Software Specifications

Refining and representing requirements

(Book chapter 28)

Software specifications

- Implementation needs much more details than the very abstract requirements
- Many of these details are constraints, prescriptions, or insights given by the stakeholders
- Microwave oven:
  - Requirement : “Cook a plate for a certain time given in input by the user”.
  - Specifications: how to use the keypad and operate the oven

Software specifications

- **Definition:** *"Specifications represent a model of how inputs are related to system reactions and outputs"*
- Specification is a **representation** process.
- Requirements are represented in a manner that ultimately leads to a smooth implementation
- Specifications will increase the level of details given in the requirements
- It will answer much more questions, thus furthering the analysis before solution writing
- Needed for : complex, large, or critical problems.

Specification Principles

- Separate functionality from implementation
- Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli from the environment
- Establish the context in which the system operates by specifying the manner in which other system components interact with the system
- Create a cognitive model rather than a design or implementation model. A cognitive model describes a system as perceived by its user community.
- A specification is always an abstraction of a complex situation. Specification will be incomplete and might require several levels of abstraction.
- Organize the content and structure of a specification in a way that will enable easy reference and changes

Some specification techniques

- Informal
  - Natural language
  - Pseudo-code
- Semi-formal
  - Entity-relationship diagrams
  - Dataflow diagrams
  - OO analysis
- Formal
  - State transition diagrams and tables
  - Formal specification languages (e.g. Z, VDM)

Plain English

- In many development projects, the specification document consists of page after page of English, or some other natural language like German, French, etc.
- ***Problems***
  - Not precise
  - Can be confusing
  - Cannot be checked for completeness/consistency.
- e.g.: Add A to B unless A is less than B in which case subtract A from B

Pseudo-code

- A “quasi” programming language, consisting of
  - Imperative sentences with a single verb and a single object;
  - A limited set of “action-oriented” verbs from which the sentences must be constructed;
  - Decisions represented with a formal IF-ELSE-ENIF structure;
  - Iterative activities represented with DO-WHILE or FOR-NEXT structures.
- An attempt to combine the informality of natural languages with the strict syntax and control structures of a programming language.

Pseudo-code

- Understandable by a non-programming person.
- Reduce the ambiguity of requirements.
- Problems
  - Not good for a high level abstraction of the project.
  - The writer does need programming skills.
  - The requirement engineer is prone to fall into programming details.

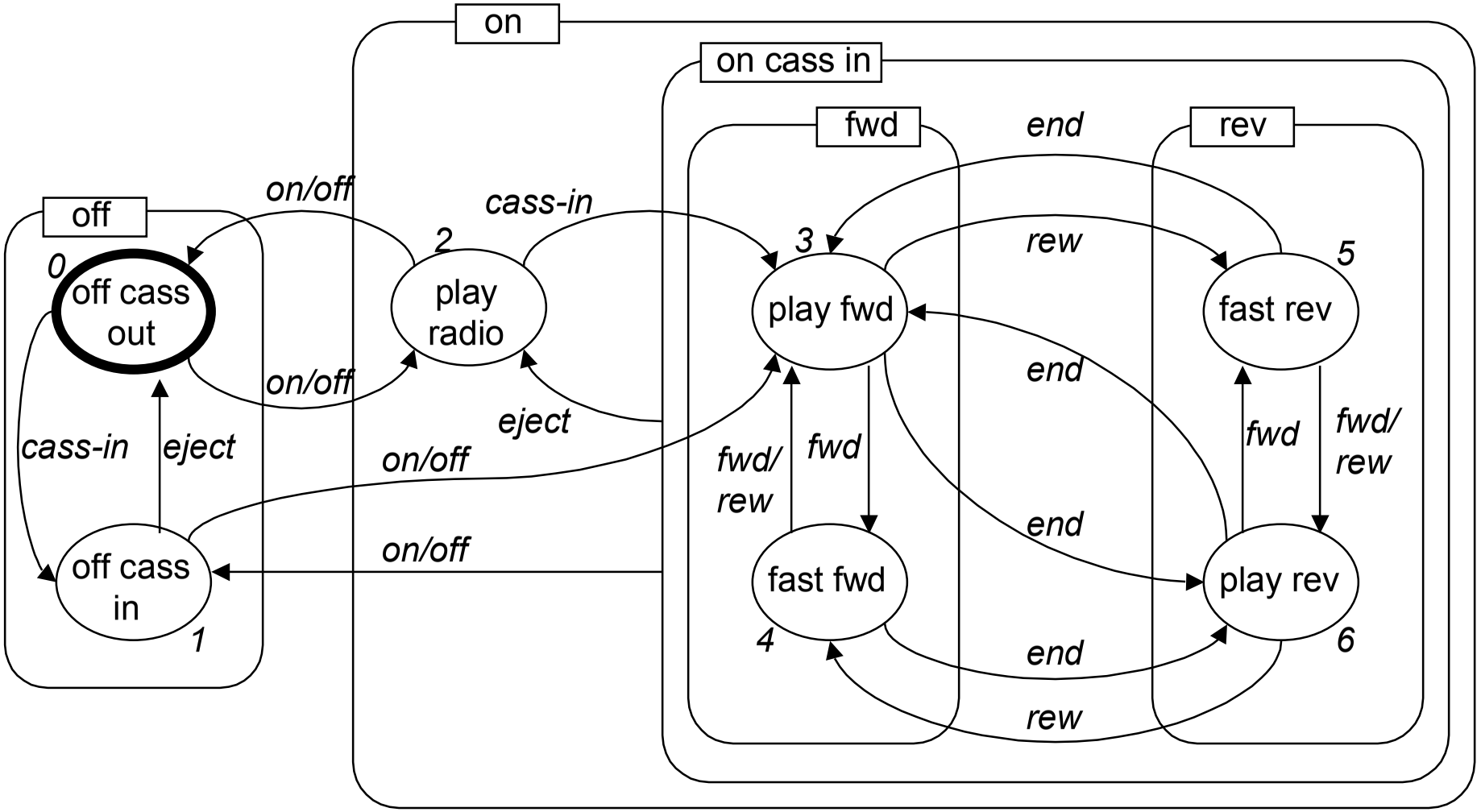
Finite state machines

- Regard the system as a “hypothetical machine”, whose output and next state can be determined by the current state and the even that caused the transition.
- Two popular notations for finite state machines:
  - State-transition diagram
  - State-transition matrix
- Suitable for describing the interaction between the user and the system.
- Not good for presenting a system’s behavior with several inputs.

