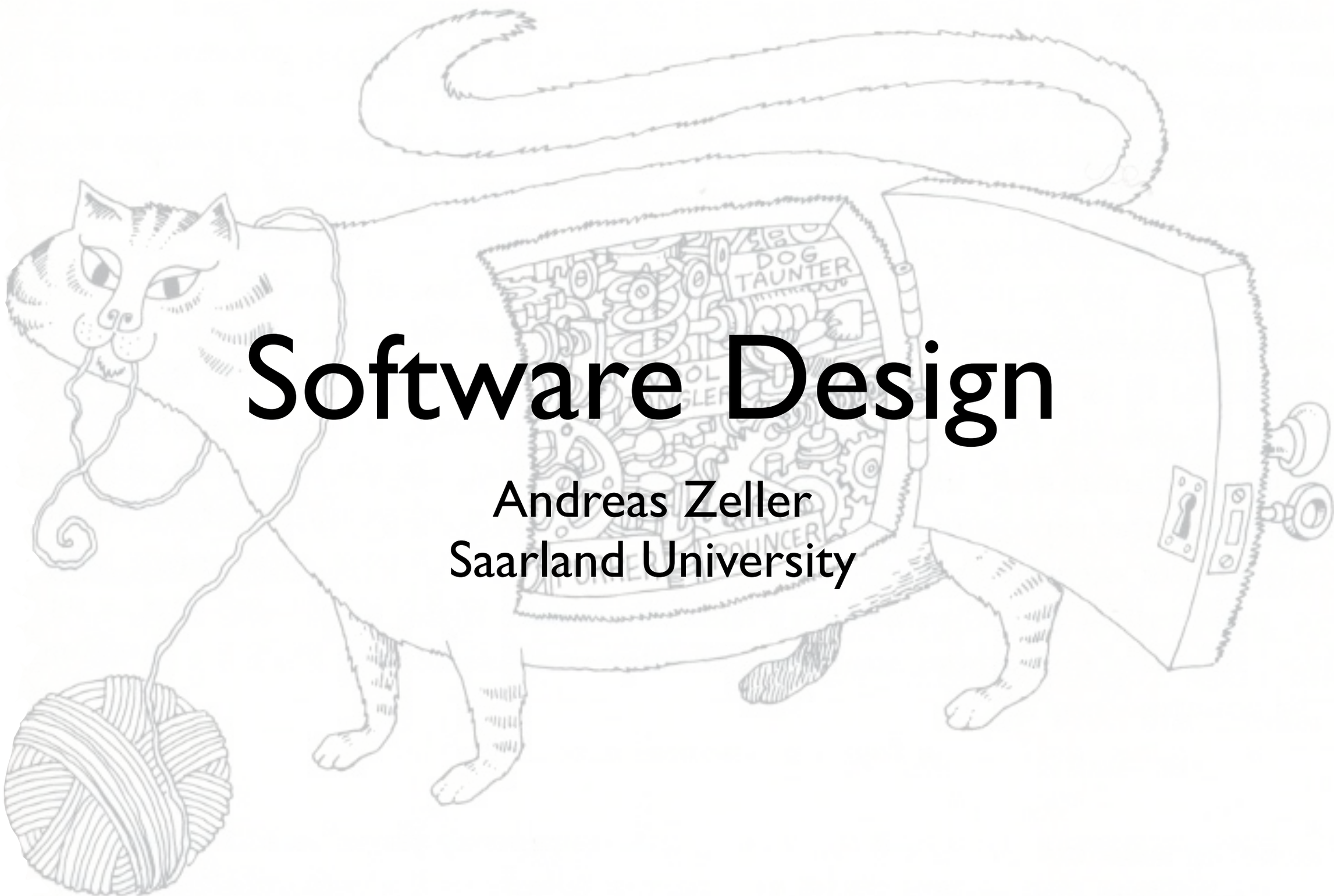


Software Design

Andreas Zeller
Saarland University

with slides from Gregor Snelting, KIT

Object-Oriented Design

The challenge: how to choose the components of a system with regard to

- similarities
- later changes.

This is the purpose of *object-oriented design*.

What's an Object?

An object offers

- a collection of *services* (methods) that work on
- a common state.

There is usually a correspondence between

- *objects* and *nouns* in the task
  ("Bug", "Field", "Marker")
- *methods* and *verbs* in the task
  ("move", "sit down", "delete")

Object-Oriented Modeling in UML

includes the following design aspects:

- *Object model*: *Which* objects do we need?
    - Which are the *features* of these objects?
      (attributes, methods)
        - How can these objects be *classified*?
          (Class hierarchy)
        - What *associations* are there between the classes?
- *Sequence diagram*: How do the objects *act together*?
- *State chart*: What *states* are the objects in?

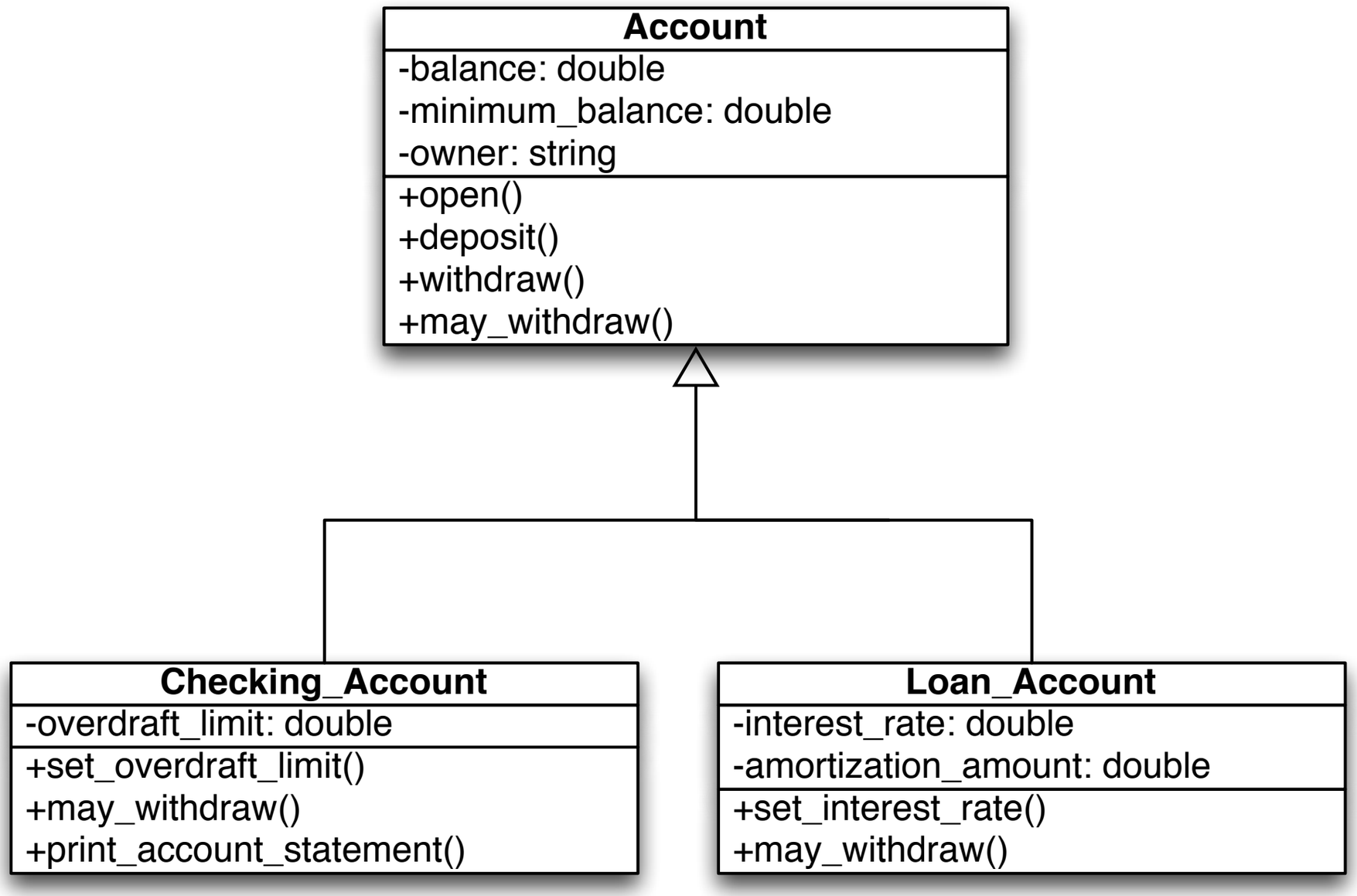
Object-Model: Class Diagram

Every class is represented by a *rectangle*, divided into:

- class name
- attributes – preferably with type information (usually a class name)
- methods – preferably with a signature

Class inheritance is represented by a *triangle* (△) connecting subclasses to superclasses.

Example: Accounts



Inherited methods (e.g. open(), deposit()) are not listed separately in subclasses.

Definitions in a subclass *override* those of the superclass (e.g. may_withdraw())

Abstract Classes and Methods

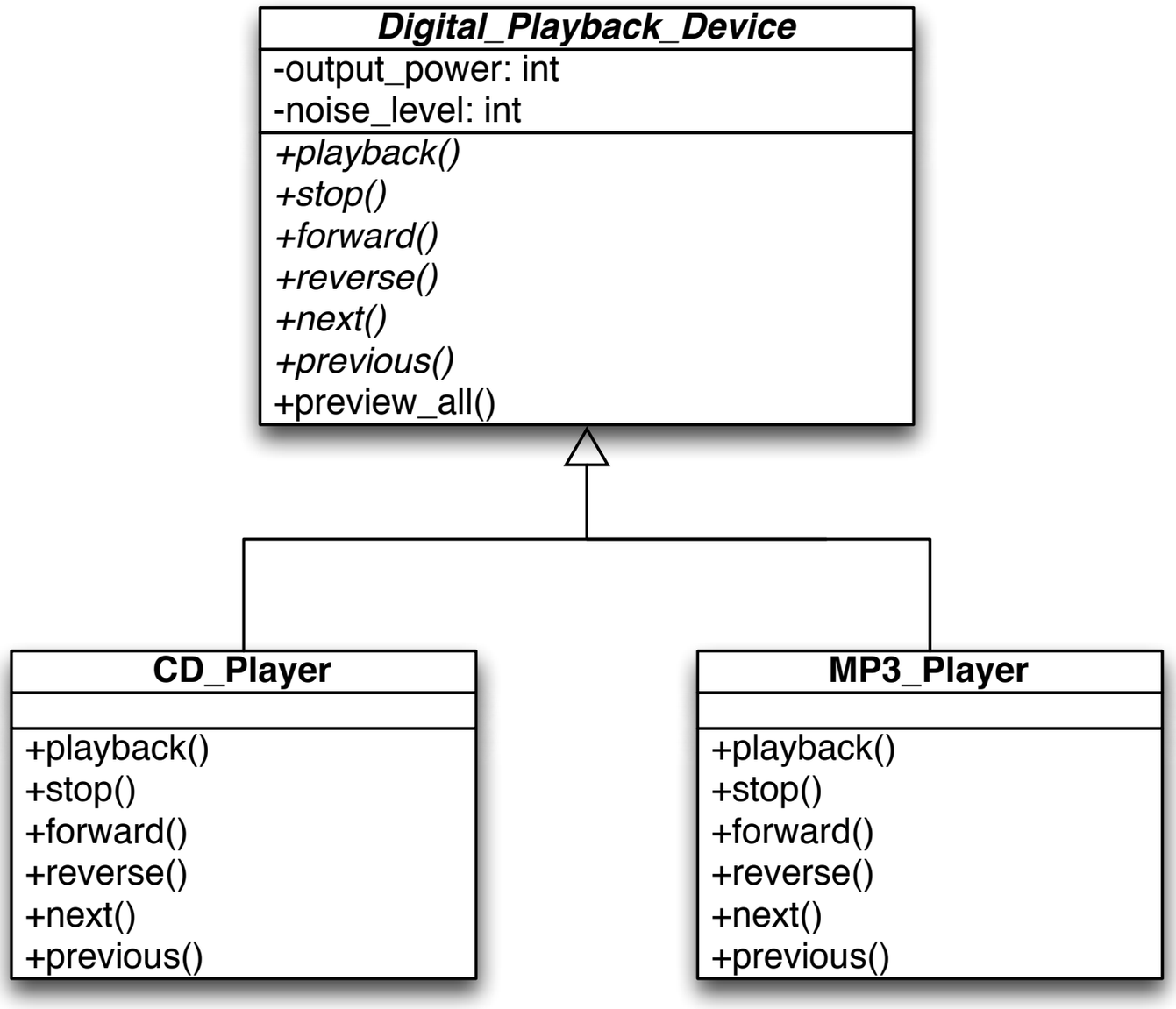
Abstract classes cannot exist as concrete objects(instances).

Usually they have one or multiple *abstract methods* which are implemented only in subclasses.

Concrete classes, on the other hand, can exist as concrete objects.

Example: Abstract Classes

"Digital playback device" is an abstract concept of its *concrete* implementations – e.g. CD-player or MP3-player.



Italicized class/method name indicates abstract class/method.

Default Values and Constraints

The attributes of an object can be provided with *default values*.

