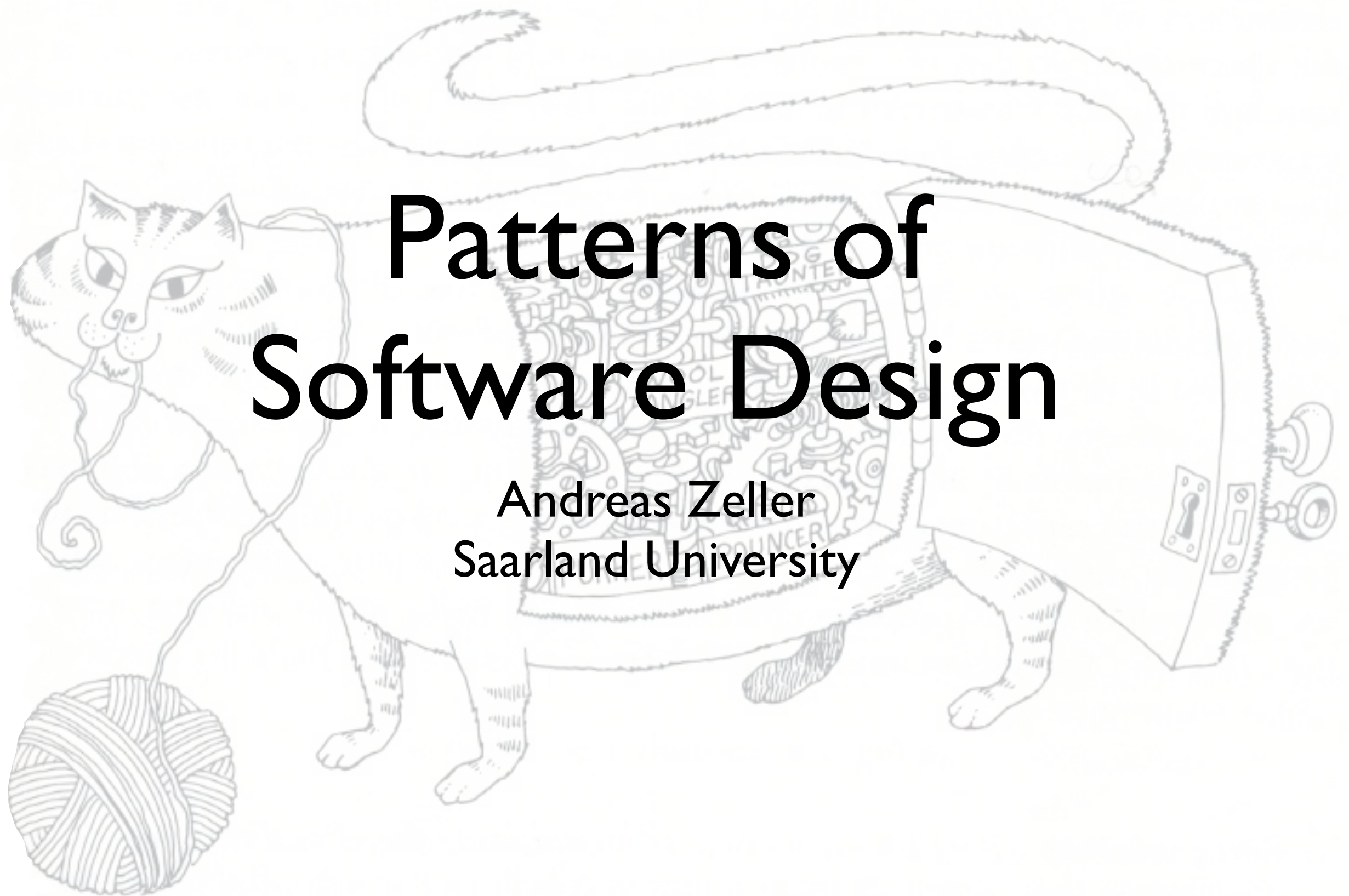


# Patterns of Software Design

Andreas Zeller  
Saarland University



# 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?