Finite state machines

- *The radio cassette player is turned on or off using a button. Turning it off disables all but the on/off button. When a cassette is put in the player, it starts playing it automatically. While a cassette is being played, one can use the fast forward or rewind buttons to rapidly advance/rewind the cassette tape. If the fast forward or rewind button is pushed while the cassette is advanced/rewinded, the player starts playing the cassette. When the cassette reaches the end of one side, the mechanism is automatically reversed and the other side starts playing. A button can be pressed anytime to eject the previously inserted cassette in the player. When the player is on with no cassette in, it plays the radio.*

Finite state machines



Finite state machines

	on/off	eject	fwd	rew	cassin	end
0	2	N/A	N/A	N/A	1	N/A
1	3	0	N/A	N/A	N/A	N/A
2	0	N/A	N/A	N/A	3	N/A
3	1	2	4	5	N/A	6
4	1	2	3	3	N/A	6
5	1	2	6	6	N/A	3
6	1	2	5	4	N/A	3

Decision trees/tables

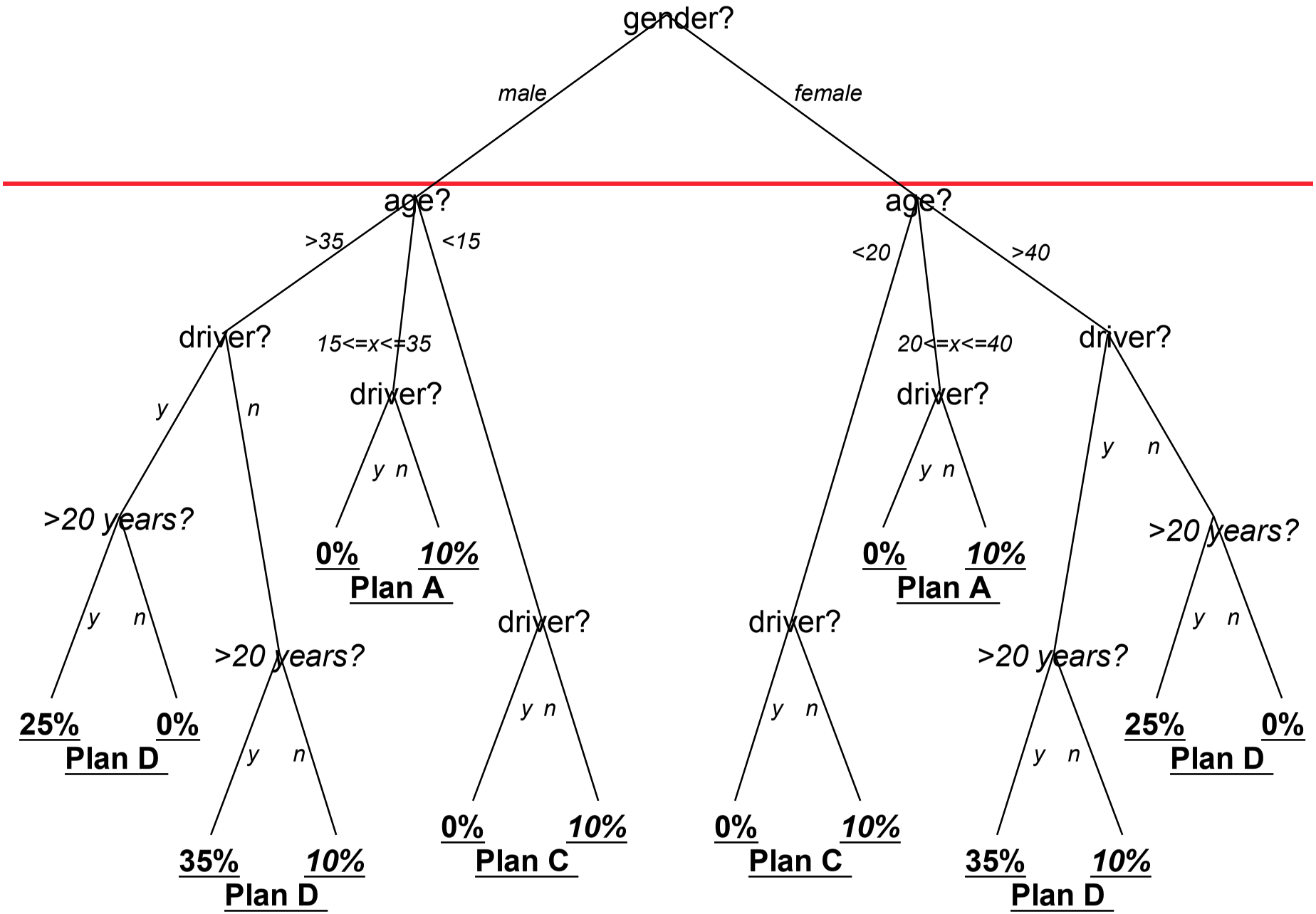
- Tree-like diagrams or tables used to describe complex logic relationships within a requirement.
- Suitable for a requirement that deals with a combination of inputs; and different combinations of the inputs lead to different outputs.

Decision trees/tables

- Develop a decision table for the following description of the insurance plan policy at the Everlasting life insurance company:
- *Males between 15 and 35 years old are applied insurance plan A. Females between 20 and 40 years old are applied insurance plan B. All males under 15 and females under 20 are applied insurance plan C. All males over 35 and all females over 40 years old are applied insurance plan D. Clients without a driver's license are applied a 10% rebate. Clients that have been insured for more than 20 years and currently part of insurance plan D are applied a 25% rebate.*

Decision trees/tables

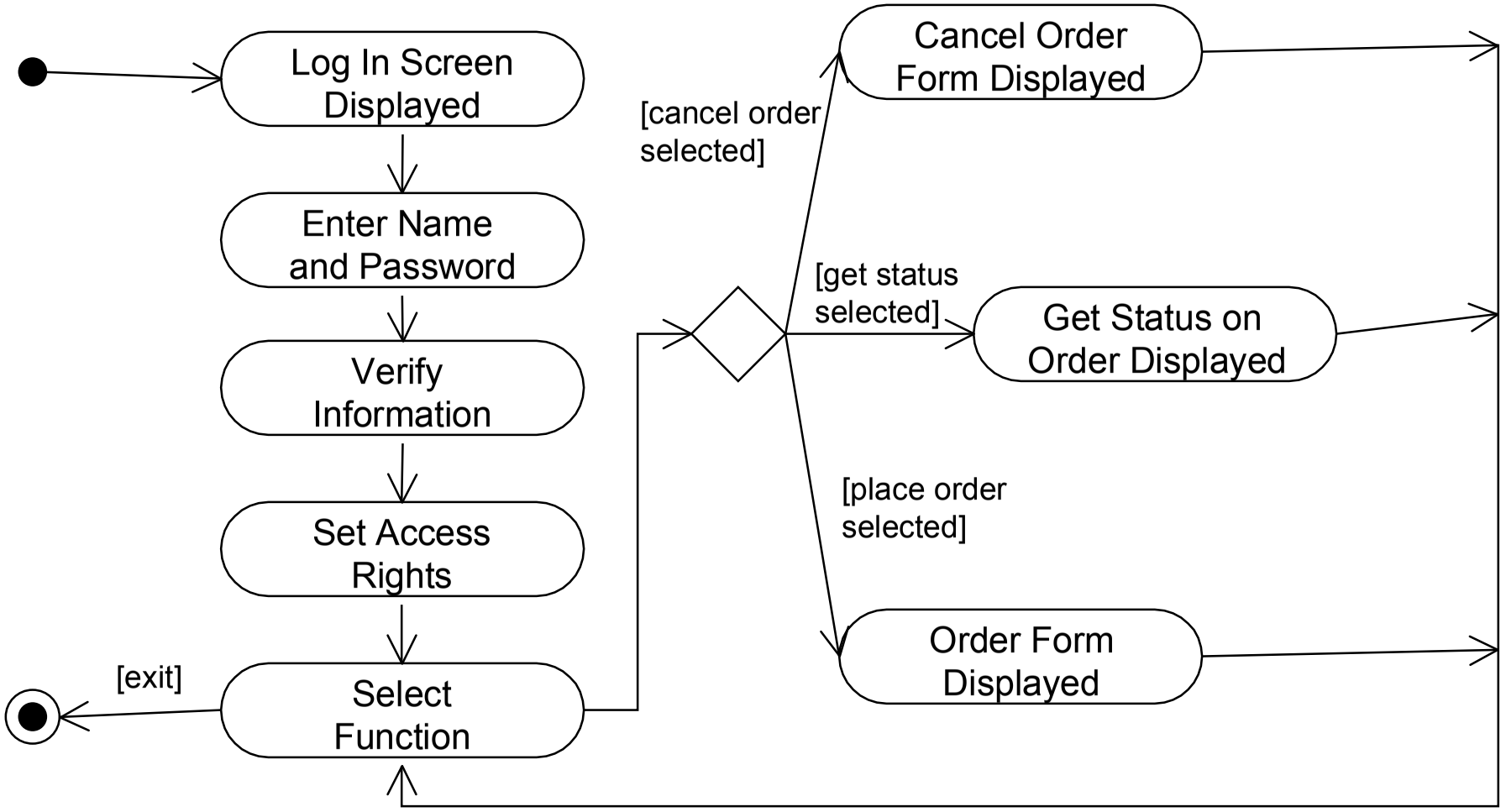
	male								female							
age <15	T	T	F	F	F	F	F	F								
age >=35	F	F	T	T	T	T	F	F								
age < 20									T	T	F	F	F	F	F	F
age > 40									F	F	T	T	T	T	F	F
insured > 20 years	F	F	F	F	T	T	F	F	F	T	F	T	F	T	F	T
has driver's license	F	T	F	T	T	F	F	T	F	F	T	T	F	F	T	T
Plan A							X	X								
Plan B															X	X
Plan C	X	X							X	X						
Plan D			X	X	X	X					X	X	X	X		
10% rebate		X		X	X			X		X			X	X		X
25% rebate					X	X						X		X		