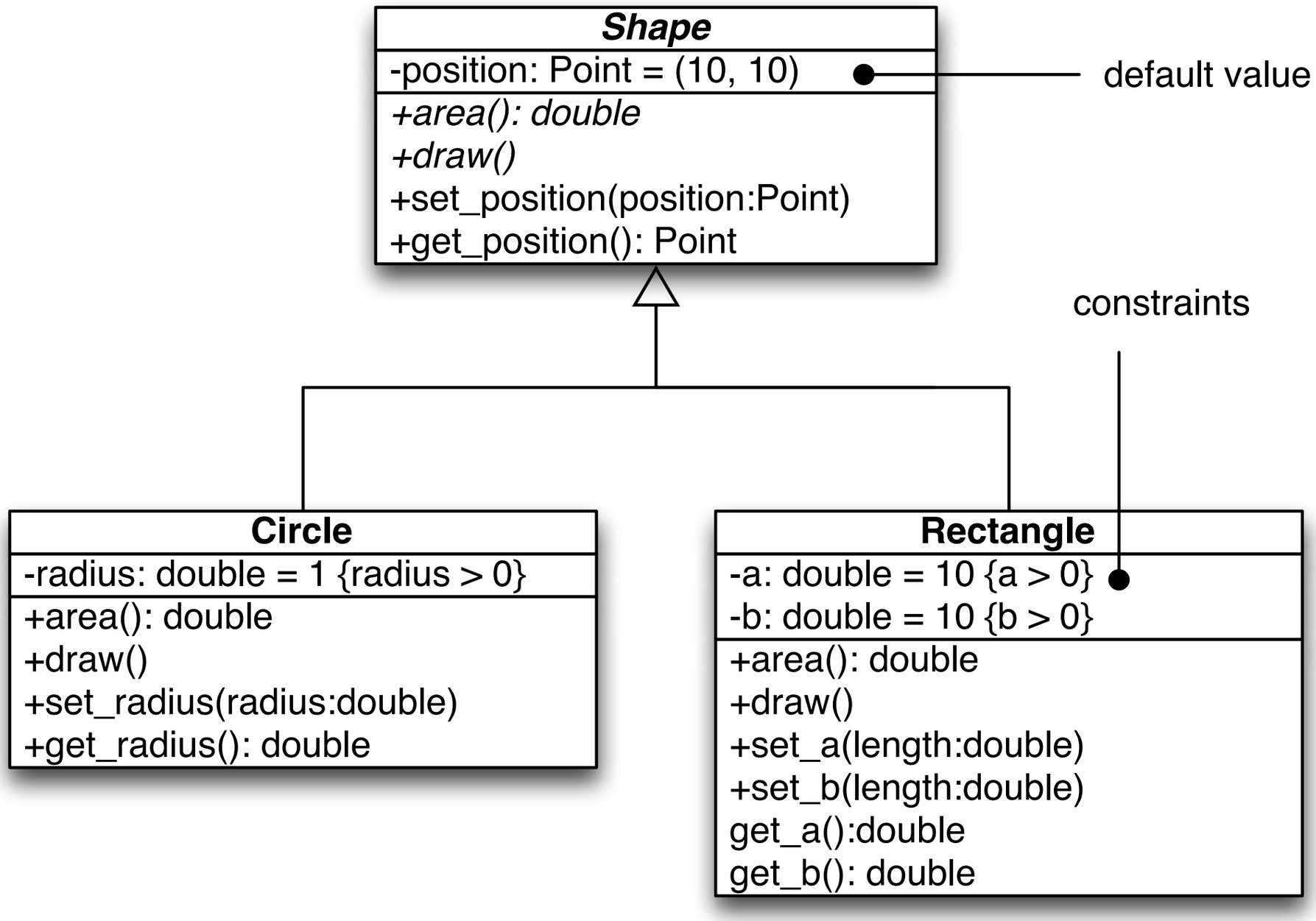
These will be used by *default* if nothing is specified upon construction.

Also, *constraints* can be used to specify *requirements* on attributes.

This allows us to express *invariants*: object properties that always hold.

Example: Constraints

These constraints ensure that circles always have a positive radius, and rectangles positive side lengths.



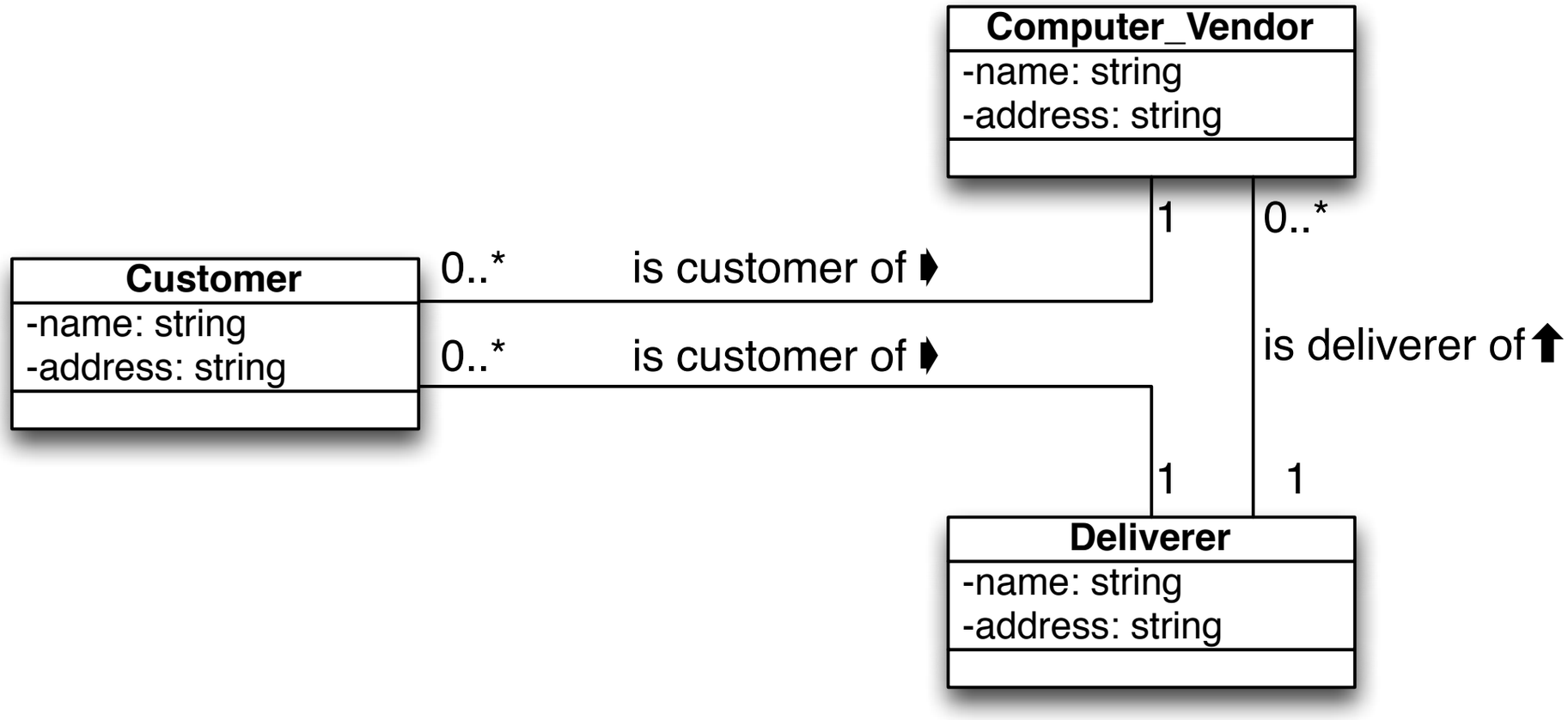
Object-Model: Associations

General associations

- Connections between non related classes represent associations(relations) between those classes.
- These describe the *semantic connection* between objects (cf. database theory).
- The number of associated objects is restricted by means of *multiplicity*.

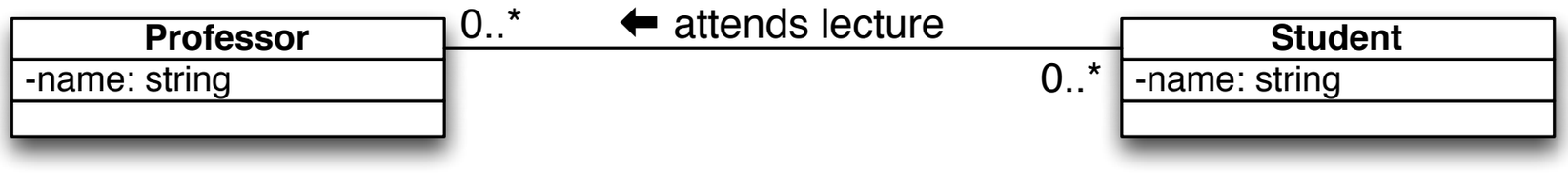
Example: Multiplicity

A *computer vendor* has multiple *customers*, a *delivery agency* also has multiple customers, but the computer vendor has only one delivery agency.

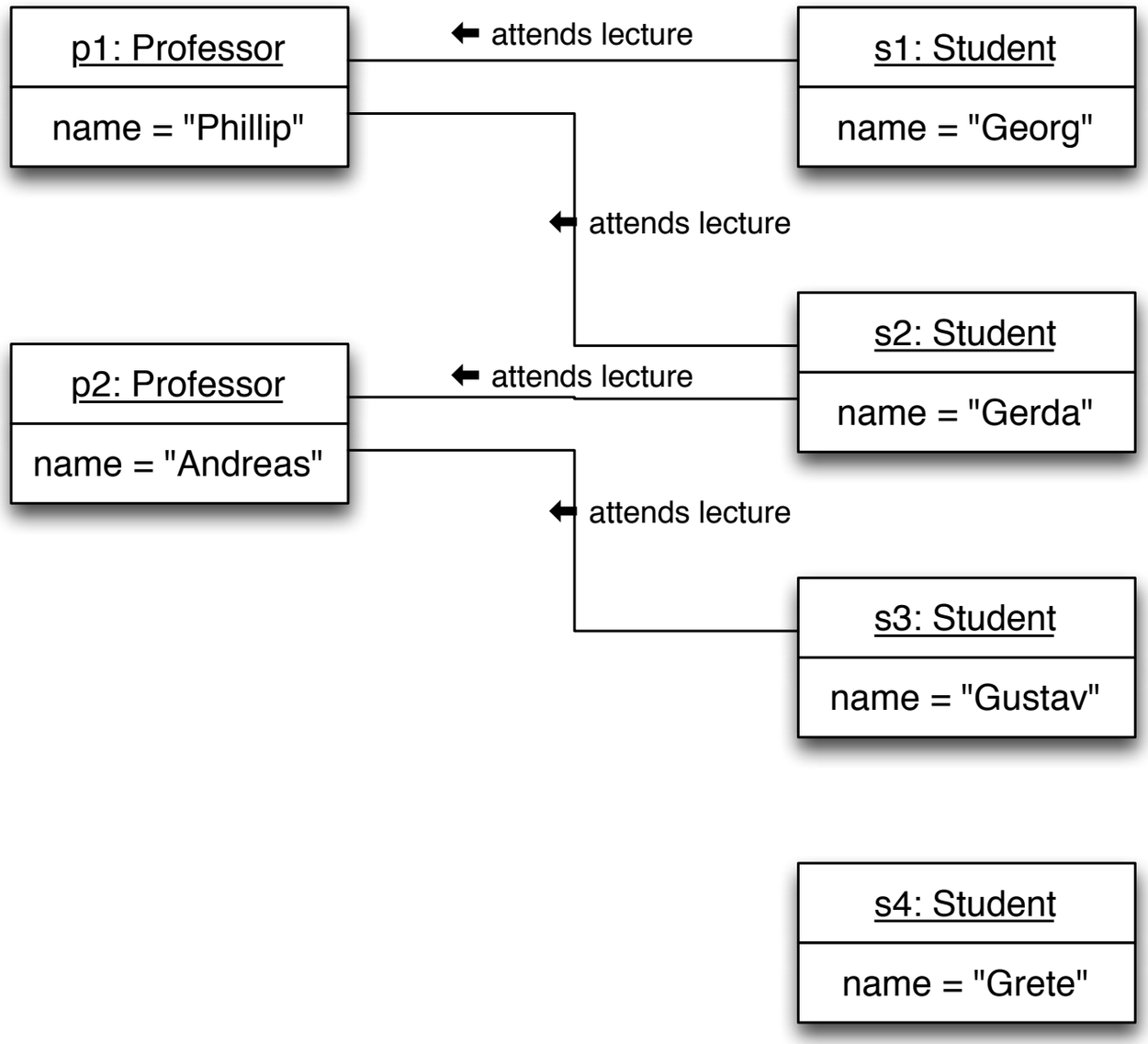


Example: Multiplicity (2)

Professors have multiple *students*, and students have multiple professors.



Example: Relationships between Objects



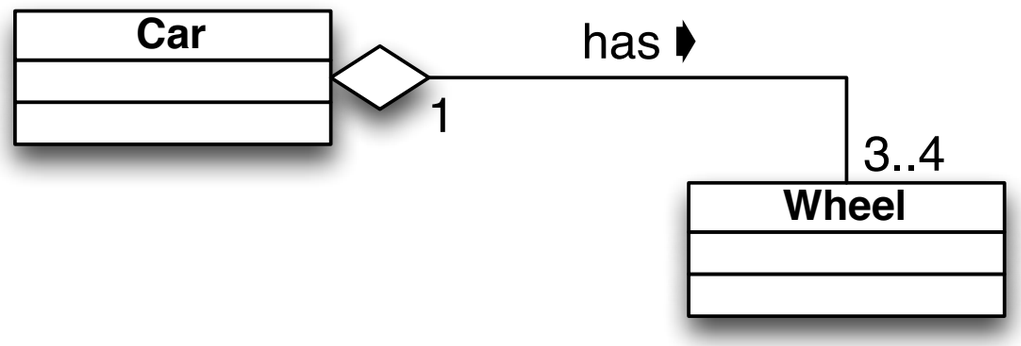
Underlined names indicate concrete objects(instances), which have concrete values for their attributes.