*



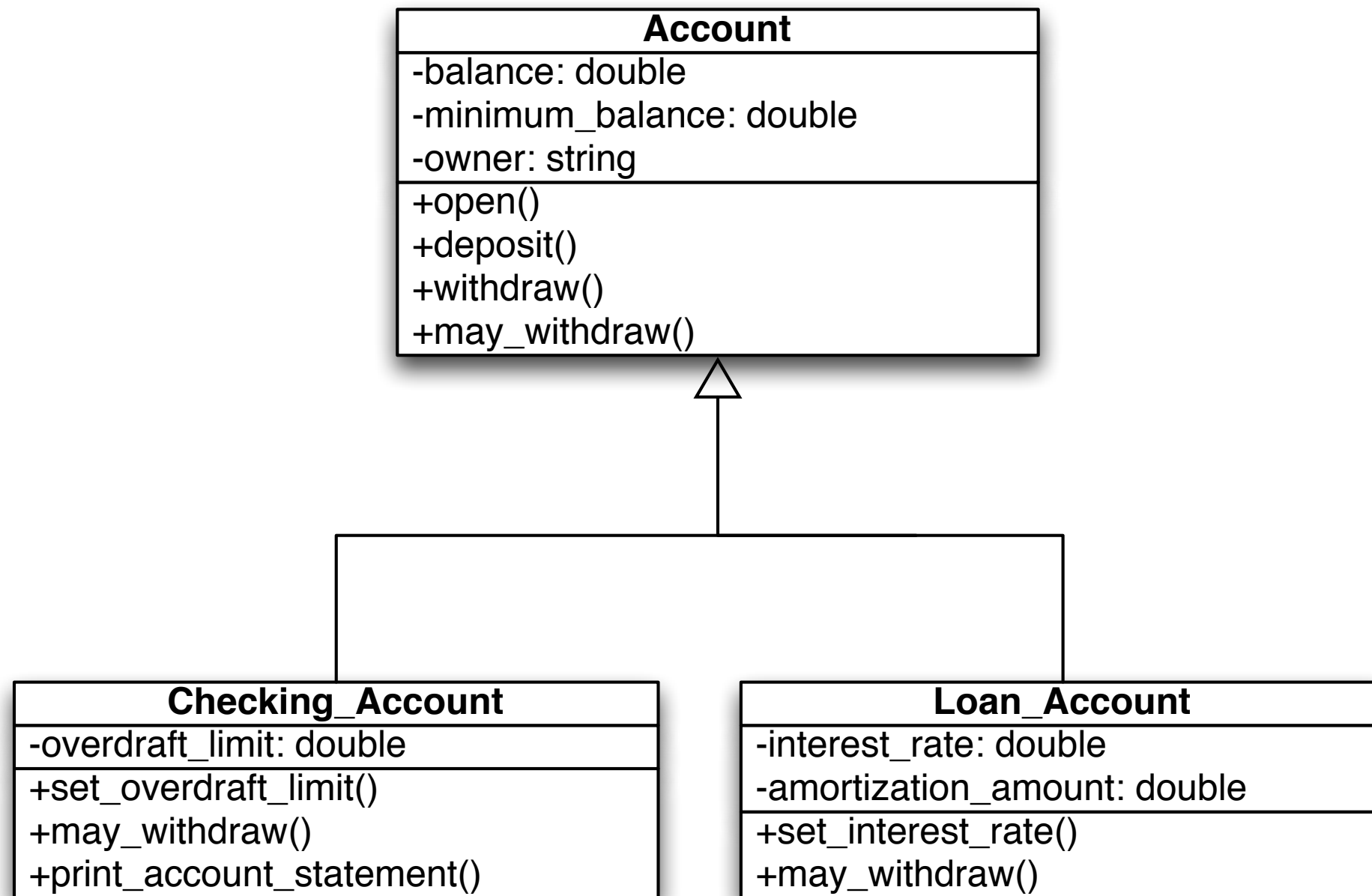
# Object-Model: Class Diagram

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