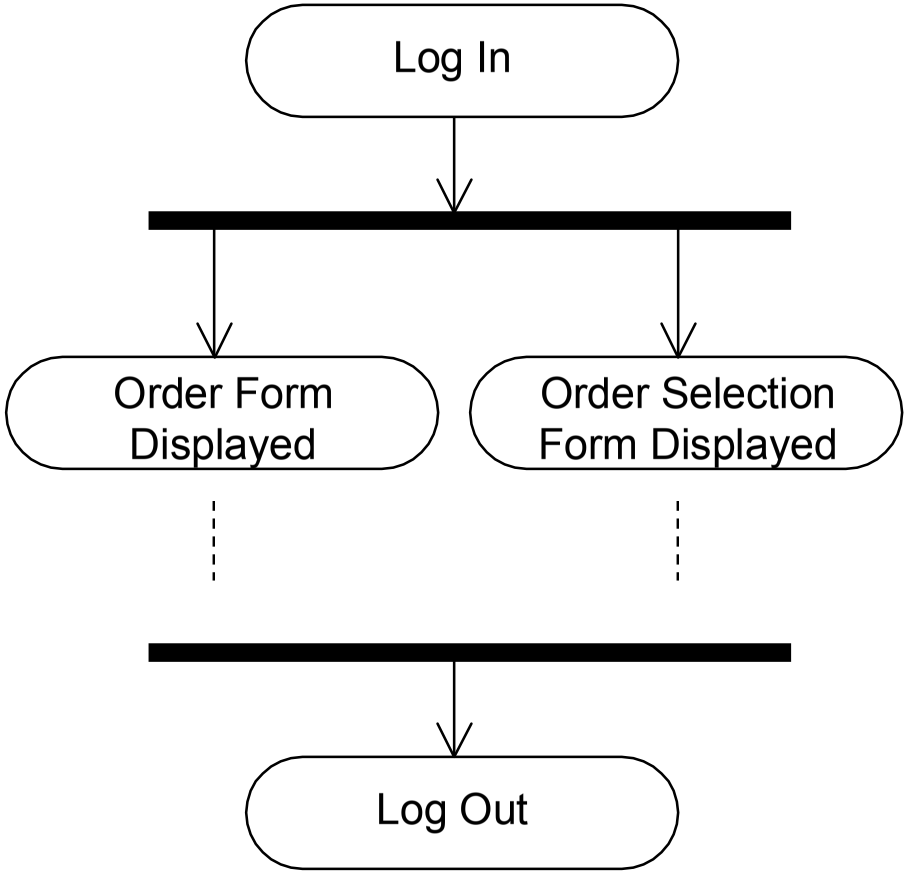
Activity Diagrams

- A state transition diagram where all states are action states
- Very easy to write and understand
- Enables a representation of branching, repetition and process forking
- Used to naturally represent flows of events in scenarios