Aggregation

The *has*-relation is a very common association as it describes the hierarchy between a whole and parts of it.

It is marked with the symbol ◇

Example: A car has 3–4 wheels.

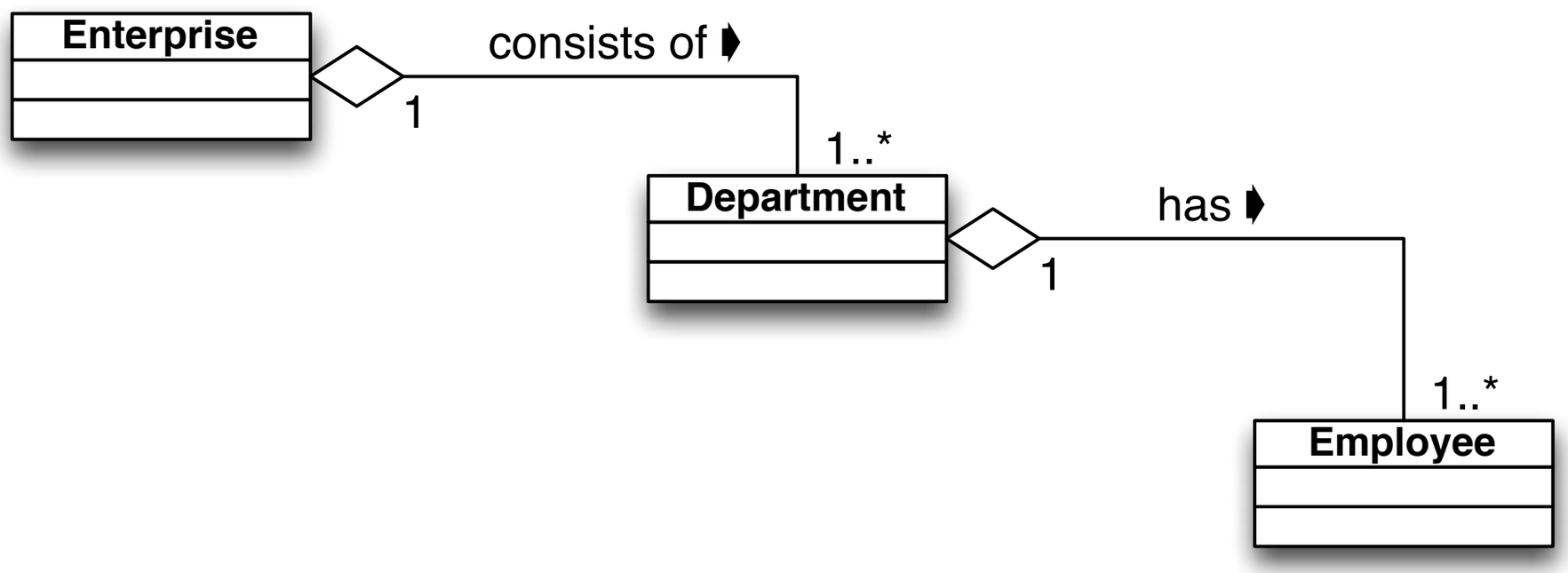


A Car

A Car

Aggregation (2)

Another example: An enterprise has 1..* departments with 1..* employees each.



Aggregation (3)

It is possible for an aggregate to be empty (usually at the beginning): the multiplicity 0 is allowed. However, its purpose is to collect parts.

The aggregate as a whole is *representative of its parts*, i.e. it takes on tasks that will then be propagated to the individual components.

e.g. The method `computeRevenue()` in an Enterprise class sums up the revenues of all the departments.

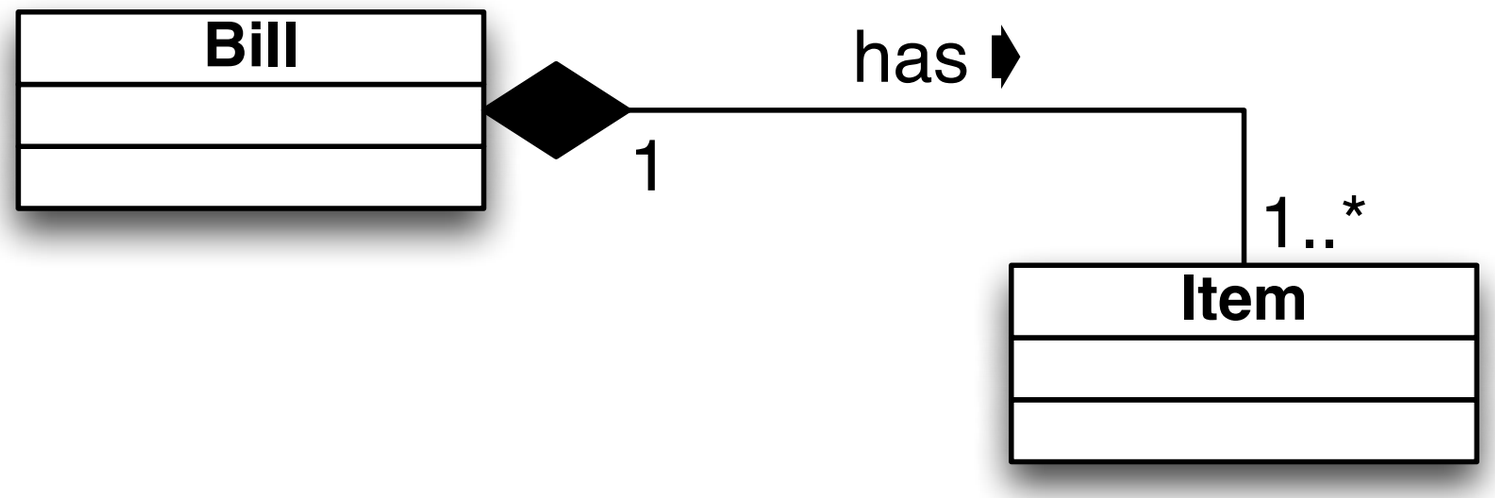
Composition

A special case of the aggregation, the *composition*, is marked with ♦

An aggregation is a composition when the part *cannot exist* without the aggregate.

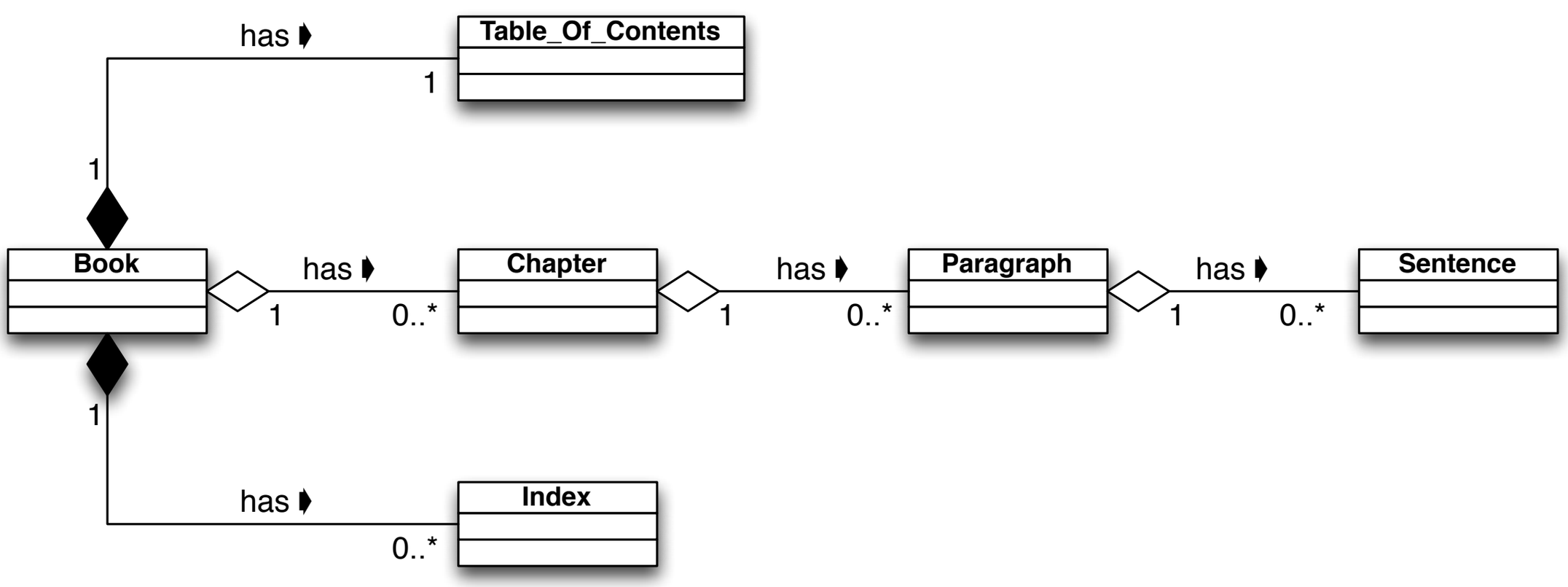
Example: Bill Item

A bill item always
belongs to a bill.



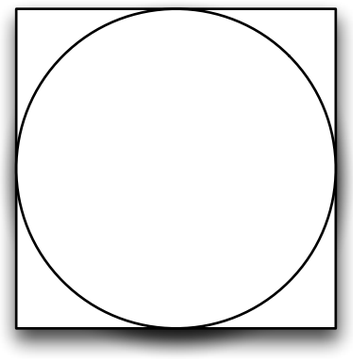
Example: Book

A *book* consists of a *table of contents*, multiple *chapters*, an *index*;
a chapter, in turn, consists of multiple *paragraphs*, and so on.



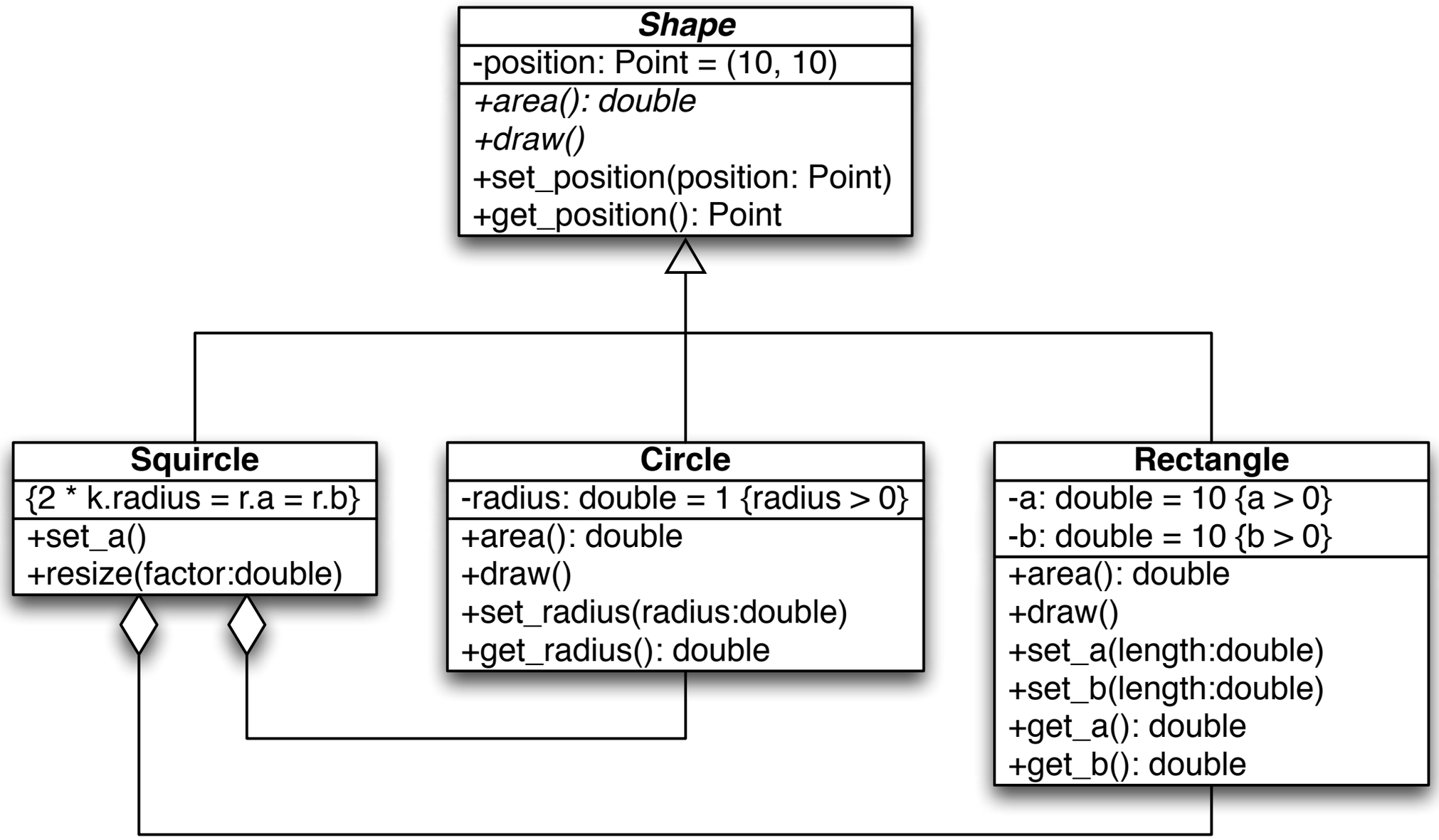
Example: Squircle

A "squircle" consists of a circle on top of a square:



Example: Squircle (2)

A squircle can be modeled as a Squircle class that contains a circle as well as a square:



Addenda

A component can only be part of *one* aggregate.

A class can also be viewed as a composition of all its attributes.

In many programming languages *aggregations* are implemented by using references (pointers to objects); however, *compositions* are values.

