Chapter 4: Capturing the Requirements

1. Developers, customers, and users all bear some responsibility for the situation, as they all had input into the requirements of the system. However, the developers (or the development organization) should probably bear most of the responsibility because they, as software professionals, should have the expertise to ensure that the requirements are complete.

2. No system can be built to be safe and reliable in all environments and under all conditions. Therefore, we need to explicate any assumptions we make about the environment, about input values, and about the ordering of inputs. Such non-functional requirements must explicitly specify all of the situations and conditions under which the system must not fail. The context and the environment in which the system will be running become pre-conditions for stating and satisfying its safety and reliability requirements. For example, "the system must not fail catastrophically when the user supplies a non-valid numerical input." Ideally, the requirement should also specify what the system should do in such a situation (e.g. give a meaningful error message to the user and continue processing). The problem with specifying the requirement to "never fail" is that it does not explicitly state all of the situations in which the system must not fail, therefore it is not testable or demonstrable. Confidence that a system meets this requirement can be built up, however, by testing under as many conditions and in as many different situations as possible, and by soliciting input from users and other experts about the different situations that could be tested.

3.

a) The client daemon must be invisible to the user

Design constraint, referring to distributed solution

Functional requirement related to user interface

b) The system should provide automatic verification of corrupted links or outdated data

Functional requirement

c) An internal naming convention should ensure that records are unique

Functional requirement related to data constraints

d) Communication between the database and servers should be encrypted

Quality requirement referring to the security of the system

e) Relationships may exist between title groups [a type of record in the database]

Functional requirement referring to data relations

f) Files should be organizable into groups of file dependencies

Functional requirement related to data constraints

g) The system must interface with an Oracle database

Design constraint referring to the interface with other systems

h) The system must handle 50,000 users concurrently

Quality requirement referring to the performance of the system

Items d and f may be premature design decisions. They could be rewritten as:

d) Internal data stores and communications should be secure against illegal accesses

f) Dependencies among information should not be cyclic

In addition, items a and c refer to design constructs (e.g., clients, internal naming conventions). These requirements could be better expressed by eliminating reference to these constructs:

a) User should believe that he is interacting with a centralized system

c) All records must be unique

4. Decision table:

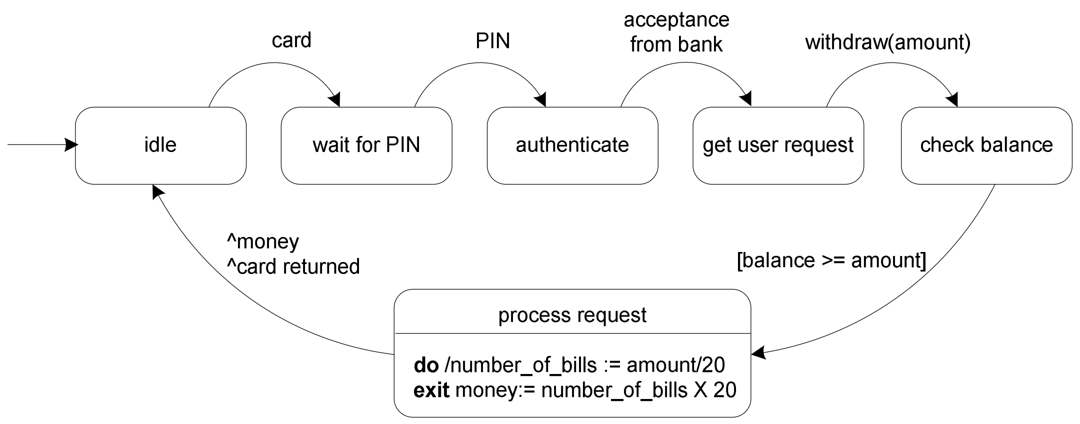
	R1	R3	R4	R5	R6	R8	R9	R10
black’s turn	T	T	-	-	F	F	-	-
black has a legal move available	T	F	-	-	-	-	-	-
red has a legal move available	-	-	-	-	T	F	-	-
black marker reaches red side of the board	-	-	T	-	-	-	-	-
red marker reaches red side of the board	-	-	-	-	-	-	T	-
no red markers on the board	F	F	F	T	F	F	F	F
no black markers on the board	F	F	F	F	F	F	F	T
black’s turn over		X						
red’s turn over						X		
red makes a move					X			
black makes a move	X							
black marker is king’ed			X					
red marker is king’ed							X	
game over; black wins				X				
game over; red wins								X

5. The specification is contradictory if there are two columns which specify the same condition (same set of truth values) but different actions. The specification is ambiguous or incomplete if there are some conditions for which there is no column (i.e. no specified action).