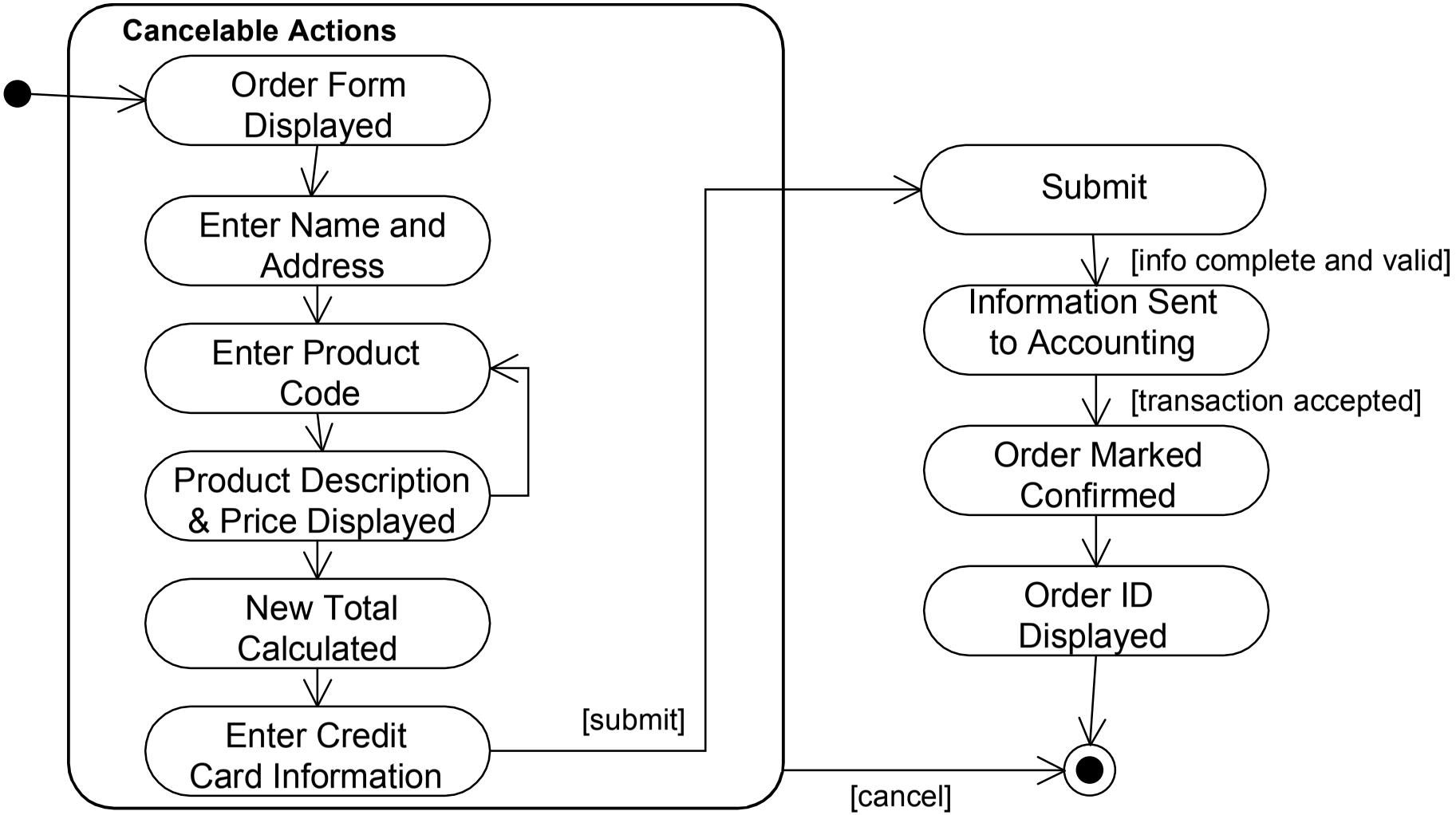
Activity Diagrams



Activity Diagrams



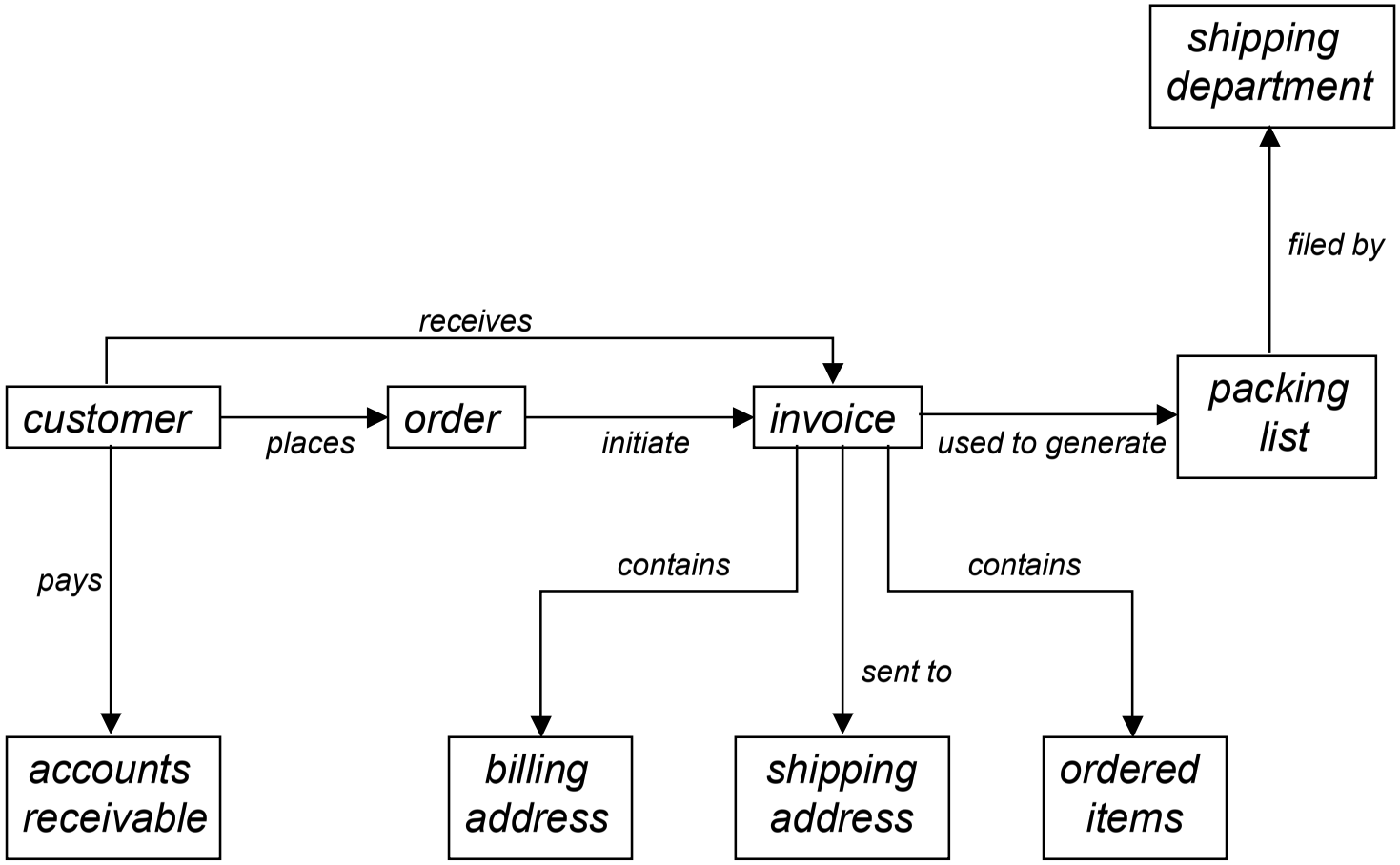
Activity Diagrams



Entity-relationship models

- Use ER diagram to represent the structure and relationships among *data* within the system.
- It provides a high-level "architectural" view of the system data.
- It focuses on the external behaviors of the system.