Sequence Diagrams

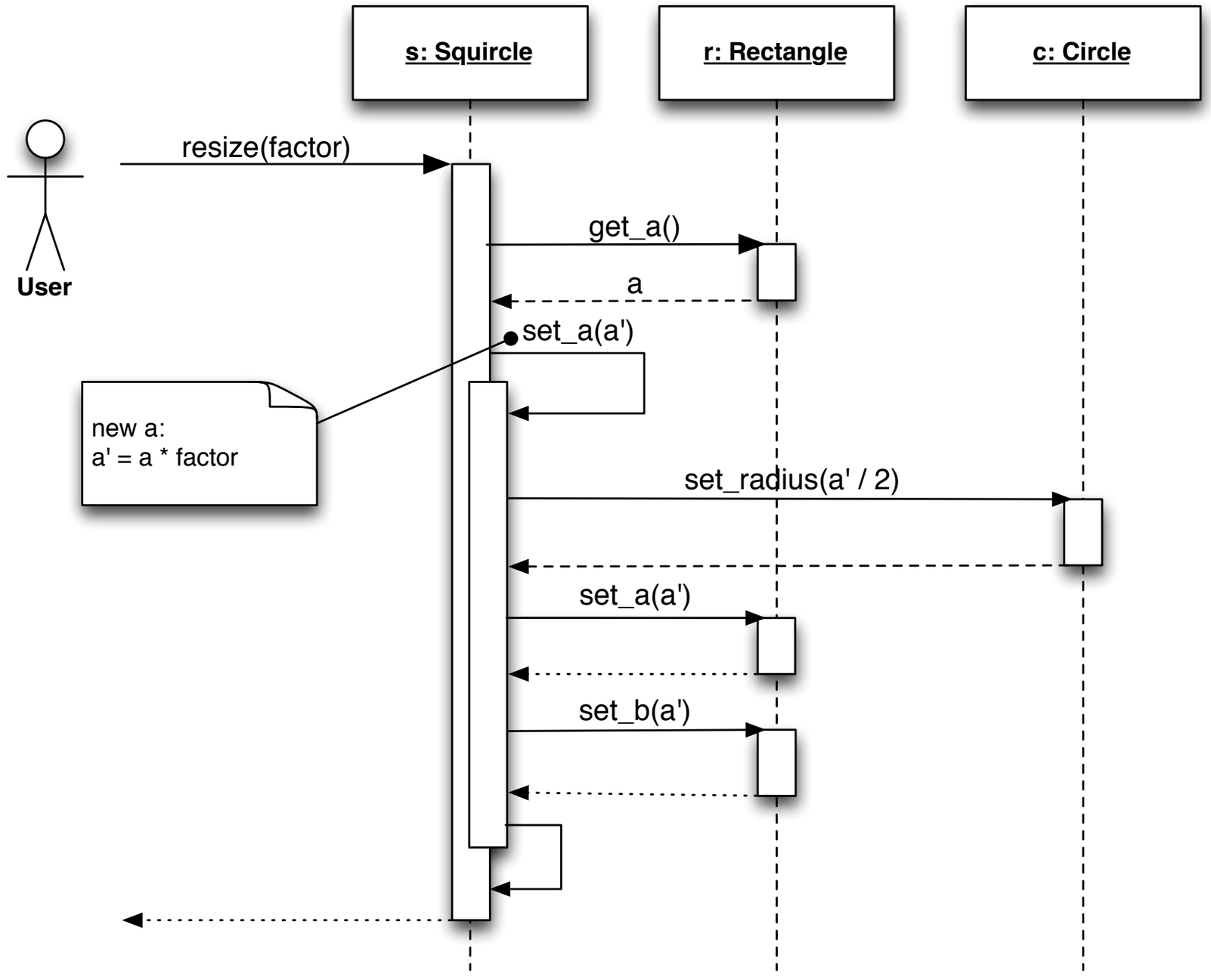
A *sequence diagram* reflects the flow of information between individual objects with an emphasis on *chronological order.*

Objects are depicted as vertical *lifelines*; the time progresses from top to bottom.

The *activation boxes*(drawn on top of lifelines) indicate active objects.

Arrows ("Messages") represent the flow of information – e.g. method calls (solid arrows) and return (dashed arrows).

Example: Resizing a Squircle



State Charts

A state chart displays

- a sequence of *states* that an object can occupy in its lifetime, and
- which *events* can cause a change of state.

A state chart represents a *finite state machine*.

State Transitions

State transitions are written as

event name [*condition*] / *action*

where

- *event name* is the name of an event (usually a method call)
- *condition* is the condition on which the transition occurs (optional)
- *action* is the action taken when the transition occurs (optional).

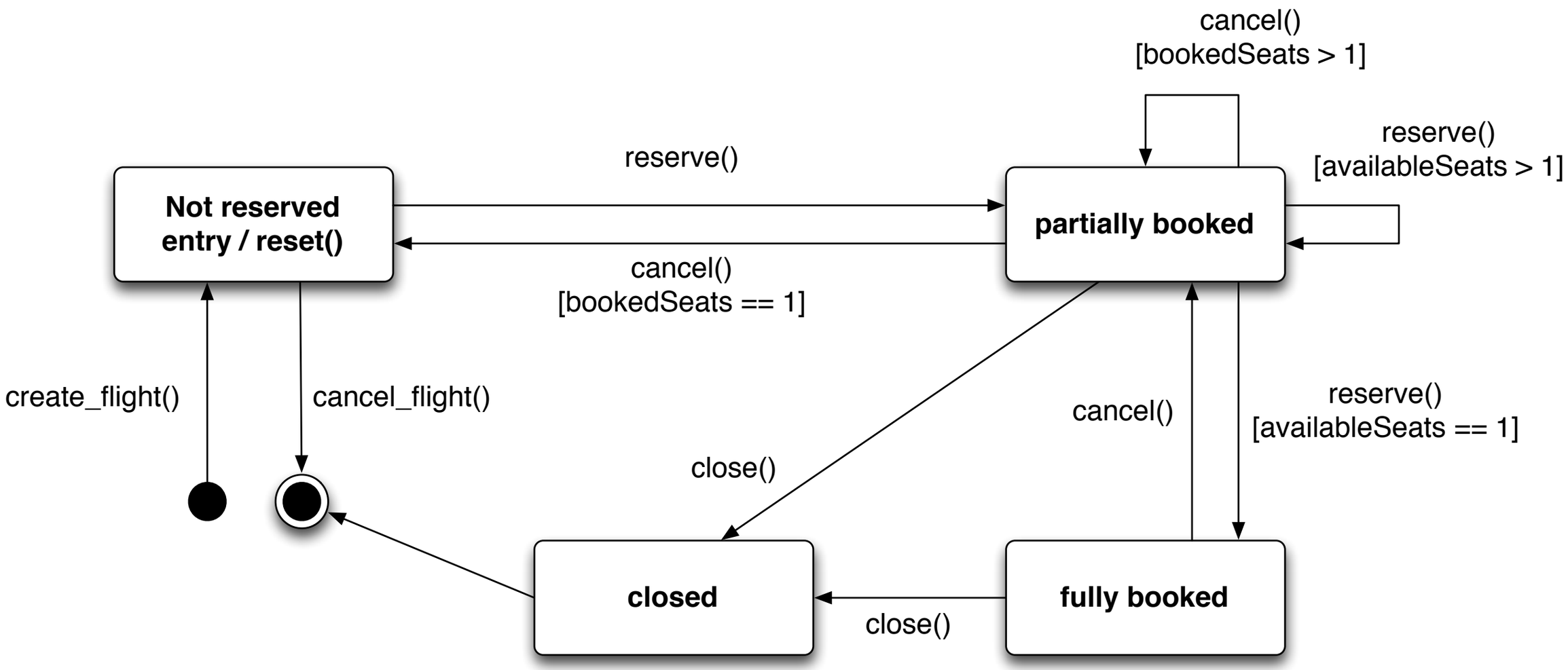
State Actions

States can also be annotated with *actions*: The *entry* event denotes the reaching of a state; the *exit* event describes the leaving of a state.

Example: Booking a Flight

When a flight is first created, nothing is booked yet.

The action reset() causes the number of free and reserved seats to be reset.



Example: ADAC

Devising Classes and Methods

"How do I come up with the objects?" is the most difficult question of the analysis.

There is no one single answer: it is possible to model any problem in multiple object-oriented ways.

Leveraging Use Cases

1. Describe *typical scenarios* by means of *use cases*

2. Extract *central classes and services* from the use cases

Use Cases

- Describe how an actor can reach his goal
- What *actors* are there, and what goals do they have?

Definitions

- An *actor* is something, that can exhibit behavior (e.g. person, system, organization)
- A *scenario* is a sequence of actions and interactions between actors
- A *Use Case* is a collection of related scenarios consisting of successful scenario and alternative scenarios

Example: Shipping of a PC

A student called Fritz orders a PC at the WorldOfPC company via a letter. After some time the PC is delivered to him in a package by the ShippingDeliverer shipping company.

- Who are the actors?
- What goals do they have?
- What can go wrong?

Example: Shipping of a PC

A student called Fritz orders a PC at the WorldOfPC company via a letter. After some time the PC is delivered to him in a package by the ShippingDeliverer shipping company.

Alternative scenarios:

- Order does not arrive

  – cannot be served –

- Computer (not yet) available

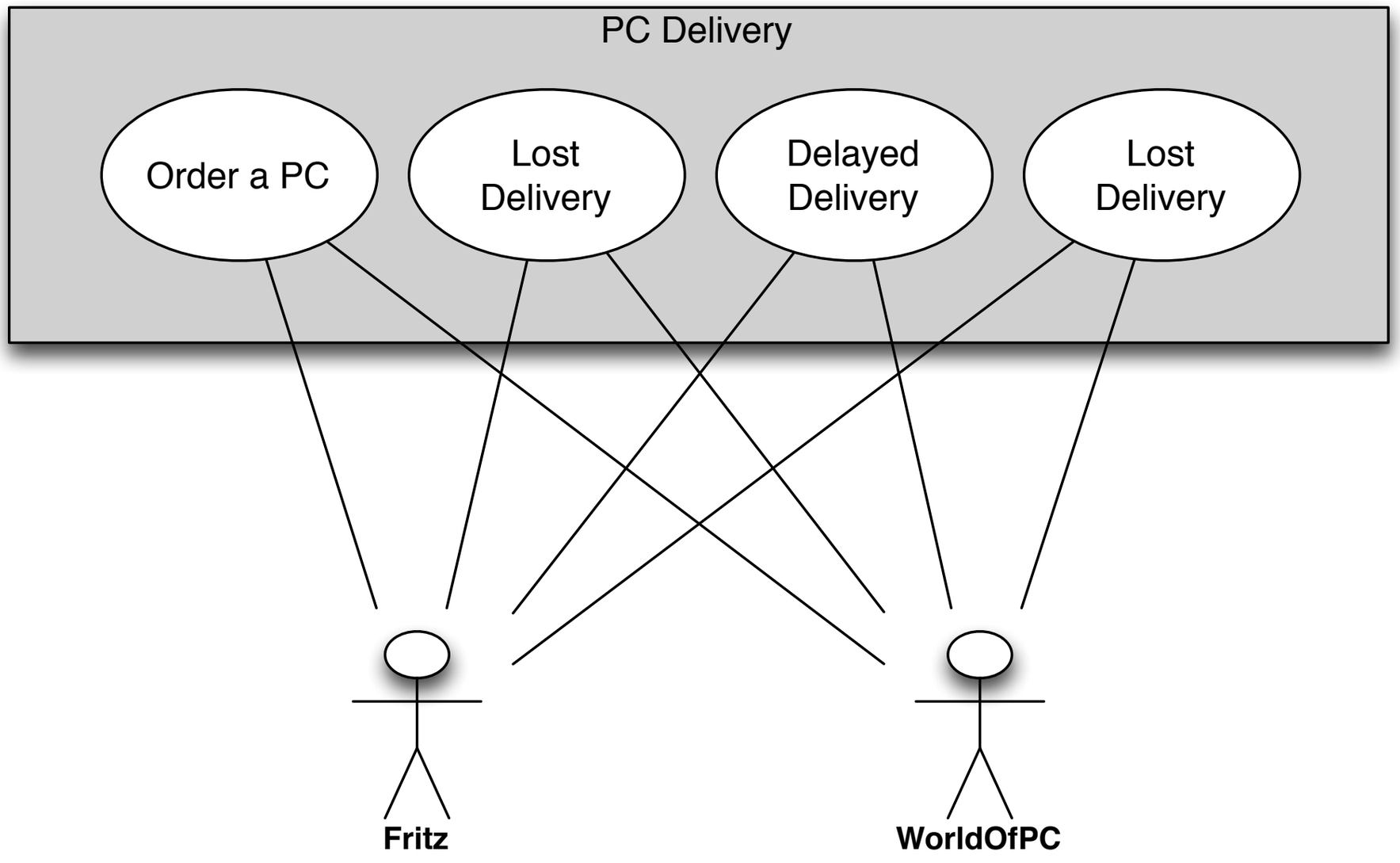
  – contact customer; possibly cancel; –

- Package cannot be delivered

  – contact customer; possibly cancel; –

Use Cases in UML

combine scenarios



Design by Responsibility

Design by responsibility is a common technique:

Each object is responsible for certain tasks and it is either capable of performing them its own, or it has to cooperate with other objects to do so.

The goal is *to devise objects according to their roles in a collaboration*.

Design by Responsibility

Begin with an informal description of the task, and examine its *key phrases*:

Nouns will become *classes* and *concrete objects*.

