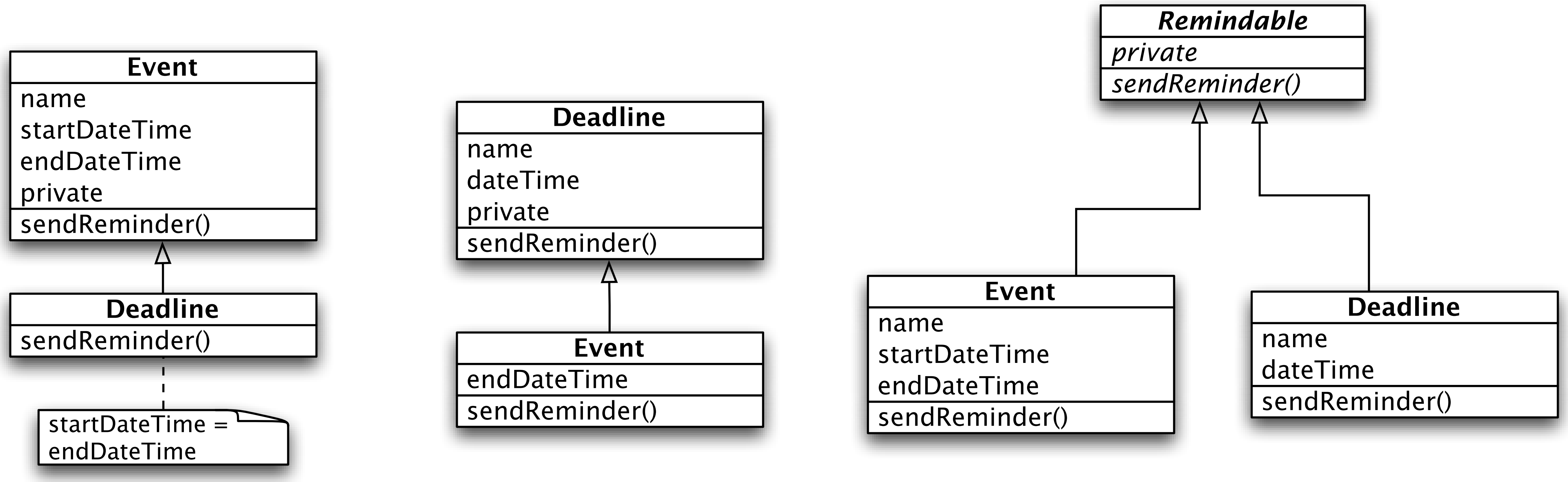
Hierarchies



Abstract classes do not inherit from concrete classes:

> Eliminate by introducing *common abstract superclass*: abstract classes should support responsibilities in an implementation-independent way

Hierarchies



Event
name
startDateTime
endDateTime
private
sendReminder()

Deadline
sendReminder()

startDateTime = endDateTime

Deadline
name
dateTime
private
sendReminder()

Event
endDateTime
sendReminder()

Remindable
private
sendReminder()

Event
name
startDateTime
endDateTime
sendReminder()

Deadline
name
dateTime
sendReminder()

Eliminate classes that do not add functionality:

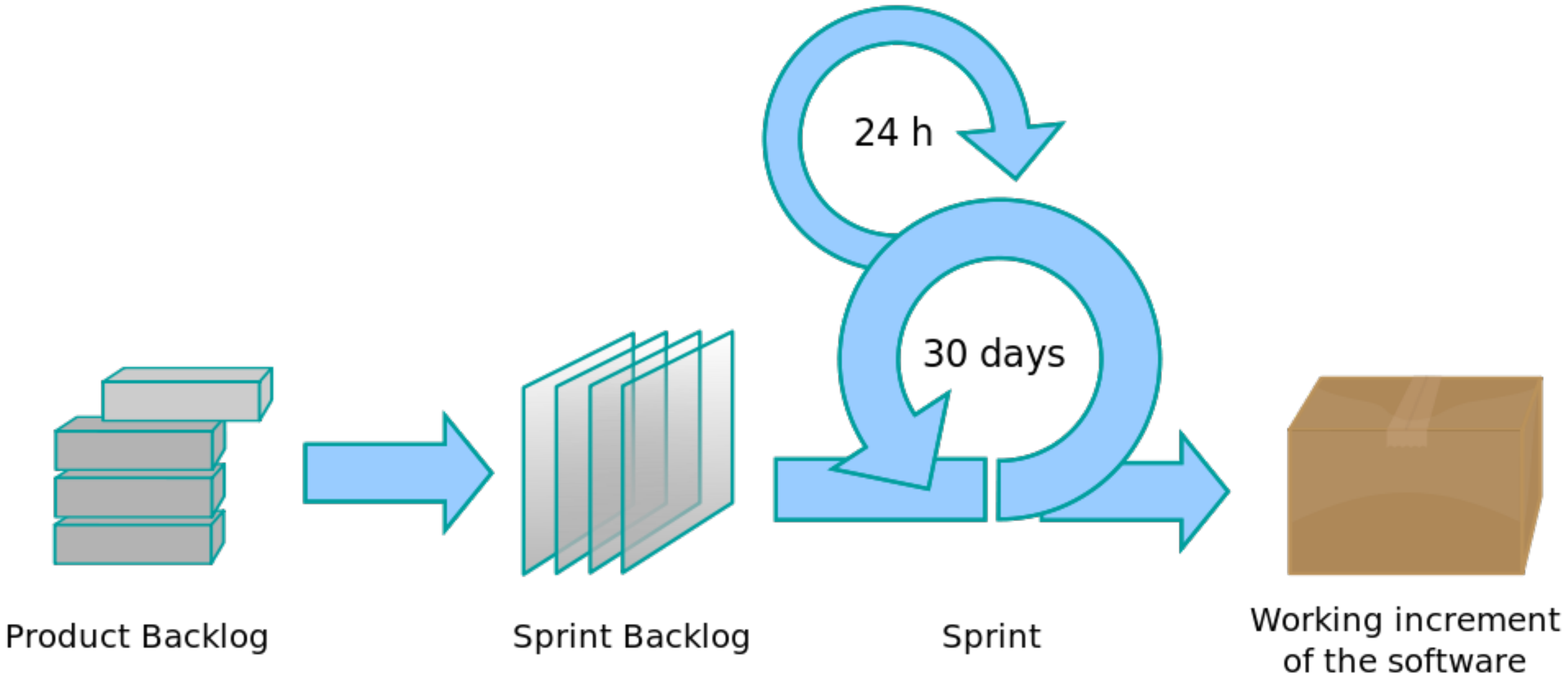
- Classes should either add new responsibilities, or a particular way of implementing inherited ones

Topics

- Terminology
- Software design/quality (principles & diagrams)
- **Software Engineering Processes**
- Software architecture (styles & properties)
- Testing (methods & techniques)

Exercise

Software Engineering Processes



Software Engineering Processes

Product backlog (PBL)
A prioritized list of high-level requirements.

Sprint backlog (SBL)
A prioritized list of tasks to be completed during the sprint.

Sprint
A time period (typically 1–4 weeks) in which development occurs on a set of backlog items that the team has committed to. Also commonly referred to as a Time-box or iteration.

Increment
The sum of all the Product Backlog items completed during a sprint and all previous sprints.

Exercise

Software Engineering Processes

- Product owner
- Scrum master
- Team

Software Engineering Processes

Product Owner

The person responsible for maintaining the Product Backlog by representing the interests of the stakeholders, and ensuring the value of the work the Development Team does.

Scrum Master

The person responsible for the Scrum process, making sure it is used correctly and maximizing its benefits.

Development Team

A cross-functional group of people responsible for delivering potentially shippable increments of Product at the end of every Sprint.

Topics

- Terminology
- Software design/quality (principles & diagrams)
- Software Engineering Processes
- **Software architecture** (styles & properties)
- Testing (methods & techniques)

Exercise

Software architecture

The **Design Structure Matrix** (DSM) is a simple, compact and visual representation of a system or project in the form of a matrix.

The off-diagonal cells are used to indicate relationships between the elements.
Reading across a row reveals what other elements the element in that row provides outputs to, and scanning a column reveals what other elements the element in that column receives inputs from.