6.

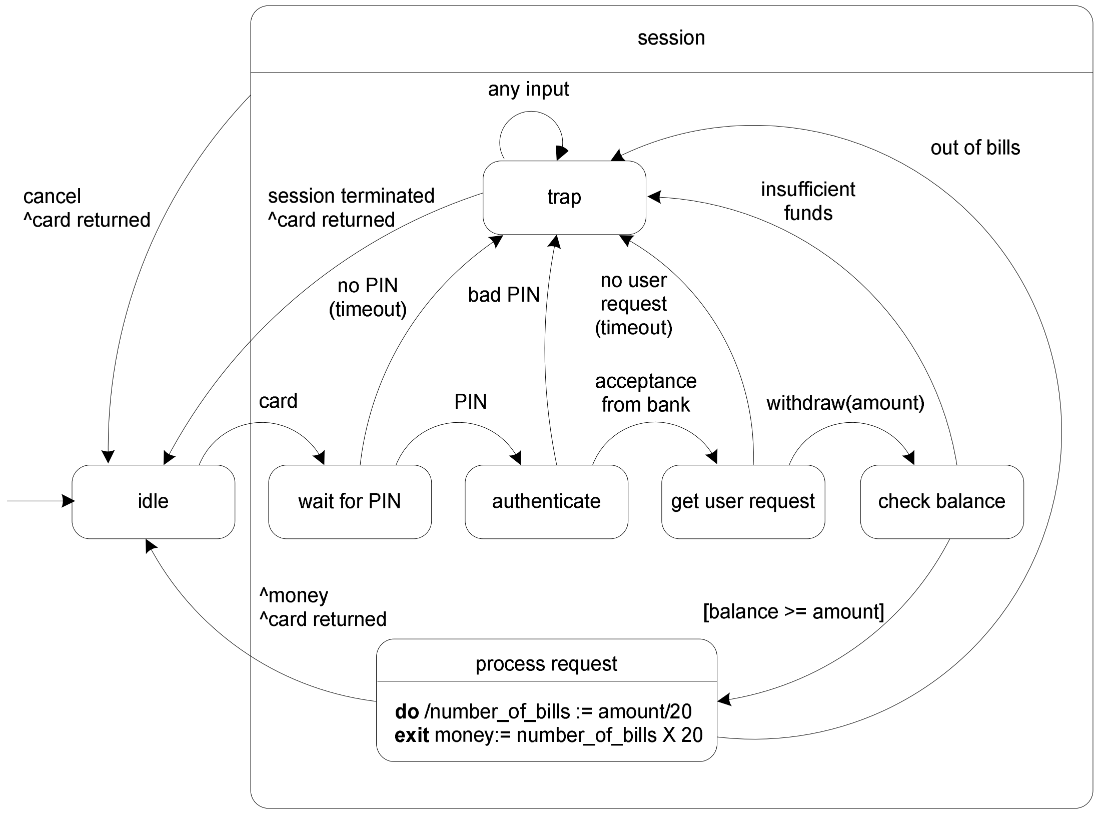
	$b^2 - 4ac < 0$	$b^2 - 4ac = 0$	$b^2 - 4ac > 0$
quadratic_real_roots (a,b,c) =	$\times$	$\frac{-b}{2a}$	$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

7. A state-machine specification to illustrate the requirements of an automatic banking machine (ABM).



Note: This state machine shows only the operation of withdrawing cash. Other operations can be modeled in a similar way.

8. A state-machine for the ABM including a trap state.



9. Safety properties:

- The amount of money returned by the ABM is never more than the amount requested by the client in the withdrawal.

  $\square$ (money $\Rightarrow$ (money $\leq$ amount))

- The amount of money returned by the ABM is never more than the client's balance.

  $\square$ (money $\Rightarrow$ (money $\leq$ balance))

Liveness properties:

- When an authenticated client specifies the amount of money to withdraw, and the client has sufficient funds in his account, and the ABM has sufficient funds to dispense, the ABM dispenses the requested amount of money

  $\square$ ((process req $\wedge$ amount $\leq$ balance $\wedge$ amount $\leq$ ABM funds) $\Rightarrow$ O(money = amount))

- When the client terminates the session pressing the cancel button, the ABM returns his card

  $\square$ ((session $\wedge$ cancel) $\Rightarrow$ O card returned)

10.