Entity-relationship models



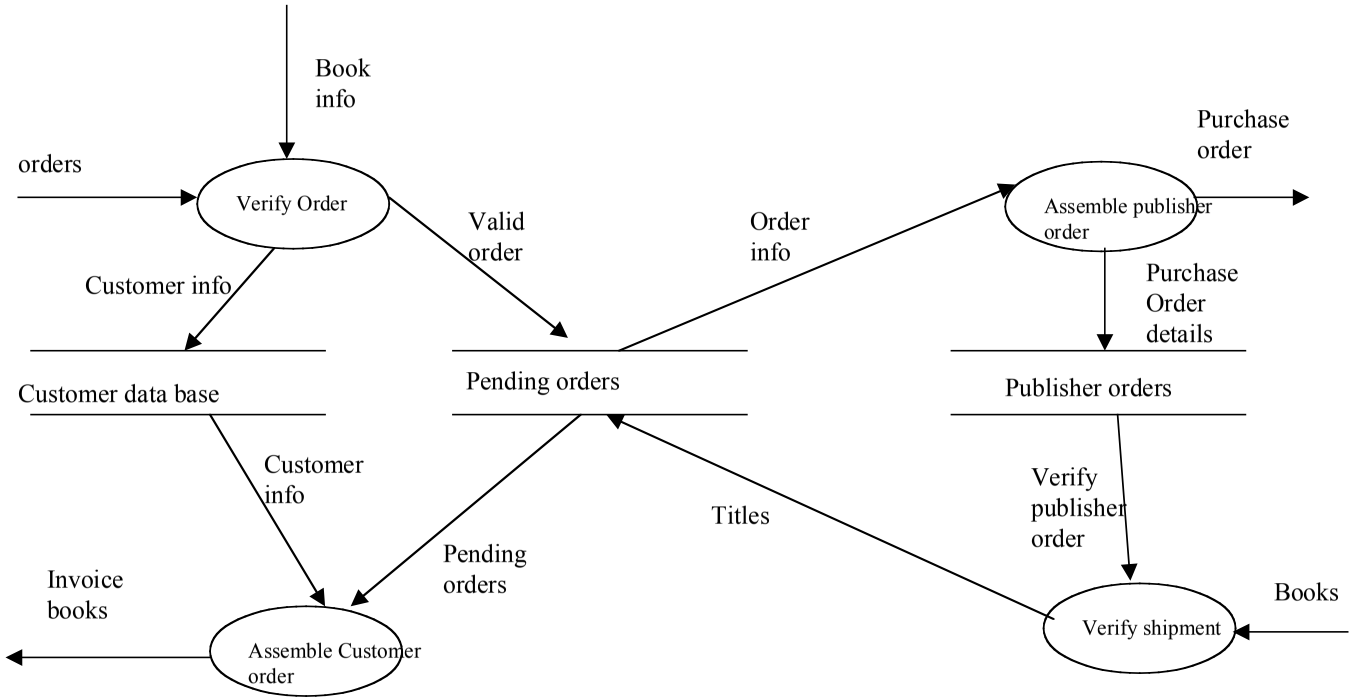
Entity-relationship models

- Problems
  - Might be difficult for non-technical readers to understand.
  - Different notations exist
  - Easy to fall in the trap of defining database structure, which is a design activity, not a specification activity.

Data flow diagrams

- High level description of the requirements.
- Visual presentation of the structure and the organization of the low-level requirements and the input/output relationship among them.
- Representation of the flow of data in the system
  - Data stores
  - Data filters
  - Relationships

Data flow diagrams



Data flow diagrams

- Could be further elaborated with lower-level diagrams to show the details.
- Could be the basis for communication between non-technical users and technical developers.
- Might be difficult for a non-technical reader to understand.

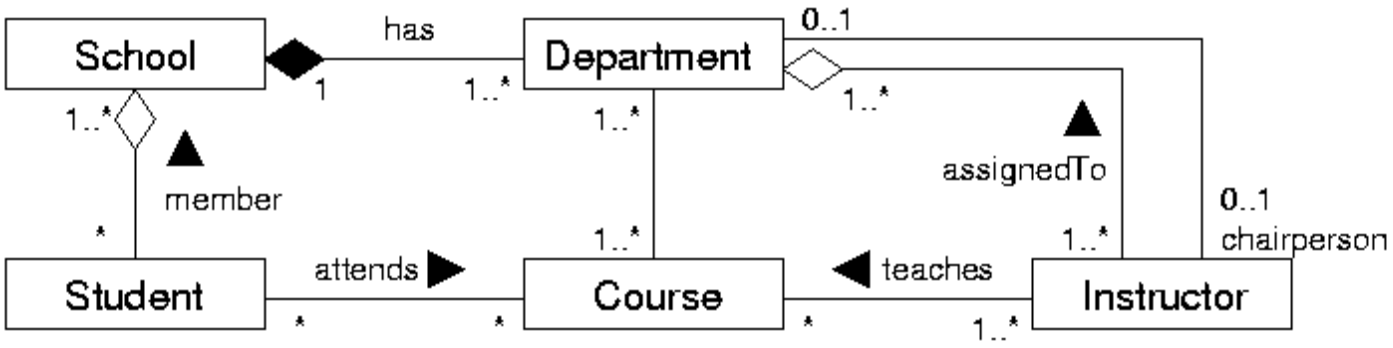
Object-oriented analysis

- Describe the structure and relationships among *entities* within the system.
- UCDA
  - Use case diagrams
  - Activity diagrams
  - Sequence diagrams
  - Collaboration diagrams
  - Class diagrams
- Almost all other kinds of diagrams and techniques can be be “objectified”.

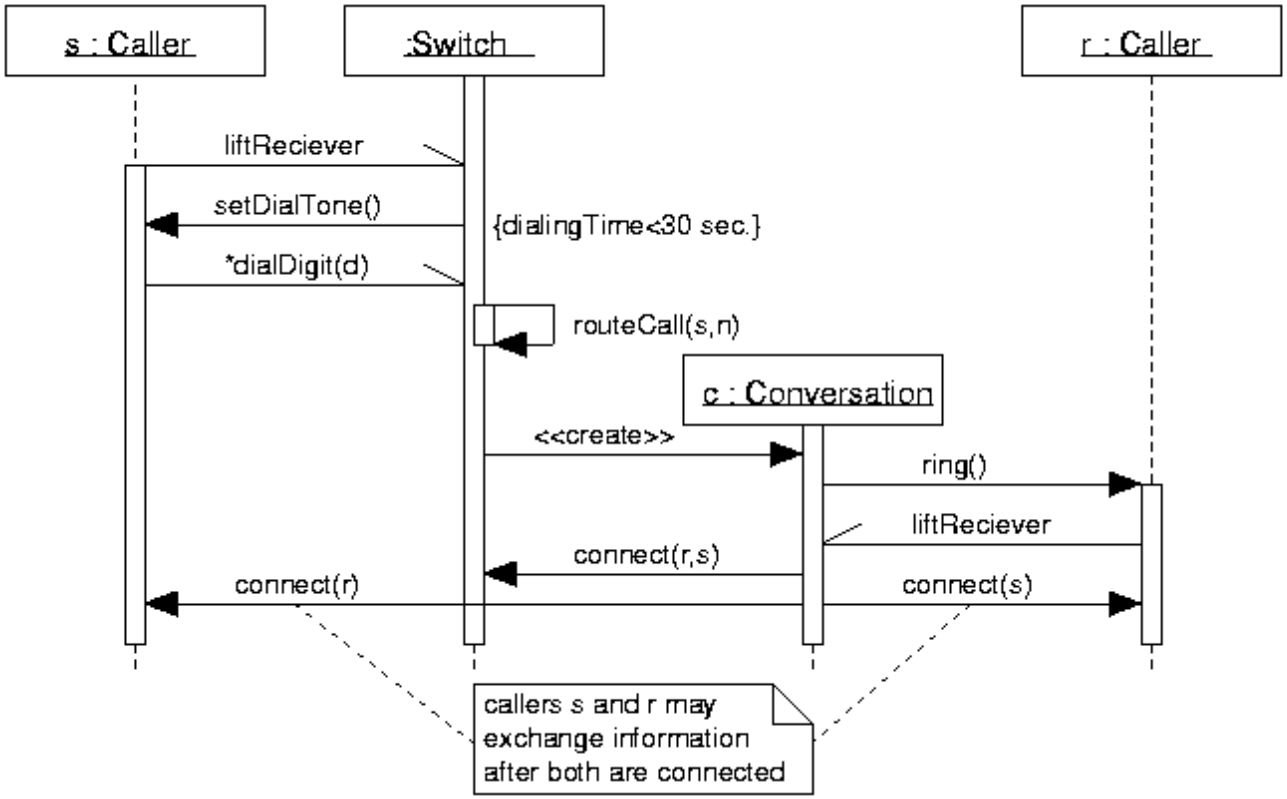
Object-oriented analysis

- Relatively new, but may be flourishing rapidly along with the popularity of OO and the adoption of the UML.
- More appropriate in the implementation models used to realize the functionalities of the system.