Verbs will become *services* –

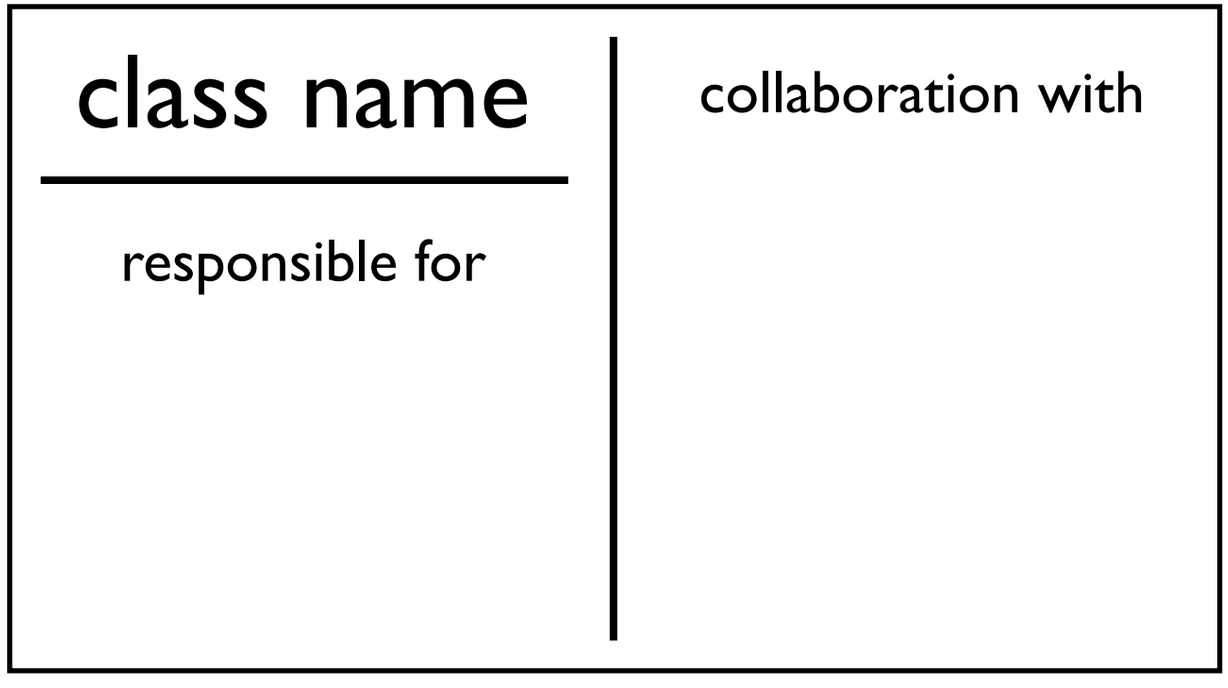
- either services provided by an object,
- or calls to services of cooperating objects.

The *services* determine the responsibilities and collaborations of each class.

Design by Responsibility

The classes that were found this way are then notated onto *CRC-Cards*

(class – responsibilities – collaboration):



The CRC-Card represents the *role* of an object in the global system.

A First Approximation

Student	*collaboration with*
responsible for	
ordering *receiving packages*	computer vendor deliverer

ComputerVendor	*collaboration with*
responsible for	
accepting orders sending pakages	student deliverer

Deliverer	*collaboration with*
responsible for	
receiving packages sending packages	computer vendor student

Refinement

The first approximation, however, is not yet complete:

- Fritz acts in his role as customer; it is not important that he is a student (unless he would get a student's discount). The class name *Customer* is better suited than *Student*.
- Letter and package are missing – these are pure data objects that have neither responsibilities nor collaborations.
- We have left out the way the letter gets to the computer vendor; possibly there is another delivery company involved.
- The *flow of information*, *state transitions* and *class hierarchies* are not taken into account.

Refinement

- We have left out the way the letter gets to the computer vendor; possibly there is another delivery company involved.
- The *flow of information*, *state transitions* and *class hierarchies* are not taken into account.

Revising a Design

The first draft can usually be significantly improved:

- Identifying common features
- Generalizing behavior
- Splitting classes into subsystems
- Minimizing relations
- Using libraries
- Genericity and design patterns

Common Features

Is it possible to connect common features (attributes, methods) of different classes?

These commonalities

- can be relocated into an *aggregate class*. The existing classes still have to offer the transferred services.
- can be moved into a *common superclass*. Usually this makes sense with common is-relationships.

Generalizing Behavior

Is it possible to provide methods with a unified interface on an abstract level?

Abstract classes can provide general methods, the details of which are implemented in the concrete subclasses.

Splitting Classes into Subsystems

Is it possible to split up classes with many features?

Consider introducing a subsystem consisting of multiple objects and affiliated classes.

Minimizing Object Relations

Is it possible to reduce the number of "uses"-relationships by regrouping classes or interfaces?

Only the newly created subsystem has to manage external relations.

Reuse and Libraries

Is it possible to reuse existing classes?

Possibly adapter classes are needed.

Genericity

Is it possible to use generic classes and methods?

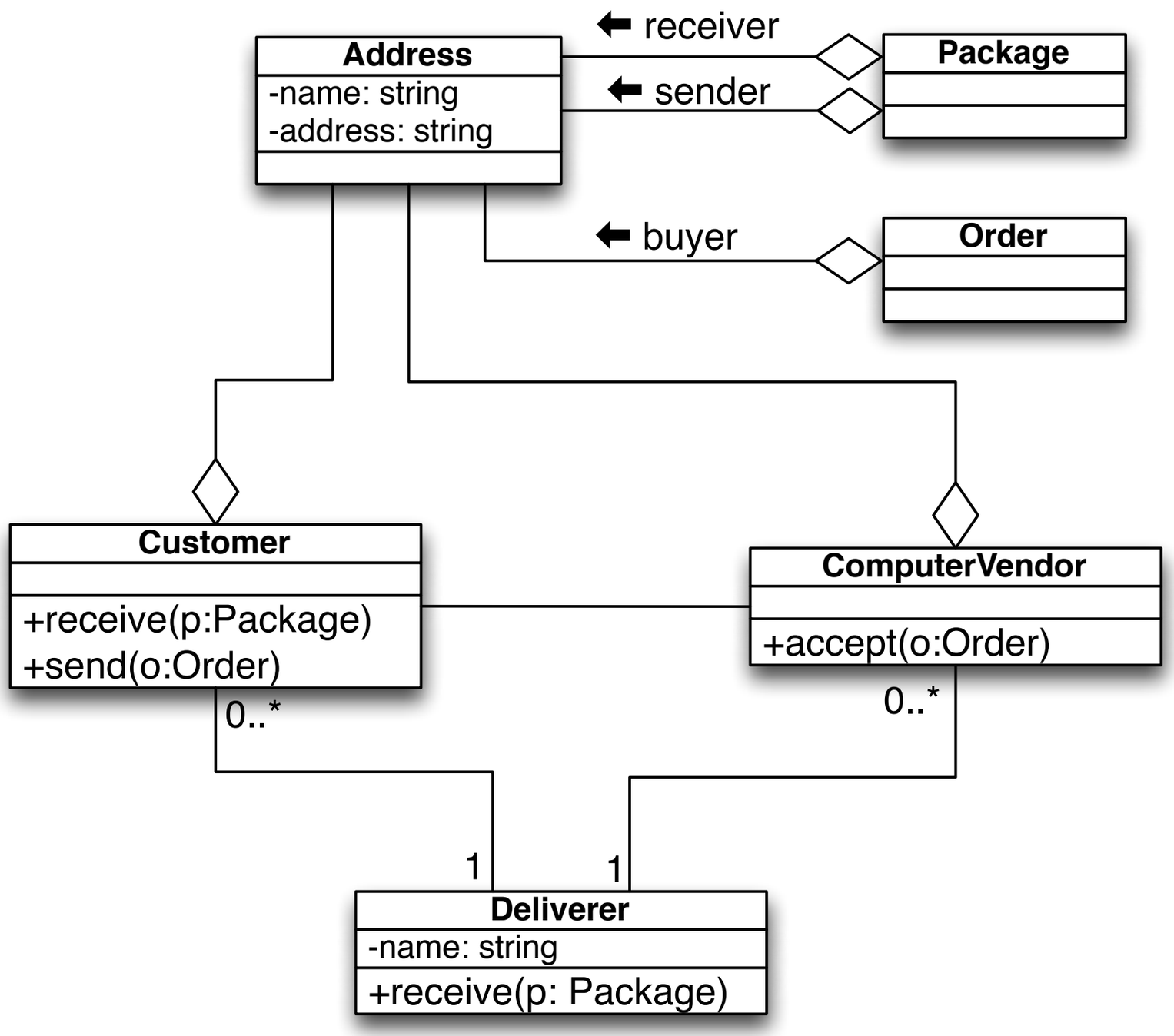
Or maybe: is it possible to design the classes and methods in a generic way?

Design Patterns

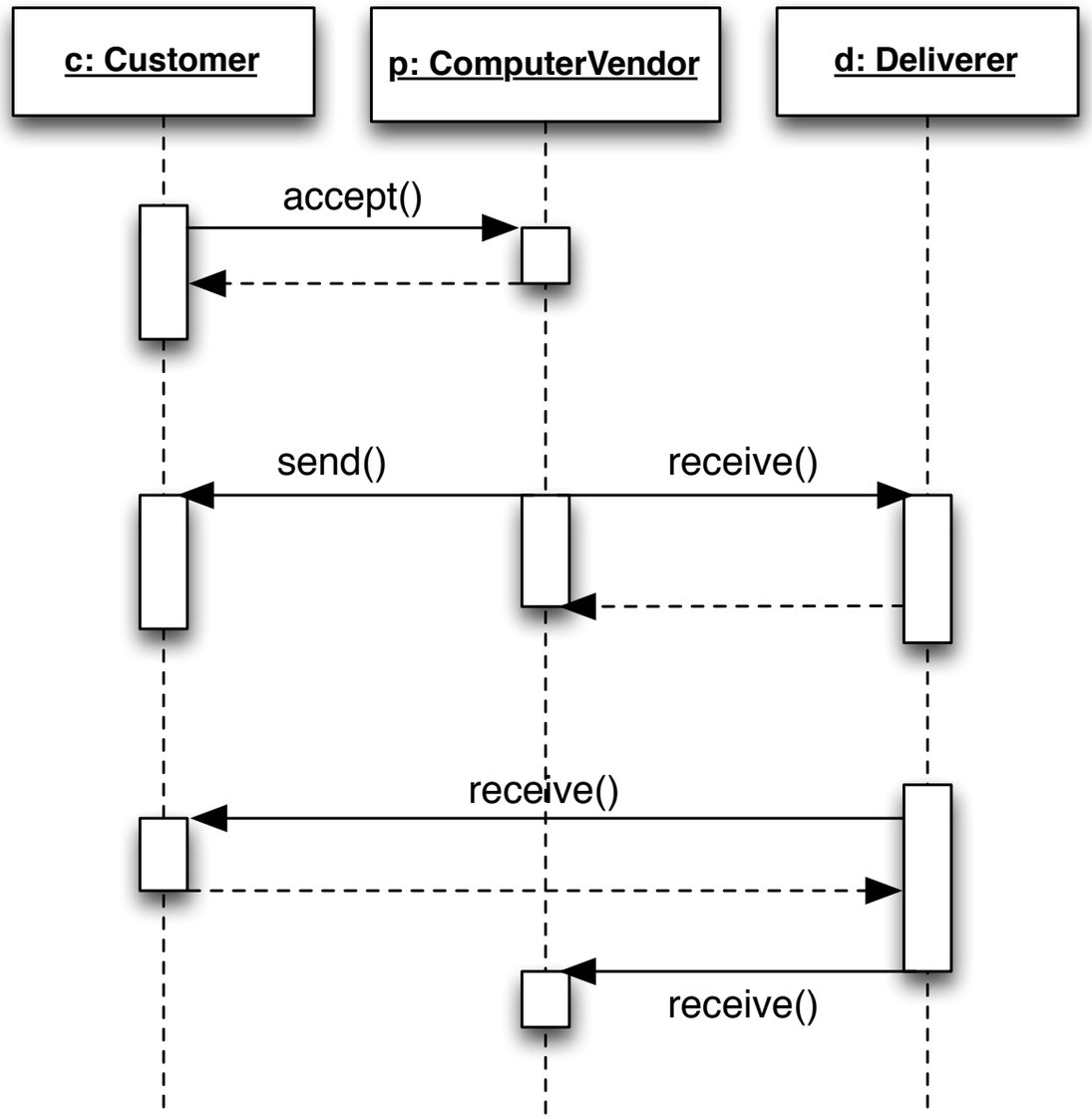
Is it possible to use standard patterns of architectural design?