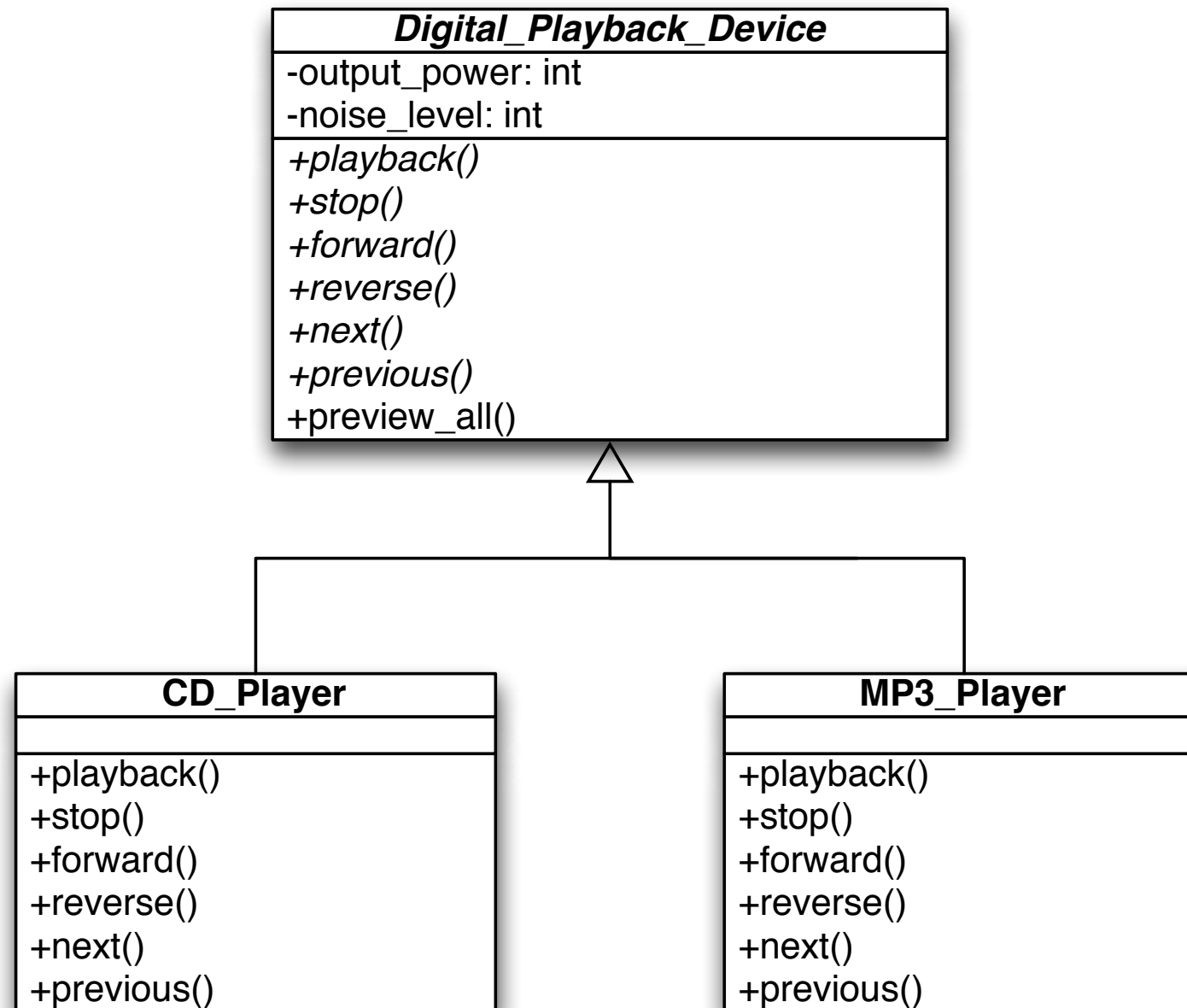
*Abstract classes* cannot exist as concrete objects(instances).

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*.