Object-oriented analysis



Object-oriented analysis



Formal specification languages: Z

- A formal specification notation based on set theory
- Has been developed at the Programming Research Group at the Oxford University Computing Laboratory since the late 1970s.
- Probably now the most widely-used formal specification language
- An international standard for the Z notation is being developed under the guidance of ISO.

Z as a specification language

- Specification are built from components called *schemas*.
- Schemas are specification building blocks
- Graphical presentation of schemas make Z specifications easier to understand
- Mathematical notation of schemas allows the building of formal completeness and consistency proofs to validate the specifications

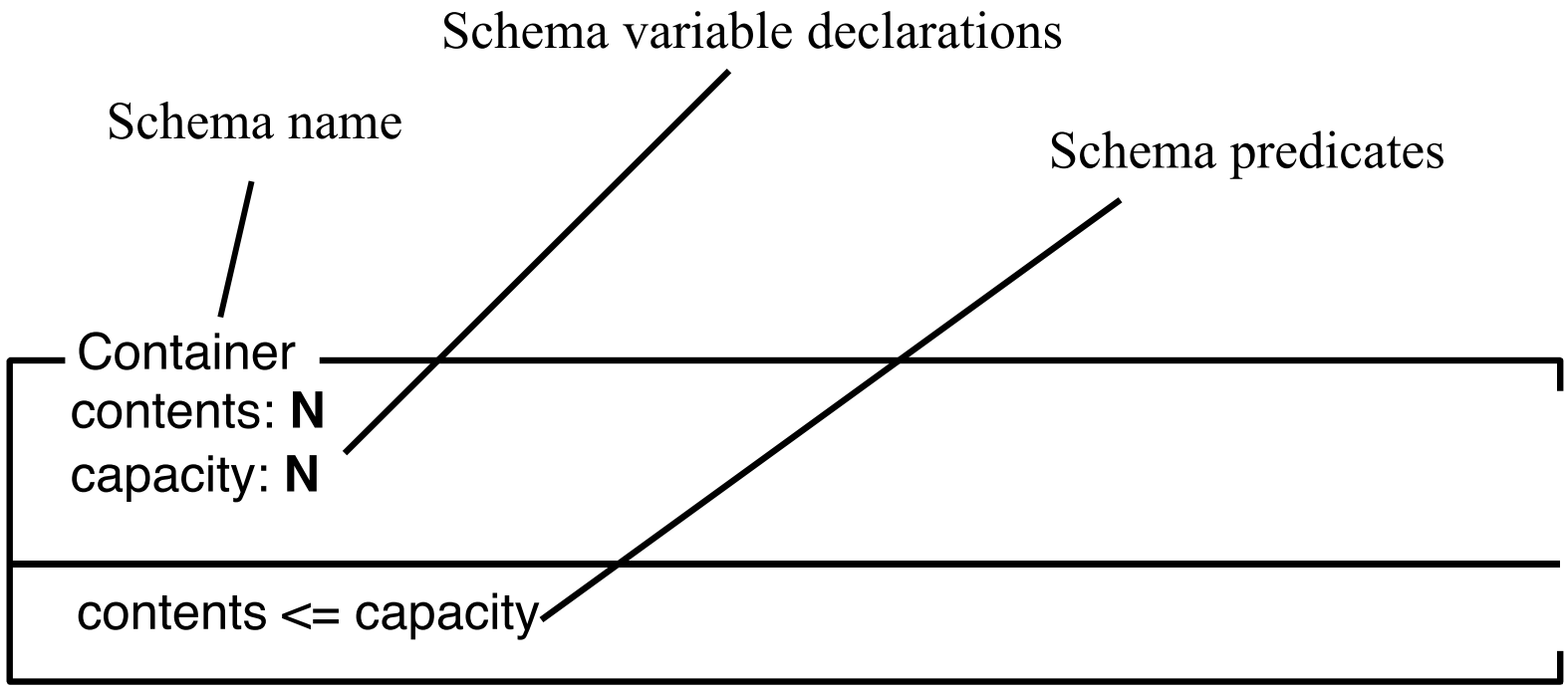
Z schemas

- Introduce specification entities and defines invariant predicates over these entities
- A *predicate* is a Boolean expression
- Some predicates can have side-effects that change the state of the entities involved
- Schemas can be included in other schemas and may act as type definitions
- Names are local to schemas
- In many respects, Z schemas are akin to objects.

Problem: storage tank monitoring

- A storage tank is monitored by a control system.
- The system consists of a container which holds something and a indicator panel which shows the current fill level and the danger level.
- If the storage tank goes over the danger level, a warning light must be lighted on the indicator panel.
- Fill operations must be specified as to ensure the tank to be refilled but never overfilled.

A container specification

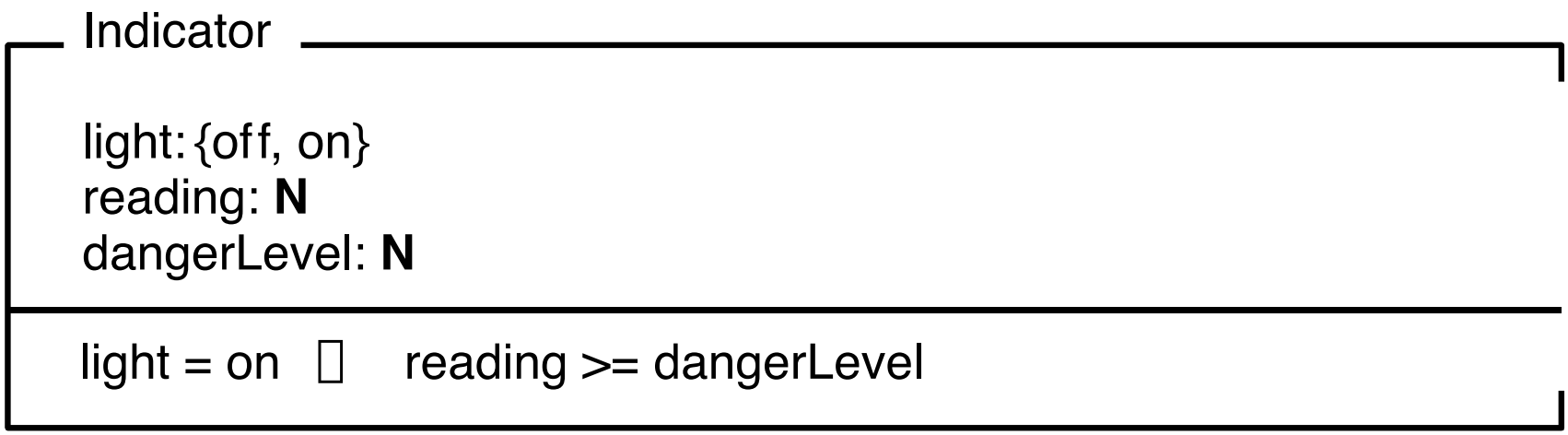


Container ≙ [contents: **N**; capacity: **N** | contents <= capacity]