(more next lecture)

Object-Model: Shipping of a PC



Sequence Diagram: Shipping of a PC

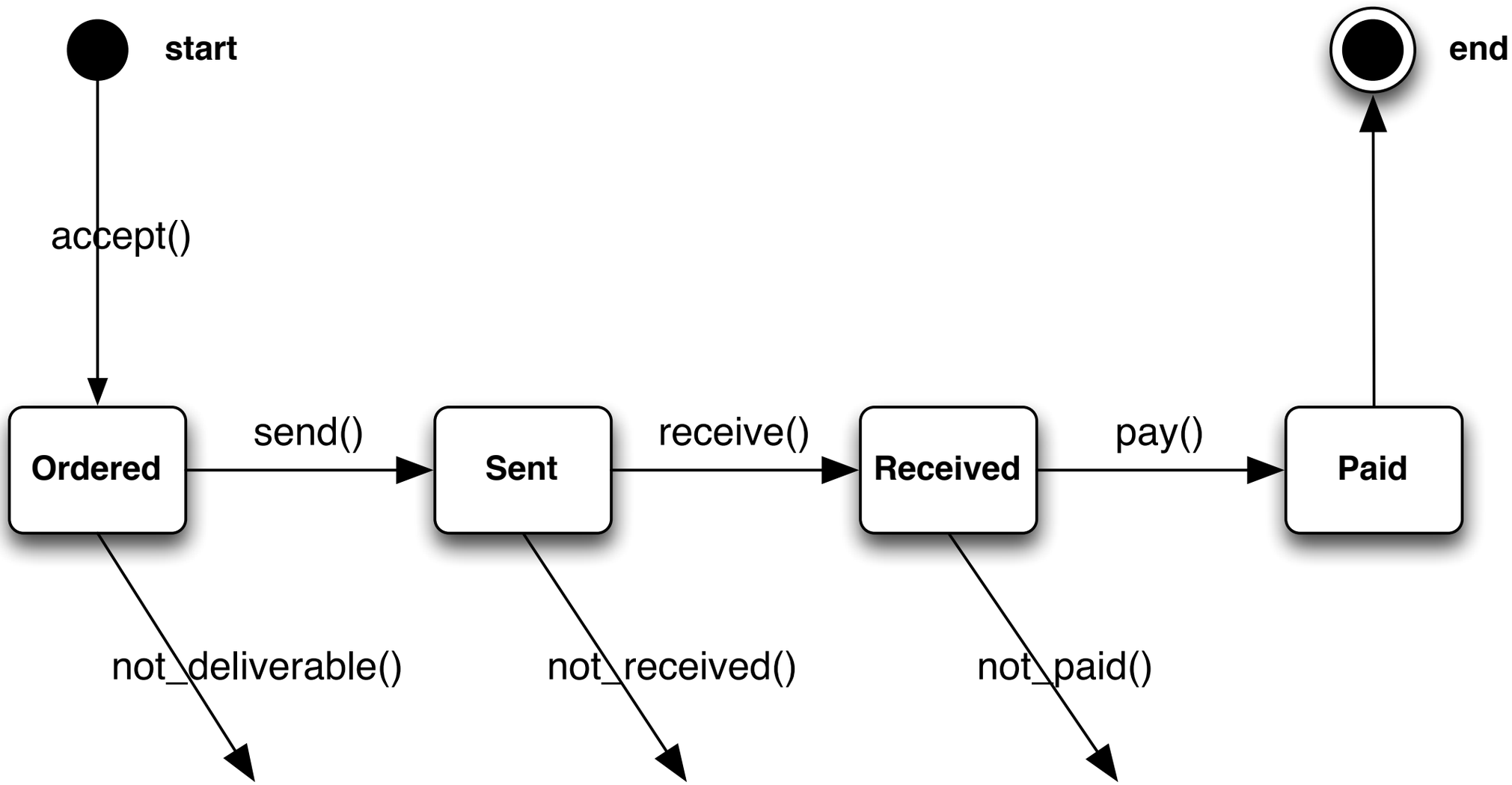


Depicts only the successful case

Failures have to be described separately

State Chart

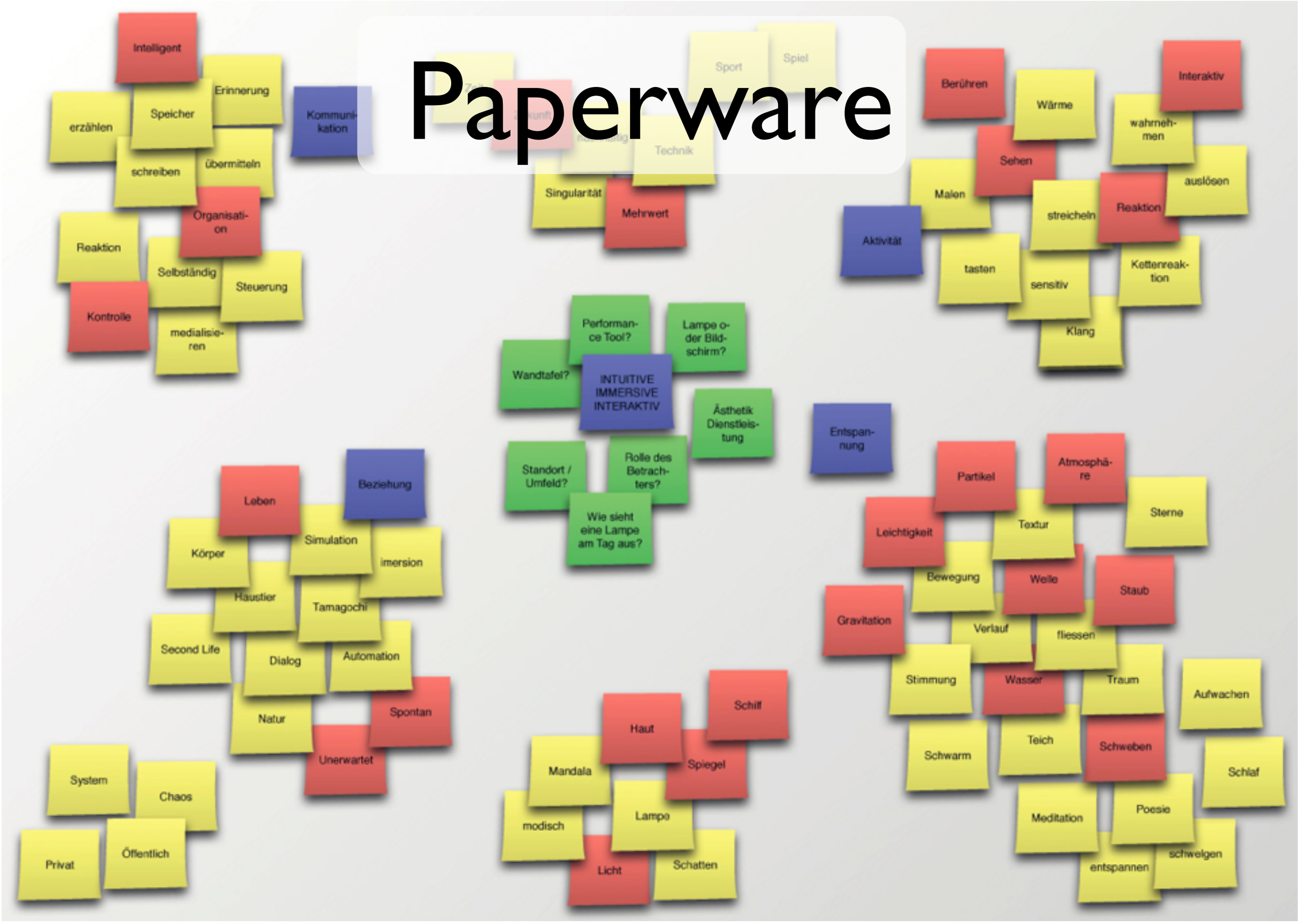
with some few failures



A Word about CRC

"One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code." *(C. Horstmann)*

Paperware


Intelligent
Erinnerung
erzählen
Speicher
Kommuni-
kation
schreiben
übermitteln
Organisati-
on
Reaktion
Selbständig
Steuerung
Kontrolle
medialisie-
ren
Sport
Spiel
Technik
Singularität
Mehrwert
Berühren
Wärme
Interaktiv
wahrneh-
men
Sehen
Malen
auslösen
streicheln
Reaktion
Aktivität
tasten
sensitiv
Kettenreak-
tion
Klang
Performan-
ce Tool?
Lampe o-
der Bild-
schirm?
Wandtafel?
INTUITIVE
IMMERSIVE
INTERAKTIV
Ästhetik
Dienstleis-
tung
Entspan-
nung
Standort /
Umfeld?
Rolle des
Betrach-
ters?
Wie sieht
eine Lampe
am Tag aus?
Beziehung
Leben
Körper
Simulation
imersion
Haustier
Tamagochi
Second Life
Dialog
Automation
Natur
Spontan
Unerwartet
System
Chaos
Privat
Öffentlich
Partikel
Atmosphä-
re
Sterne
Leichtigkeit
Textur
Bewegung
Welle
Staub
Gravitation
Verlauf
fliessen
Stimmung
Wasser
Traum
Aufwachen
Schwarm
Teich
Schweben
Schlaf
Meditation
Poesie
entspannen
schwelgen
Schilf
Haut
Spiegel
Mandala
modisch
Lampe
Licht
Schatten

From Model to Program

How does one transform a design into code?

- *Classes* and *hierarchies* can be taken directly from the class diagram.
- For each method a complete signature has to be provided.

From Model to Program

- *Associations between classes* are implemented using attributes.
  - *n*:1 and 1:1 associations from P to Q are implemented as an attribute *q* of type Q in P.
  - 1:*n* and *n*:*m* associations from P to Q are implemented as a set *qs* of type set(Q). (e.g. array, list...)

From Model to Program

- Methods belonging to associations have to be implemented in extra (helper) classes.
- For each class an invariant has to be formulated and documented; for each method a pre- and postcondition.
- The method bodies have to be implemented using techniques from traditional programming.

From Model to Program

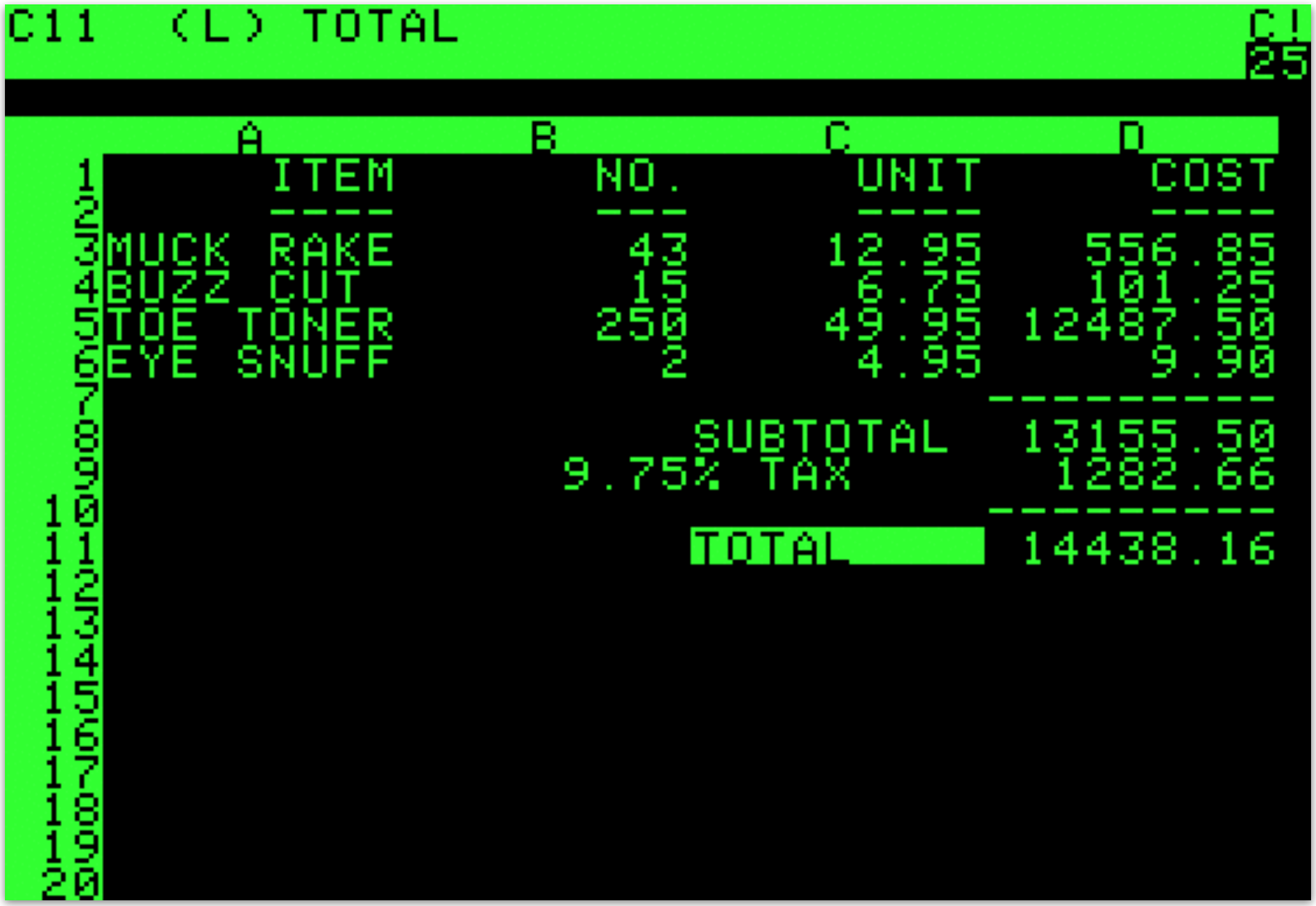
- Verification of the state chart: are the only legal call sequences exactly those documented in the dynamic model?
  Illegal calls have to be intercepted using exceptions/errors! For that it is often useful to dynamically check the precondition.
- Testing is done using conventional methods.

Model-Driven Engineering

Modern programming environments automatically create *code templates* from a model:

1. You design a system with all its classes and attributes.
2. The programming environment creates the corresponding code templates.
3. Now you "only" have to add implementations in the method bodies.

Case Study: Spreadsheet



The VisiCalc spreadsheet program – the first "killer app"

Case Study: Spreadsheet

A *spreadsheet* consists of *m x n* cells.