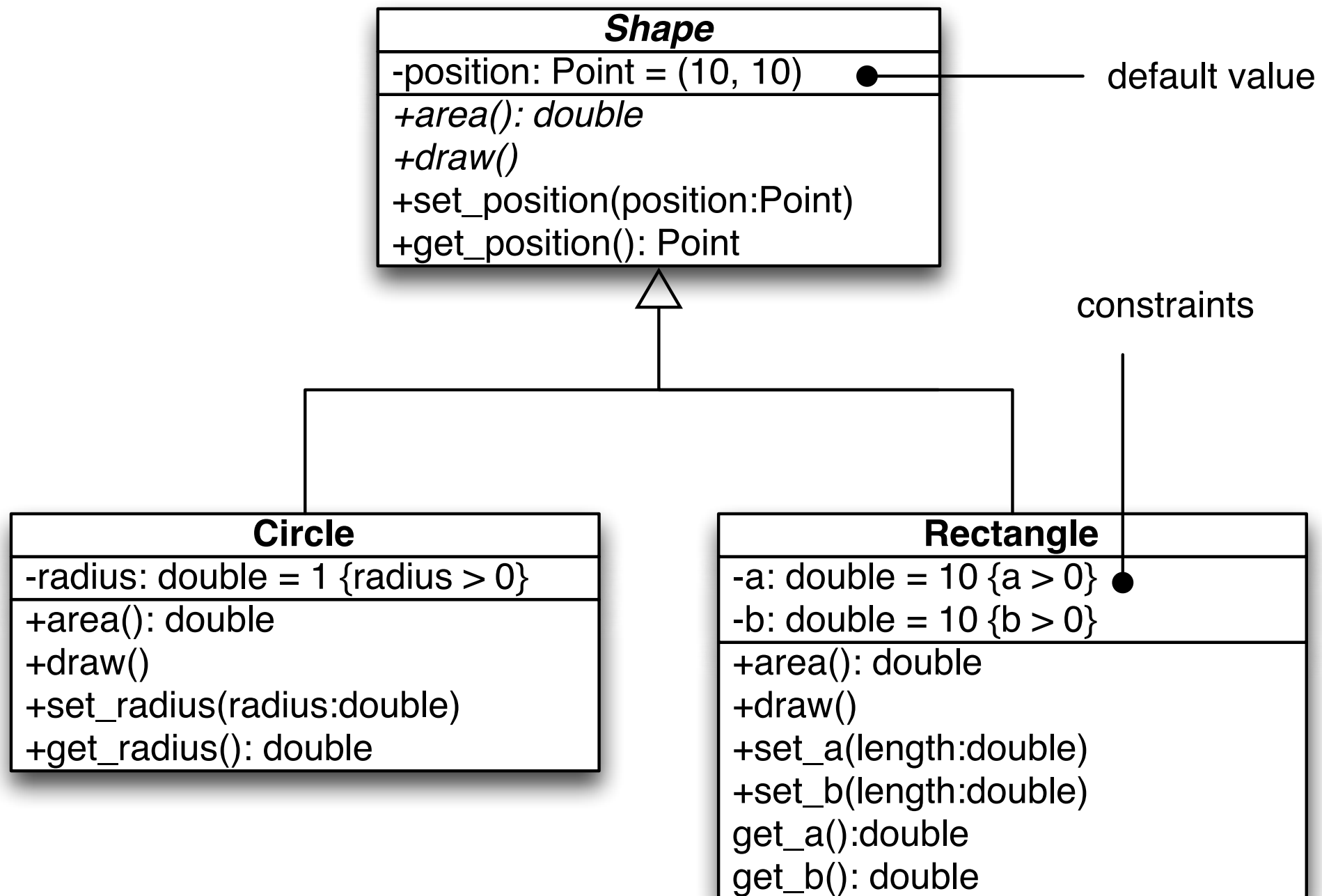
These will be used by *default* if nothing is specified upon construction.

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.