The Z Notation

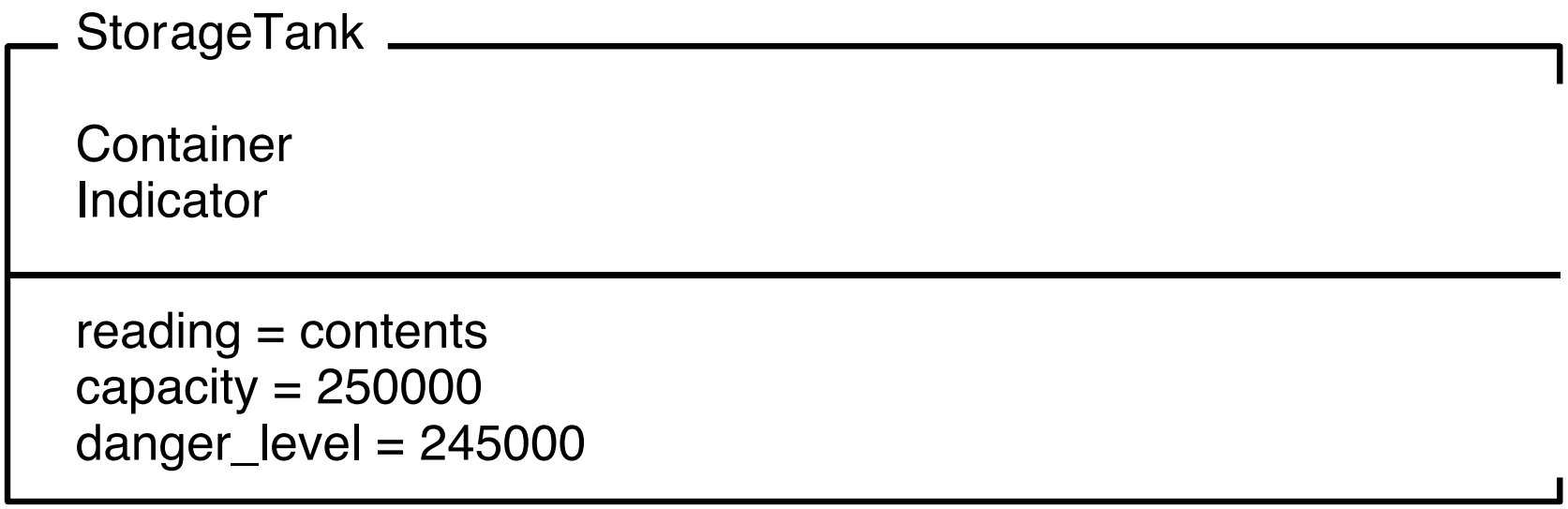
- A schema includes
  - A name identifying the schema
  - A signature introducing entities and their types
  - A predicate part defining relationships between the entities in the signature by stating a logical expression which must always be true (an invariant).

An indicator specification



- If the storage tank goes over the danger level, a warning light must be lighted on the indicator panel.

Specification of a storage tank



Specification of a storage tank

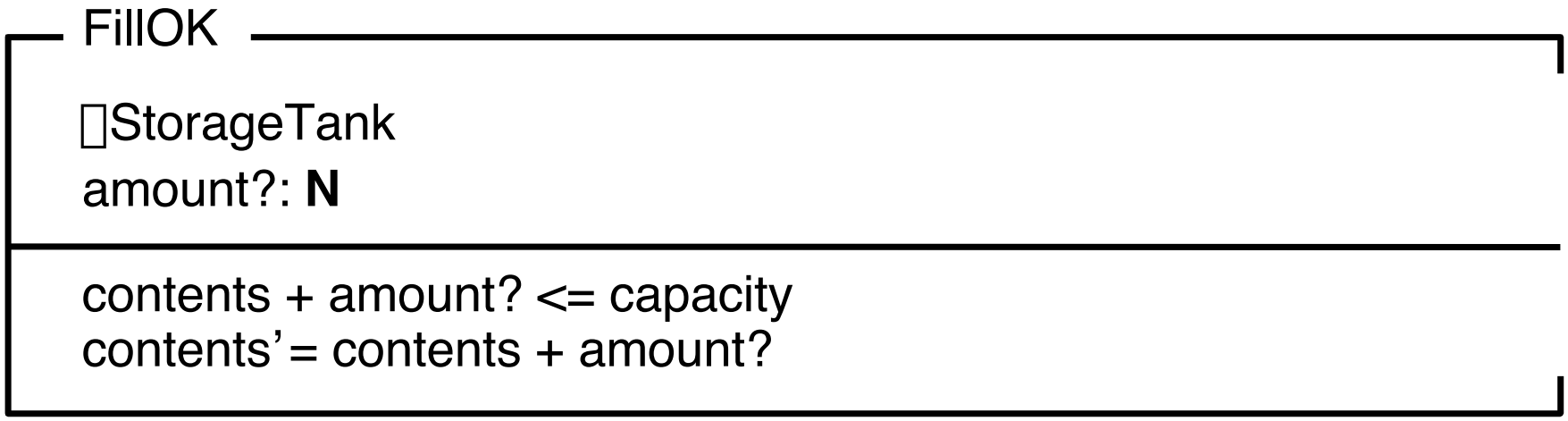
StorageTank

contents: **N**
capacity: **N**
reading: **N**
dangerLevel: **N**
light: {off, on}

---

contents <= capacity
light = on ⇒ reading >= dangerLevel
reading = contents
capacity = 250000
danger_level = 245000

A partial spec. of a fill operation



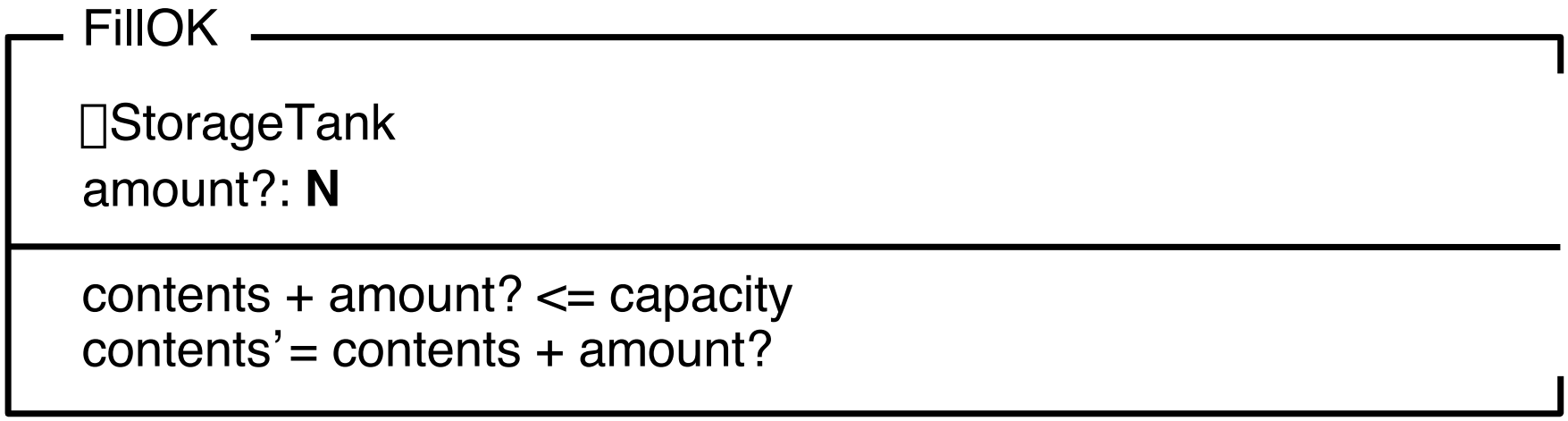
Z conventions

- A schema name prefixed by the Greek letter Delta (Δ) means that the operation changes some or all of the state variables introduced in that schema
- A variable name decorated with a ? represents an input
- A variable name decorated with a quote mark (N') represents the value of the state variable N after an operation