Proof 1 : The amount of money returned by the ABM is never more than the amount requested by the client in the withdrawal.

<u>PROOF</u>	<u>RATIONAL</u>
a. money	*given in formula*
b. ABM dispenses $20 bills	*assumption*
c. number_of_bills = amount/20	*action in state process request, a*
d. ABM has sufficient cash	*assumption*
e. money = number_of_bills x 20	*action in state process request, c*
f. money = (amount/20) x 20	*substitution b in d*
g. money <= amount	*number_of_bills is integer part of amount/20*

Assumptions about the environment

ABM dispenses $20 bills

ABM has sufficient cash

Proof 2: The amount of money returned by the ABM is never more than the client's balance:

PROOF	RATIONAL
a. money	*given in formula*
b. ABM dispenses $20 bills	*assumption*
c. number_of_bills = amount/20	*action in state process request, a*
d. money <= number_of_bills x 20	*action in state process request*
e. money <= (amount/20) x 20	*substitution of b in c*
f. amount <= balance	*guard on entering state process request*
g. money <= balance	*transitivity of <=*

Assumptions about the environment

ABM dispenses $20 bills

Proof 3:

PROOF	RATIONAL
a. in state process request	*given in formula*
b. number_of_bills = amount/20	*action in state process request*
c. amount = (amount/20)x20	*assumption*
d. ABM has sufficient cash	*given in formula*
e. money = number_of_bills x 20	*action in state process request, c*
f. money = amount	*substitution of a, b into d*
g. O (money=amount)	*no guard on transition leaving state process request*

Assumptions about the environment

Amount request is divisible by 20

Proof 4:

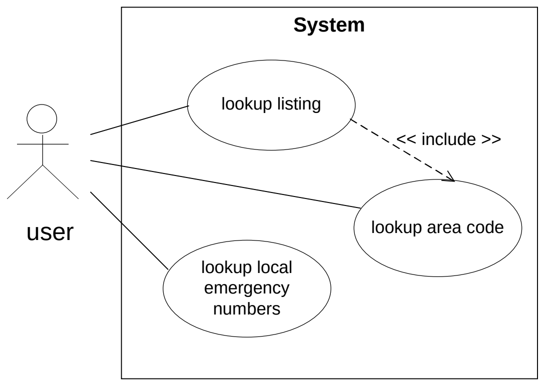
PROOF	RATIONAL
a. session	*given in formula*
b. cancel $\Rightarrow$ O(card returned)	*guard and action on transition leaving*
c. amount = (amount/20)x20	*state session*

11. Some factors to consider in this decision are

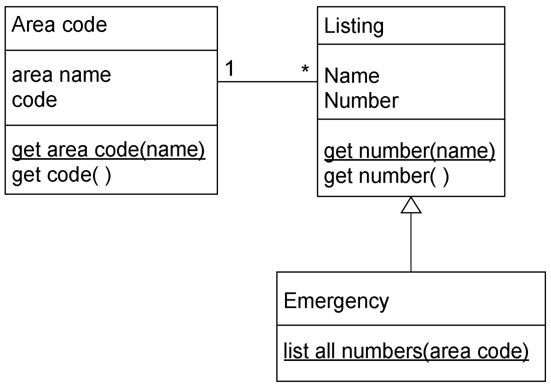
- Who will be building the prototype? If they have a good understanding of the requirements, then they could write the requirements after the prototype has been evaluated. Otherwise, they will need some guidance or documentation, and a draft of the requirements should be written by someone else so that the prototype builders can use them.

- How tight are the schedule and budget? If the requirements are written before the prototype is developed, they will likely have to be substantially modified after the prototype is evaluated.
- How much is known about the requirements before the prototype is evaluated? If little is understood about the requirements, then it does not make sense to try to write them down. They will be clarified through the evaluation of the prototype(s) with the customer.

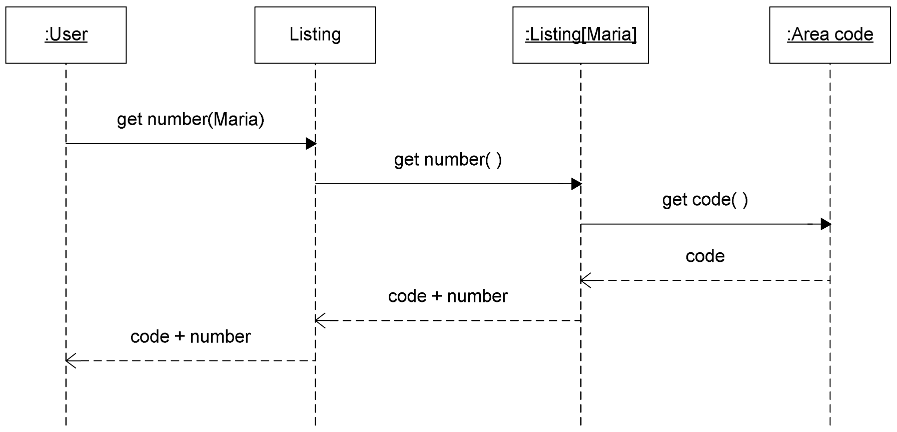
12. UML use-case diagram for an on-line telephone directory.