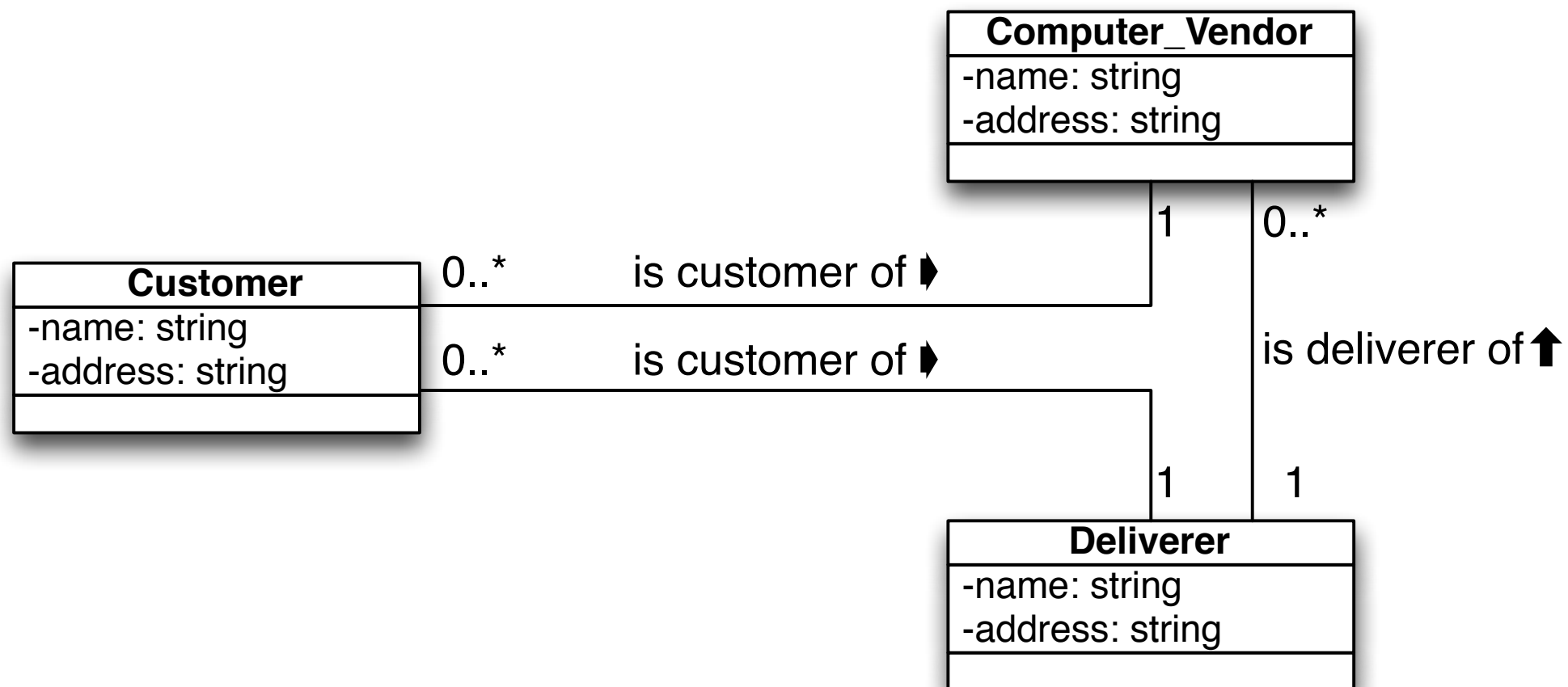
# Object-Model: Associations

## *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

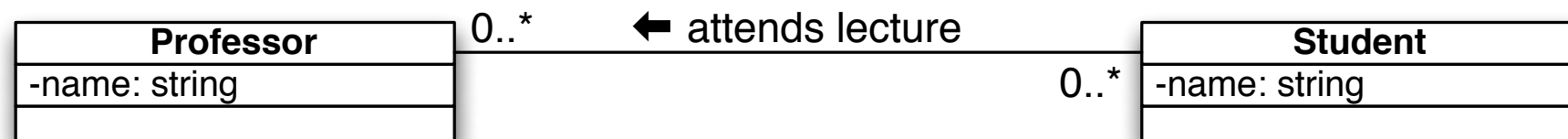
A *computer vendor* has multiple *customers*, a *delivery agency* also has multiple customers, but the computer vendor has only one delivery agency.