Exercise

Software architecture

	A	**B**	**C**	**D**
A	–	15	0	0
B	0	–	18	0
C	0	0	–	13
D	0	0	0	–

- draw as package diagram
- which architectural style ?

	M	**V**	**C**
M	–	5	0
V	7	–	8
C	6	5	–

- is MVC correctly implemented ?

Topics

- Terminology
- Software design/quality (principles & diagrams)
- Software Engineering Processes
- Software architecture (styles & properties)
- **Testing** (methods & techniques)

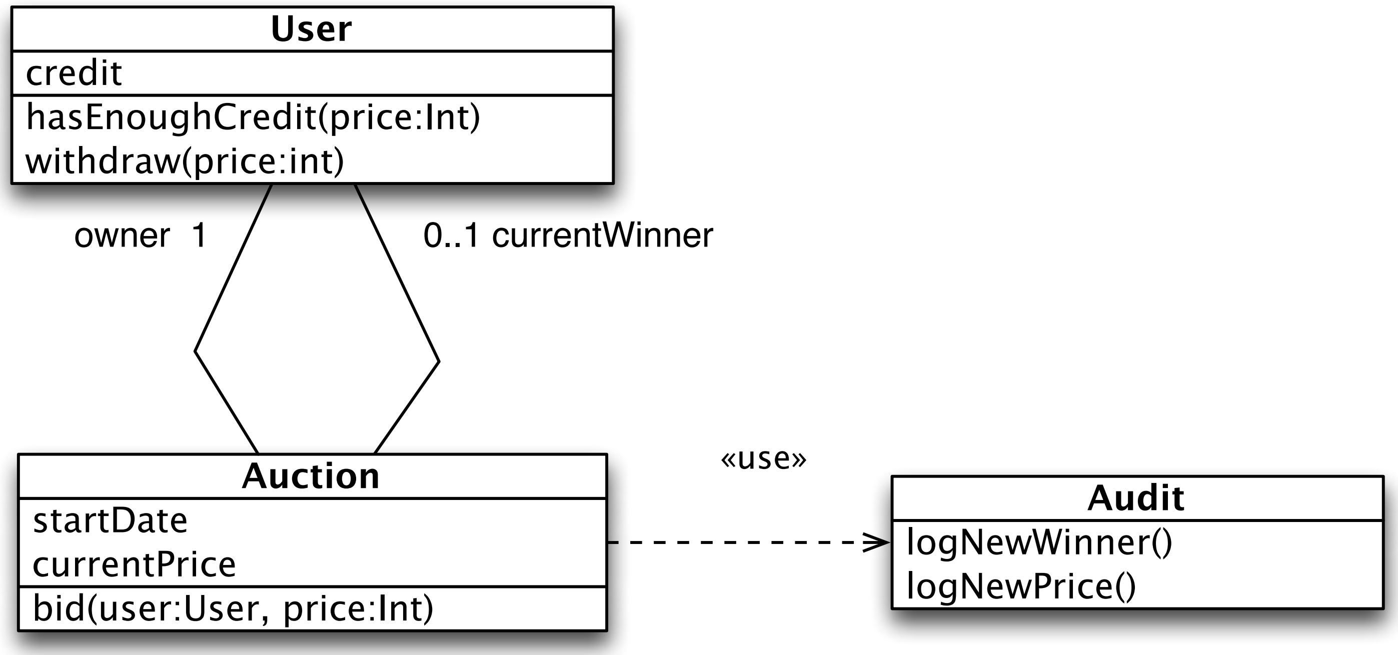
Testing

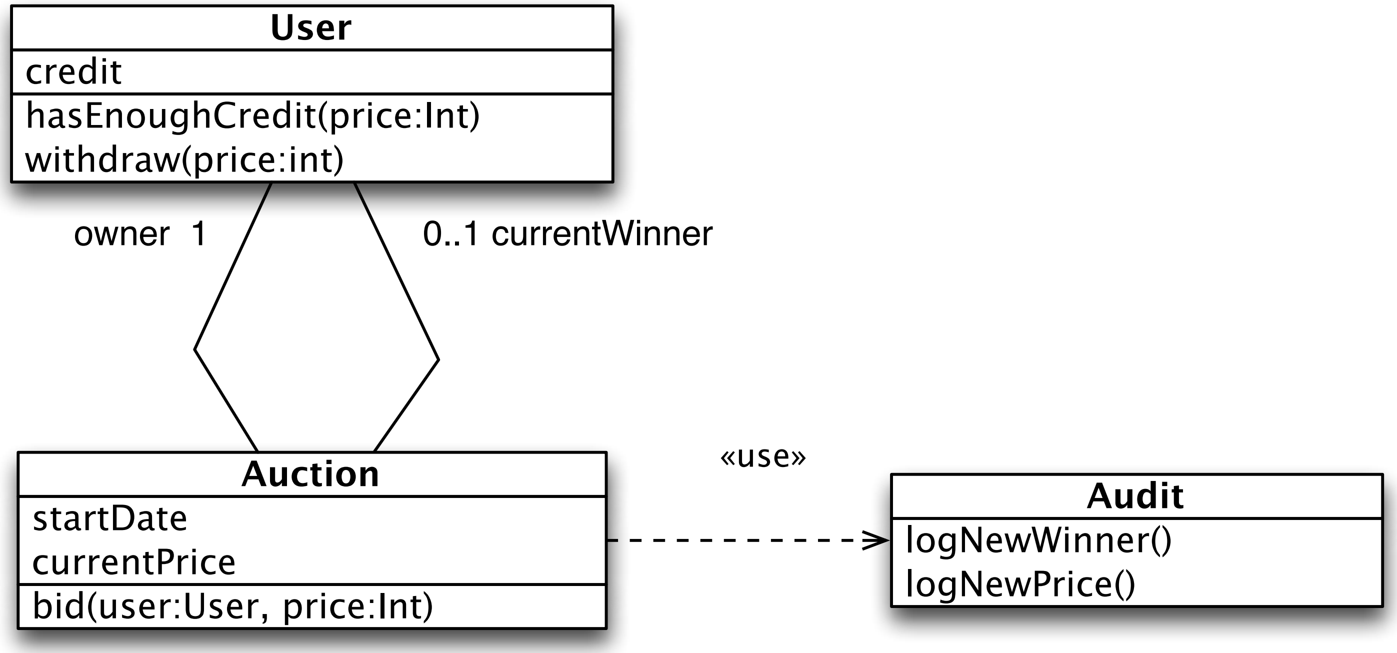
You are working as developer in a start-up and your main product is an online auction system. The auction mechanism is very simple:

1. An item for sale at the auction has an owner, a current price (the highest bid), a current winner (the highest bidder), and a start date. Auctions are automatically closed after one day.

2. The winner of an auction is the user who placed the highest bid before the auction was closed; a bid is valid only if the user has enough credit.

Your software is modeled with two entities `Auction` and `User`:





```
class Auction
	def initialize(User owner)
		@current_price = 0
		@start_date = Time.now
		@owner = owner
	end

	def bid( user, price )

		raise 'Bidder can not be the owner' if user == owner
		raise 'User had not enough credit' unless user.has_enough_credit(price)

		if( price > @current_price )
			@current_winner = user
			Audit.log_new_winner( user )

			@current_price = price
			Audit.log_new_price( price )
		end
	end

	def close()
		@current_winner.withdraw( @current_price ).
	end
end
```

Exercise

Testing

4. List the tests you need to achieve full branch coverage in `bid(user,price)` 2 points

   No need to write code, just textual explanation for each test

Branches

```
class Auction
    def initialize(User owner)
        @current_price = 0
        @start_date = Time.now
        @owner = owner
    end

    def bid( user, price )
                     1
        raise 'Bidder can not be the owner' if user == owner
        raise 'User had not enough credit' unless user.has_enough_credit(price)
                     2

        if( price > @current_price )
            @current_winner = user
          3 Audit.log_new_winner( user )

            @current_price = price
            Audit.log_new_price( price )
        end
    end

    def close()
        @current_winner.withdraw( @current_price ).
    end
end
```

Good Luck !