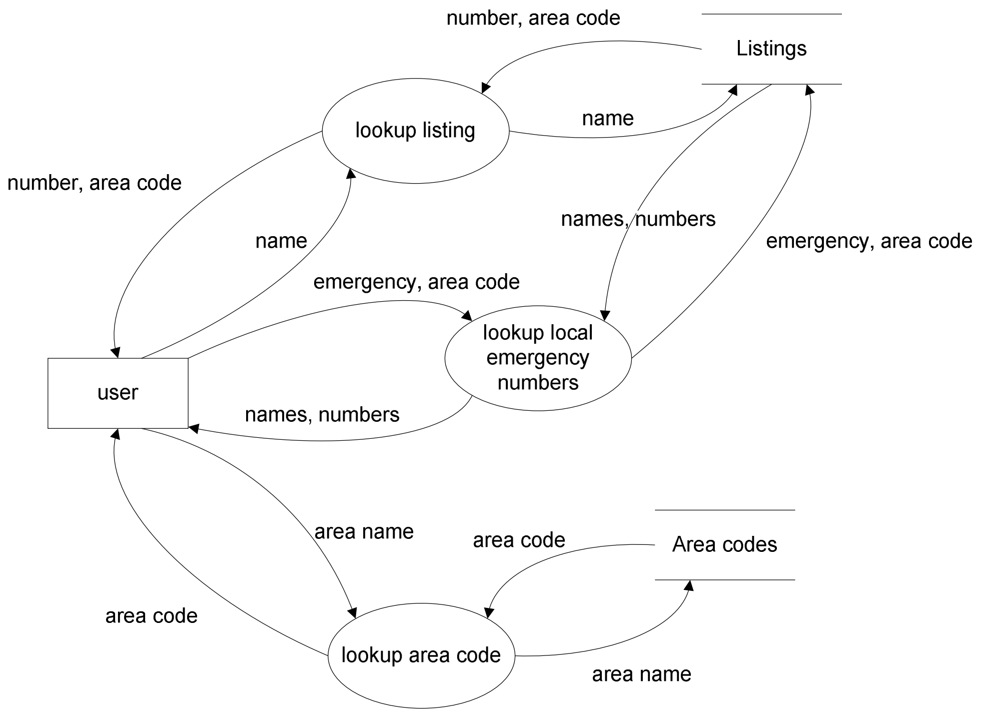
UML class diagram. Note that this is one way the objects might be organized.



UML sequence diagram showing the realization of the 'lookup listing' use case.



13. Data-flow diagram to illustrate the functions and data flow for the on-line telephone directory system.



14. Although the two are related, separating functional flow and data flow often simplifies the specification of the system by dealing with just one source of complexity at a time. Also, the two types of flow often correspond to different development tasks (e.g. writing computational subroutines vs. designing the interfaces between modules).

15. The major difficulty with specifying real-time systems is specifying feasible timing constraints for computations and communications. The order of inputs to a real system is often nondeterministic, and so timed responses for every input combination must be considered in the specification.

16. Most users think of requirements in terms of features, which describe new functionality. Thus, functional specifications are often easier to read and write. The primary advantage of object-oriented specifications is that they are supposed to be easier to change. OO specifications organize data and operations on data into separate classes, such that most requirements and design changes can be realized by local additions or changes to the model, new methods, or new sequences of method calls.