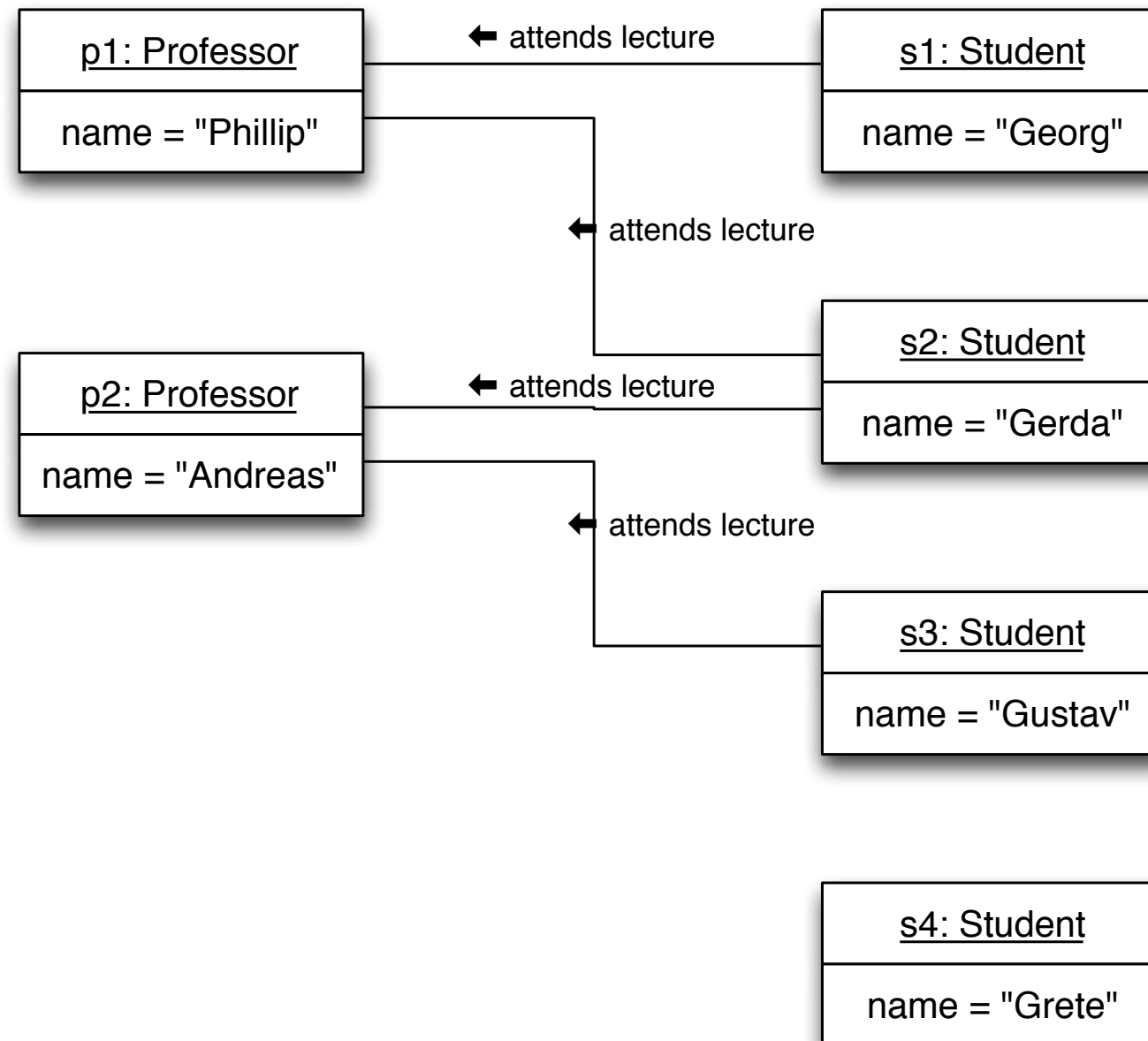


# Example: Multiplicity (2)

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



# Example: Relationships between Objects



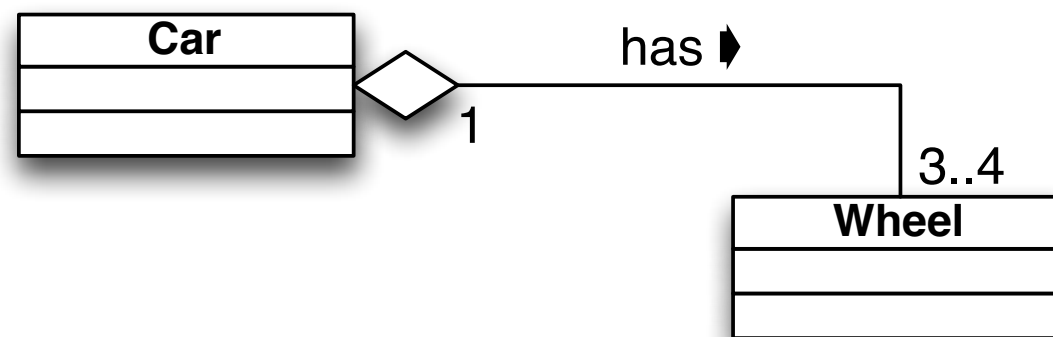