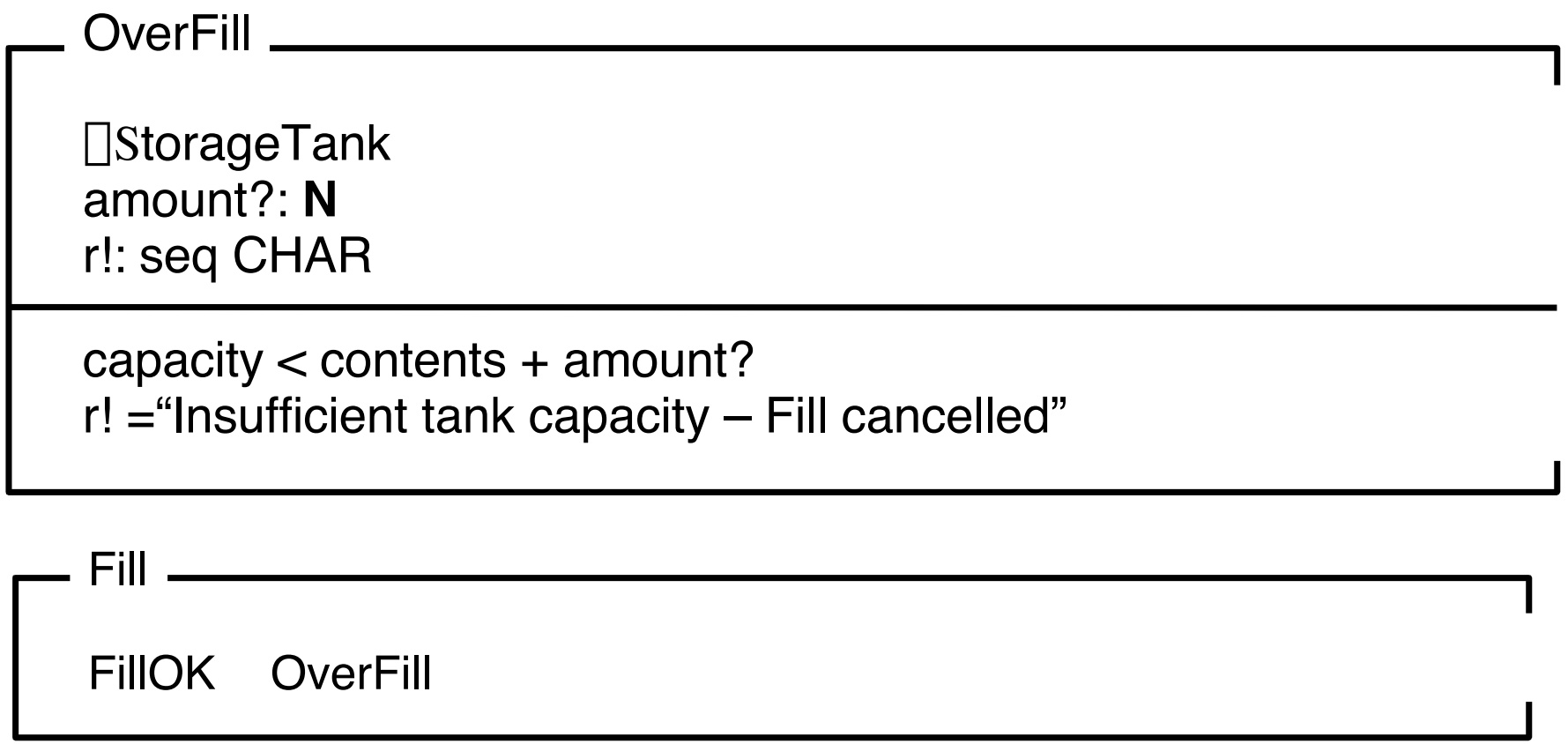
Z conventions

- A schema name prefixed by the Greek letter Xi (Ξ) means that the defined operation does not change the values of state variables
- A variable name decorated with a ! represents an output

A partial spec. of a fill operation



Storage tank fill operation



Full problem specification

Container ≙ [content: **N**; capacity: **N** | content <= capacity]

Indicator ≙ [light:{off,on};reading: **N**; dangerLevel: **N**
| light = on ⇒ reading >= dangerLevel]

StorageTank ≙ [Δ Container; Δ Indicator | reading=content;
capacity=250000; dangerLevel = 245000]

FillOK ≙ [Δ StorageTank; amount?: **N** |
contents + amount? <= capacity;
content' = content + amount?

OverFill ≙ [Δ StorageTank; amount?: **N**; r!: seq CHAR |
capacity < contents + amount?;
r! ="Insufficient tank capacity – Fill cancelled"

Fill ≙ [FillOK ∨ OverFill]

Operation specification

- Define the “normal” operation as a schema
- Define schemas for exceptional situations
- Operations may be specified incrementally as separate schema then the schema combined to produce the complete specification
- Combine all schemas using the disjunction (or) operator (written)

Z Notations

- A schema includes
  - A name identifying the schema
  - A signature introducing entities and their types
  - A predicate part defining invariants over these entities
- Conventions
  - Marks before a schema,
    - Delta (Δ) means *changes*
    - Xi (Ξ) means no *changes*
  - Marks after a state variable,
    - ? means *input*
    - ! means *output*
    - ‘ represents the value of the variable after an operation

Z symbols

Sets:	
S : **P** X	S is a set of Xs
S : **F** X	S is a finite set of Xs
x ∈ S	x is a member of S
x ∉ S	x is not a member of S
S ⊆ T	S is a subset of T
S ∪ T	union of S and T
S ∩ T	intersection of S and T
S \ T	difference of S and T
∅	empty set
N	set of natural numbers : {0,1,2, …}
Z	set of integer numbers : {...-2,-1,0,1,2, …}
max(S)	maximum value of the (non-empty) set S
min(S)	minimum value of the (non-empty) set S
#S	cardinality of S

Z symbols

Functions:	
$f : X \pfun Y$	f is a partial function from X to Y
$f : X \fun Y$	f is a total function from X to Y
dom f	domain of f
ran f	range of f
$f : \{x_1 \mapsto y_1, x_2 \mapsto y_2\}$	extensional function definition
$f \oplus \{x_1 \mapsto y_1\}$	extensional addition of mappings

Logic:	
$\neg P$	negation
$P \wedge Q$	conjunction
$P \vee Q$	disjunction
$P \Leftrightarrow Q$	equivalence
$P \Rightarrow Q$	implication

Summary

- **Definition:** *"Specifications represent a model of how system inputs are related to system reactions and outputs"*
- Extremely valuable input for the system developers
- Different techniques can be used. The choice is made according to the situation
- Level of details needed depend on:
  - The funds available for such an analysis
  - Criticality of the system
  - Level of understanding of the problem
- Beware: adding too much details might hinder the developers rather than help them