17.

```
┌─ Schedule ──────────────────────────────────────────────
│ presentation: Presenter ⇸ Date
├─────────────────────────────────────────────────────────
│ ∀ d: Date | # { p: Presenter | presentation(p) = d • p } ≤ 4
└─────────────────────────────────────────────────────────

┌─ AddPresentation ───────────────────────────────────
│ Δ Schedule
│ p?: Presenter
│ d?: Date
├─────────────────────────────────────────────────────
│ ∀ d: Date | # { p  | presentation(p) = d • p } < 4
│ presentation′ = presentation ⊕ { p? ↦ d? }
└─────────────────────────────────────────────────────
```

```
┌─ RemovePresentation ──────────────────
│ Δ Schedule
│ p?: Presenter
├───────────────────────────────────────
│ presentation′ = { p? } ⩤ presentation
└───────────────────────────────────────
```

```
┌─ SwapDates ──────────────────────────────────────────────────────────────
│ Δ Schedule
│ p1?, p2?: Presenter
├──────────────────────────────────────────────────────────────────────────
│
│ presentation′ = presentation
│    ⊕ { p1? ↦ presentation(p2?) }  ⊕ { p2? ↦ presentation(p1?) }
└──────────────────────────────────────────────────────────────────────────
```

```
┌─ ListPresentations ─────────────────────────────
│ Ξ Schedule
│ d?: Date
│ l!: ℙ Presenter
├─────────────────────────────────────────────────
│ l! = { p: Presenter | presentation(p) = d? • p }
└─────────────────────────────────────────────────
```

```
┌─ ListDate ────────────────────────
│ Ξ Schedule
│ p?: Presenter
│ d!: Date
├───────────────────────────────────
│ d! = presentation(p?)
└───────────────────────────────────
```

```
┌─ SendReminders ──────────────────────────────────────────────────────
│ Ξ Schedule
│ today?: Date
│ peopleToRemind!: ℙ Presenter
├──────────────────────────────────────────────────────────────────────
│  peopleToRemind! = { p: Presenter | presentation(p) = today? • p }
└──────────────────────────────────────────────────────────────────────
```

18.

```
isOnLoan(New, i) ≡ false;
isOnLoan(borrow(lib, i), i2) ≡ if (i=i2) then true;
                               else isOnLoan(lib, i2);

isOnLoan(buy(lib, i), i2) ≡ if (i=i2) then false;
                            else isOnLoan(lib, i2);
isOnLoan(reserve(lib, i), i2) ≡ isOnLoan(lib, i2);
```

```
unreserve (New, i) ≡ New;
unreserve(buy(lib, i), i2) ≡ if (i=i2) then buy(lib, i);
                             else buy(unreserve(lib, i2), i);
unreserve(borrow(lib, i), i2) ≡ borrow(unreserve(lib, i2), i);
unreserve(reserve(lib, i), i2) ≡ if (i=i2) then unreserve(lib,i2);
                                 else reserve(unreserve(lib, i2), i);

isOnReserve(New, i) ≡ false;
isOnReserve(reserve(lib, i), i2) ≡ if (i=i2) then true;
                                   else isOnReserve(lib, i2);
isOnReserve(buy(lib, i), i2) ≡ if (i=i2) then false;
                               else isOnReserve(lib, i2);

isOnReserve(borrow(lib, i), i2) ≡ isOnReserve(lib, i2);
```

Note: The unreserve operation is supposed to cancel all previous reservations. Thus, even if unreserve finds a matching reservation, it continues to search the rest of the sequence of library operations, looking for more matching reservations.

19. In most cases, an application domain-specific checklist is preferable. However, some things that would be included in any checklist are:

- For each function described, are all conditions under which the function can be invoked specified?
- For each function described, are all conditions under which the function can terminate specified?
- For each user input, are appropriate responses specified for invalid input?
- For each output, is the format specified in all situations?
- Are there any ambiguous statements (e.g. the use of “can” or “should”)?
- Are there any “to be specified” features?
- Are all statements understandable?
- Is each requirement specific enough to be testable?

20. Yes, they could be merged if the customer is familiar with software development and is to be highly involved in the entire software development process. In this case, merging the documents reduces effort and the amount of documentation to keep track of. Otherwise, the benefit of having two documents is that it provides documentation for communicating with the customer as well as documentation from which to develop a system design. Separating the two helps keep them clear and understandable to their two separate audiences.

21. Specifications that use formal methods could be compared with specifications developed using other methods meant to ensure thoroughness, such as the use of requirements reviews.

22. Some factors to consider:

- If a customer consistently ends up with systems with less functionality than specified, how will that affect his or her view of the software provider? of the software industry?
- If a customer has unrealistic ideas about what software can deliver, is it up to the software provider to educate the customer? Can the customer be expected to know what is realistic? Is the customer expecting to be educated in this way?
- Say that you decide to be honest and tell the customer that you cannot implement the requirement. What if the customer cancels the contract after finding another software vendor that claims the system can be implemented?