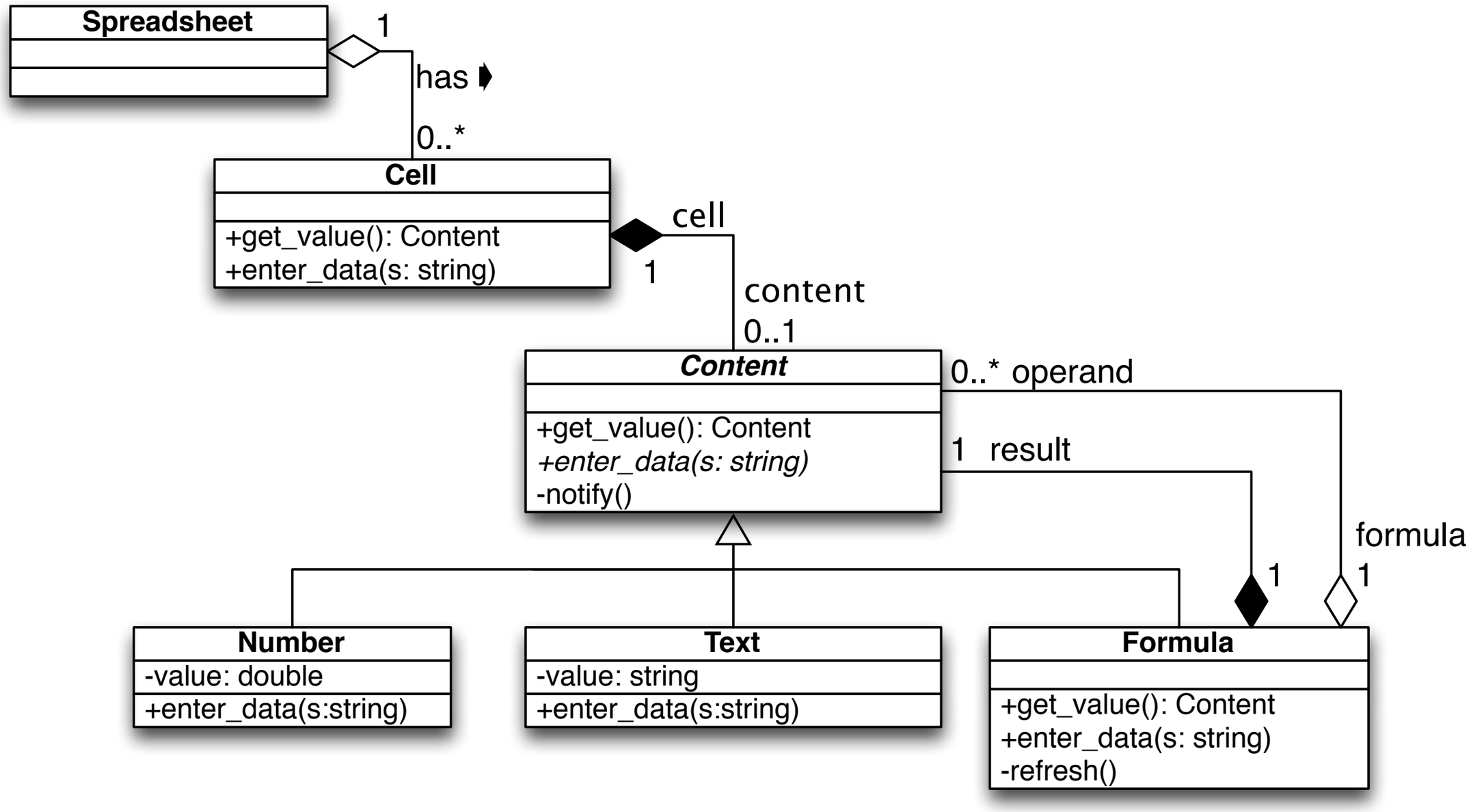
Cells are either empty or they have *content*.

Contents can be *numbers*, *texts*, or *formulas*.

There are multiple formulas for a content (that reference the content)

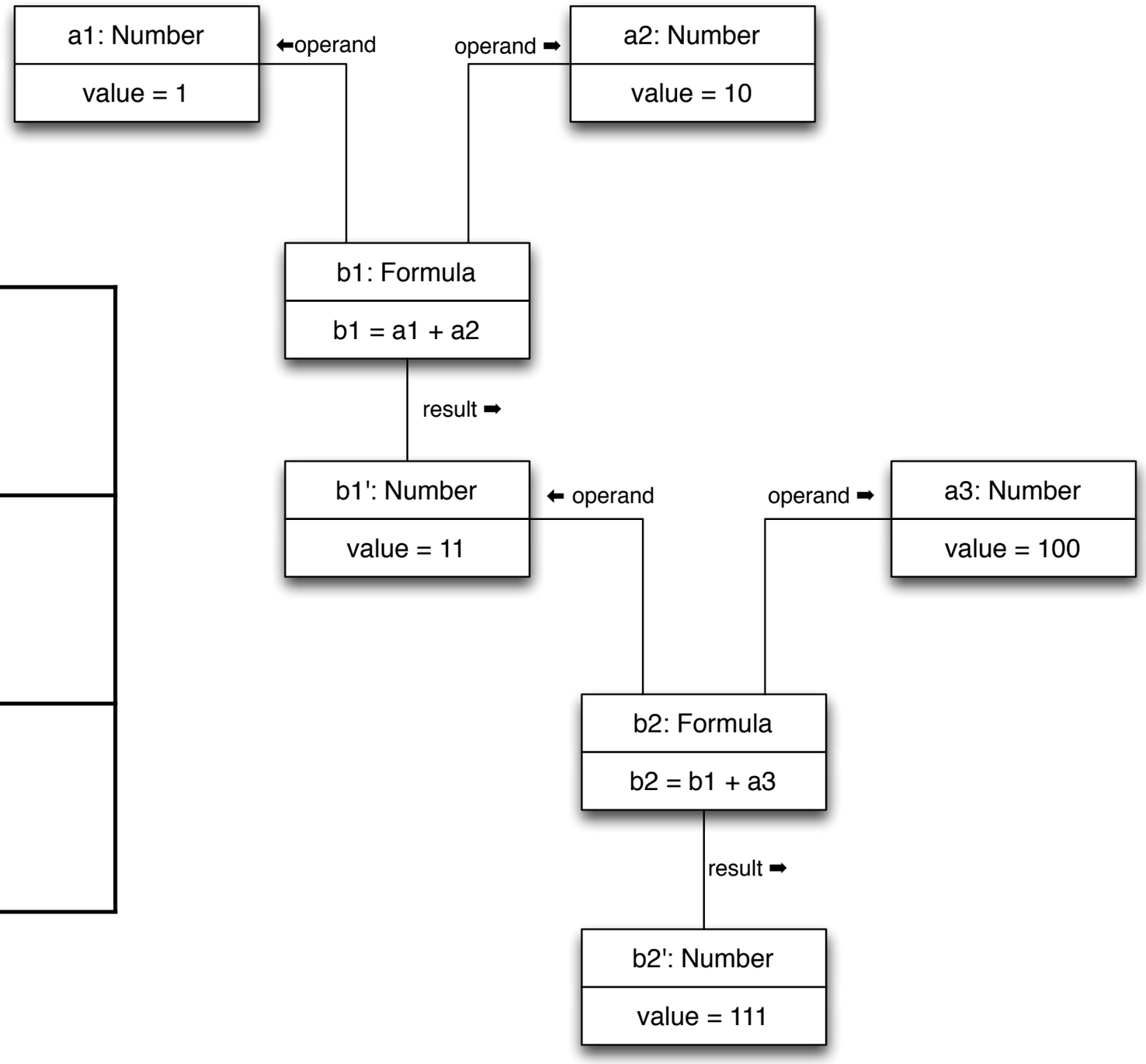
There are multiple contents for a formula (that serve as operands)

Object Model



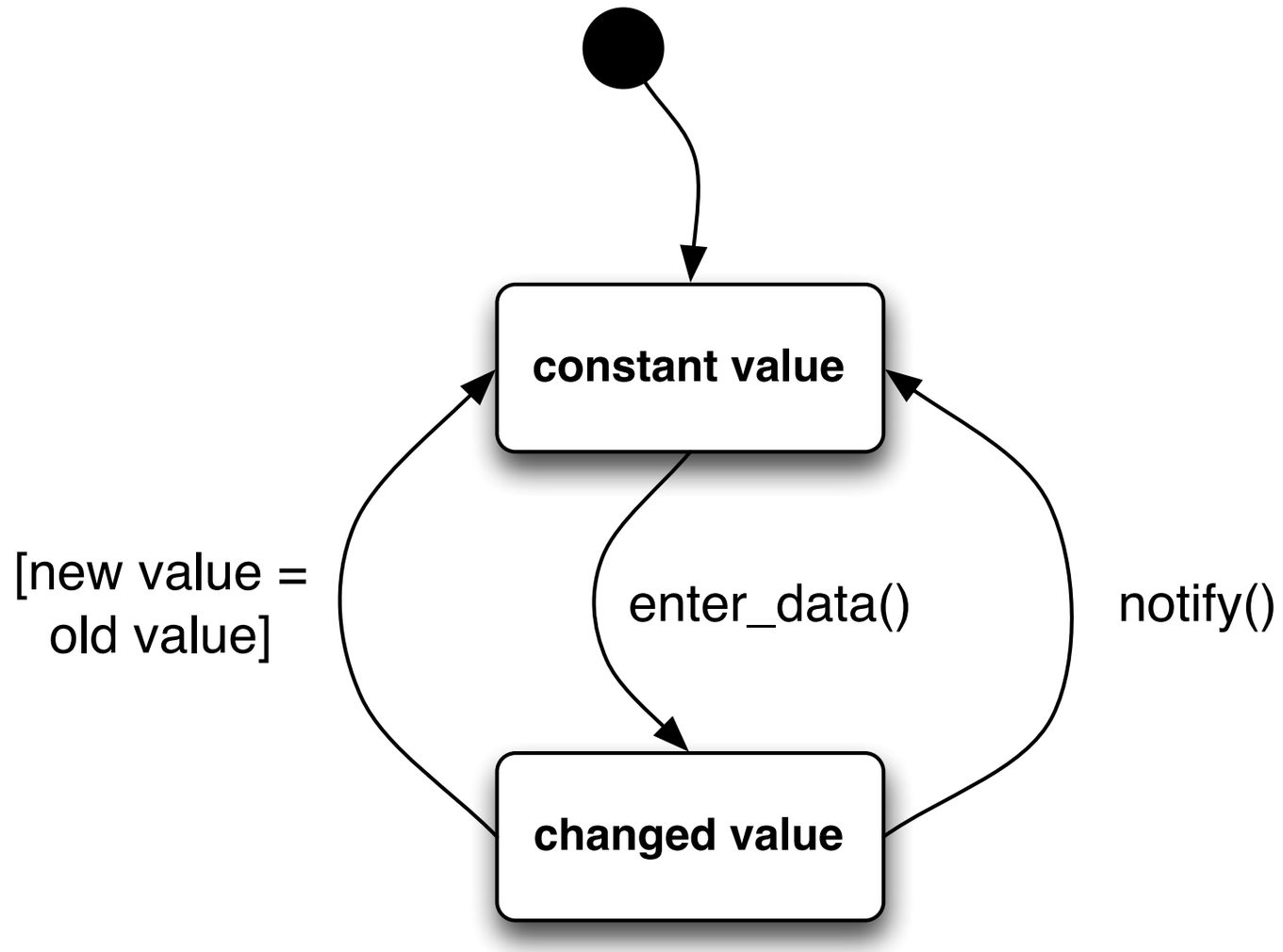
Relationships between Objects

	A	B
1	1	11
2	10	111
3	100	



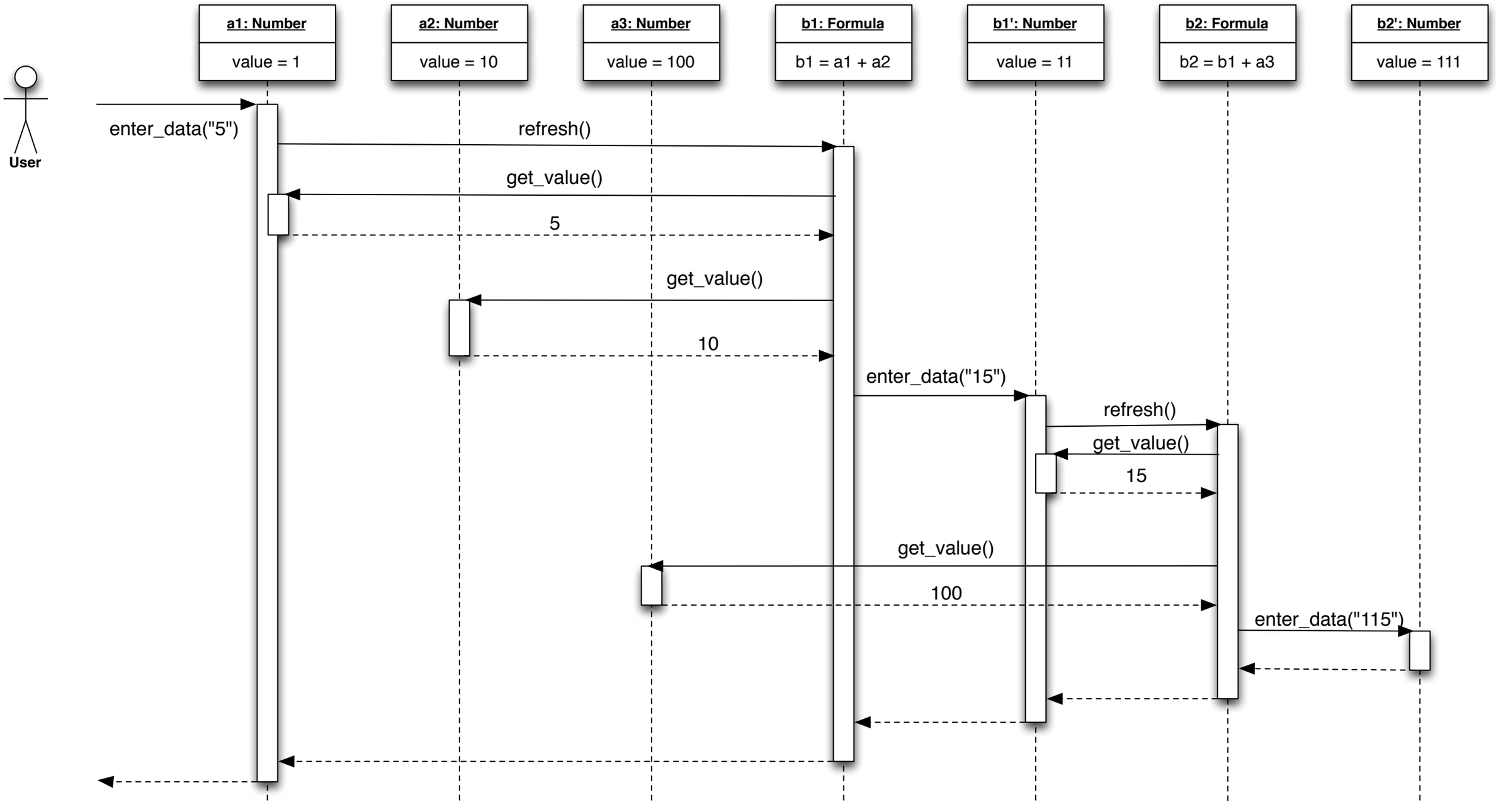
State Chart

The method *enter_data*() of the *Content* class examines whether the actual value has changed. If so, every *Formula* that has this *Content* as an operand is notified by means of the method *notify*().



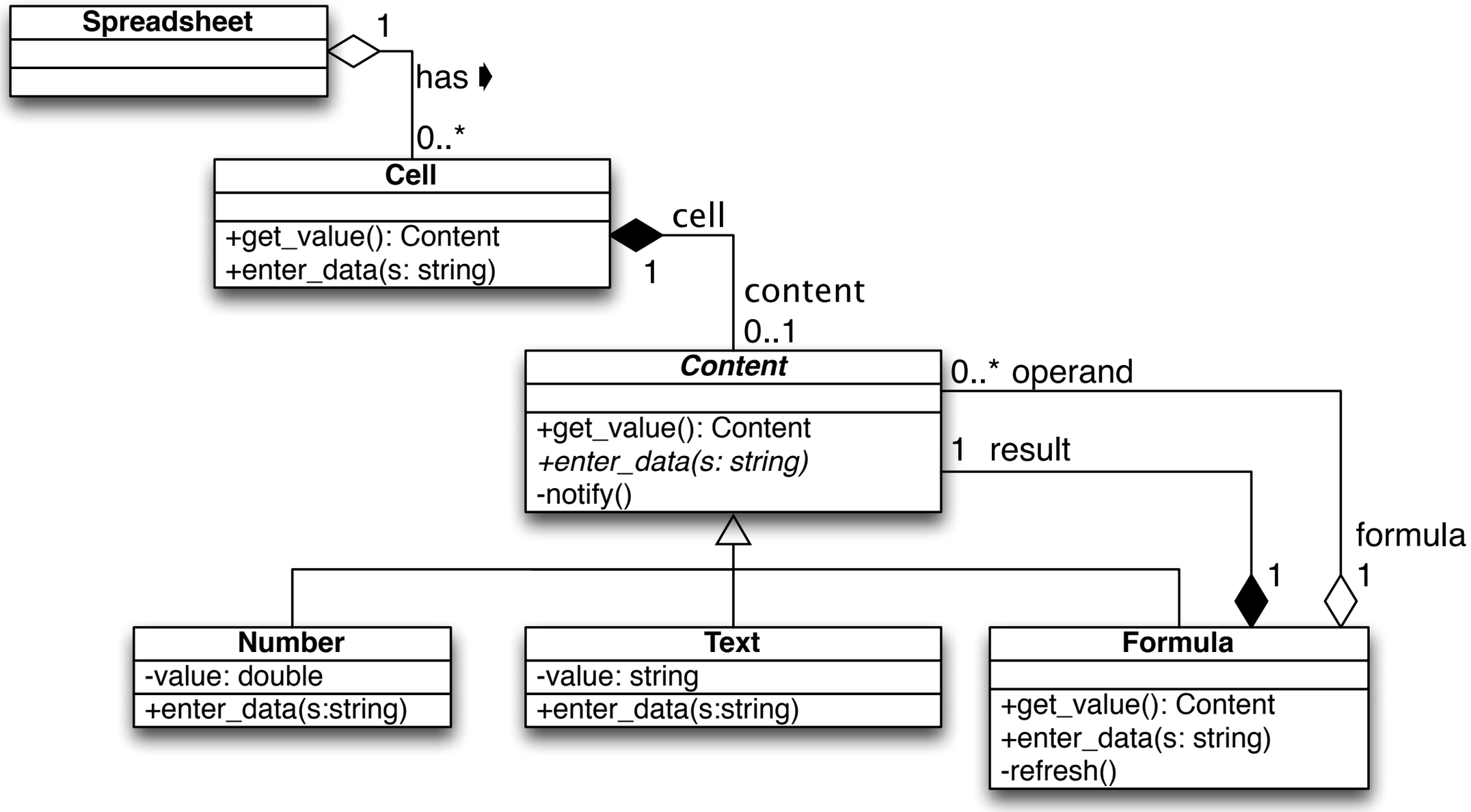
Sequence Diagram

Example: Let the spreadsheet be filled out as just described; now the value of cell A1 is changed from 1 to 5.



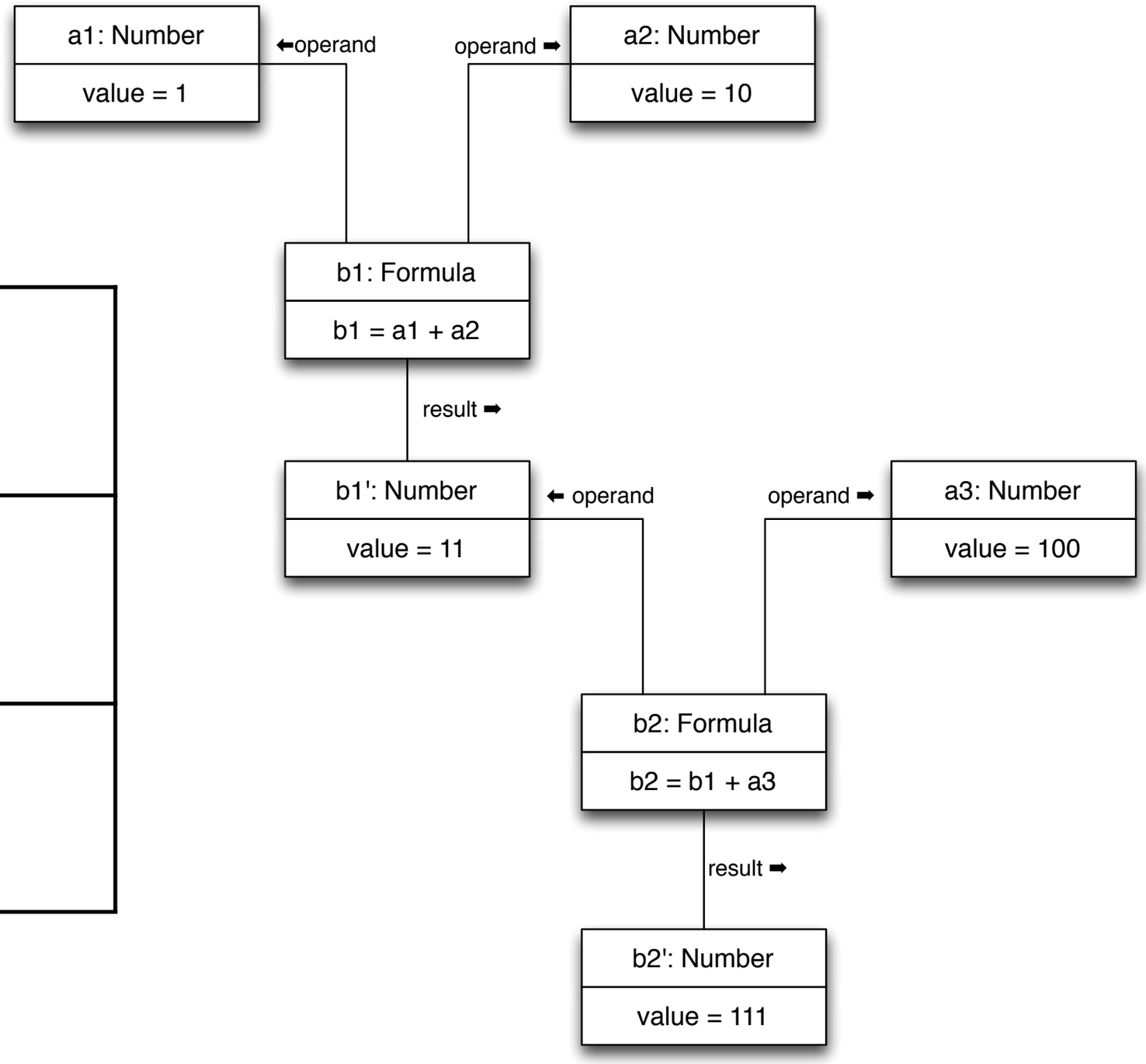
Handouts

Object Model



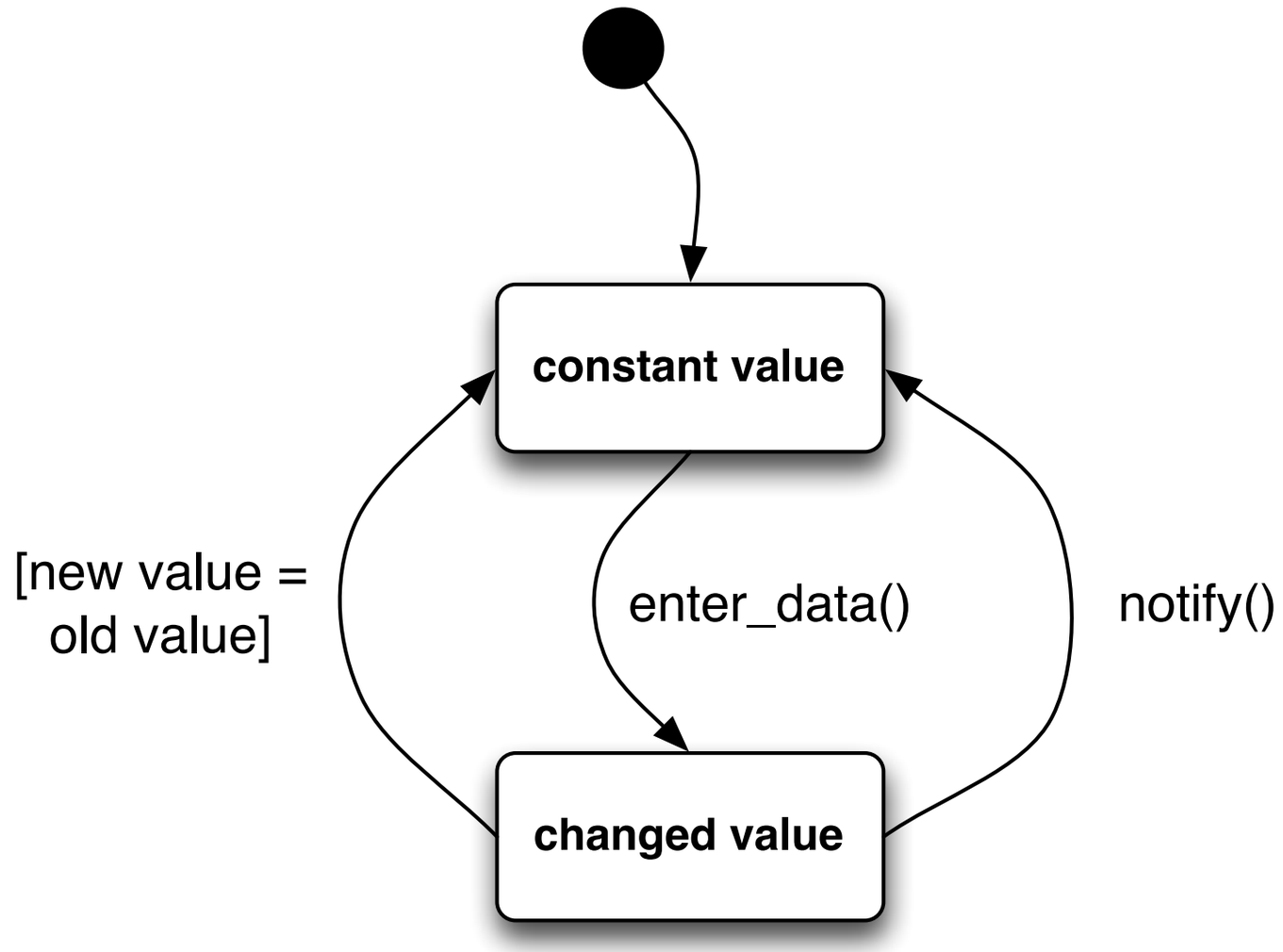
Relationships between Objects

	A	B
1	1	11
2	10	111
3	100	



State Chart

The method *enter_data*() of the *Content* class examines whether the actual value has changed. If so, every *Formula* that has this *Content* as an operand is notified by means of the method *notify*().



Sequence Diagram

Example: Let the spreadsheet be filled out as just described; now the value of cell A1 is changed from 1 to 5.