Underlined names indicate concrete objects(instances), which have concrete values for their attributes.

# 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



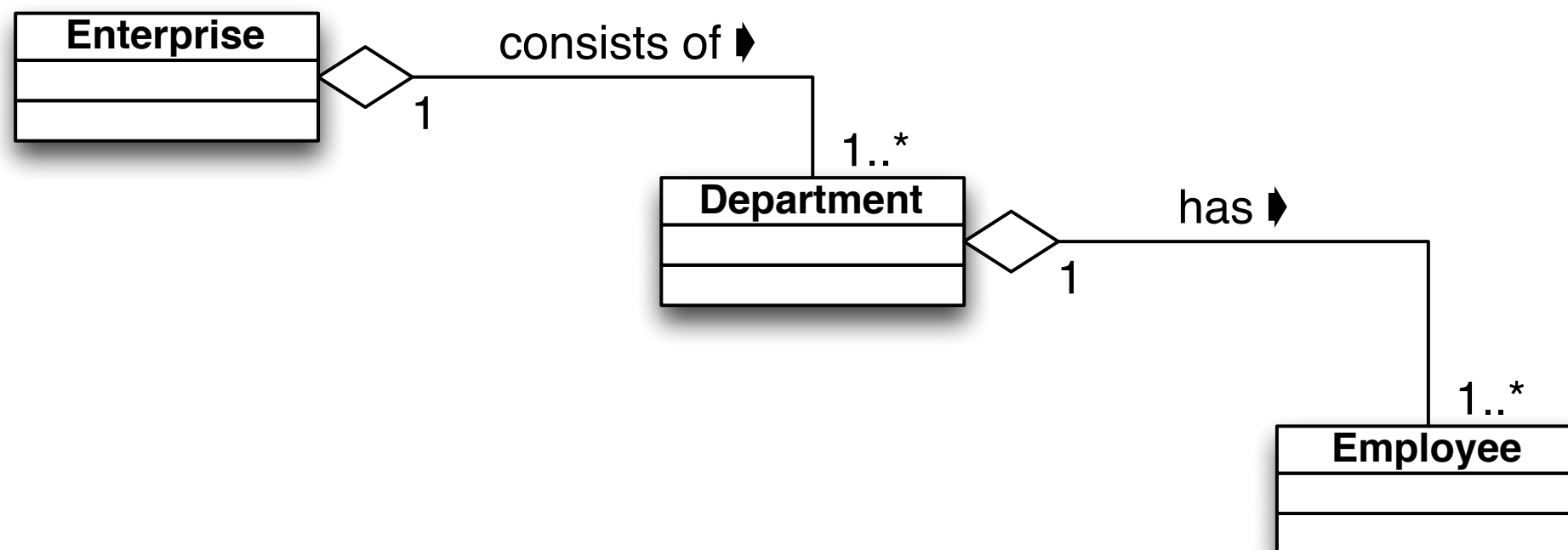


# A Car



# 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

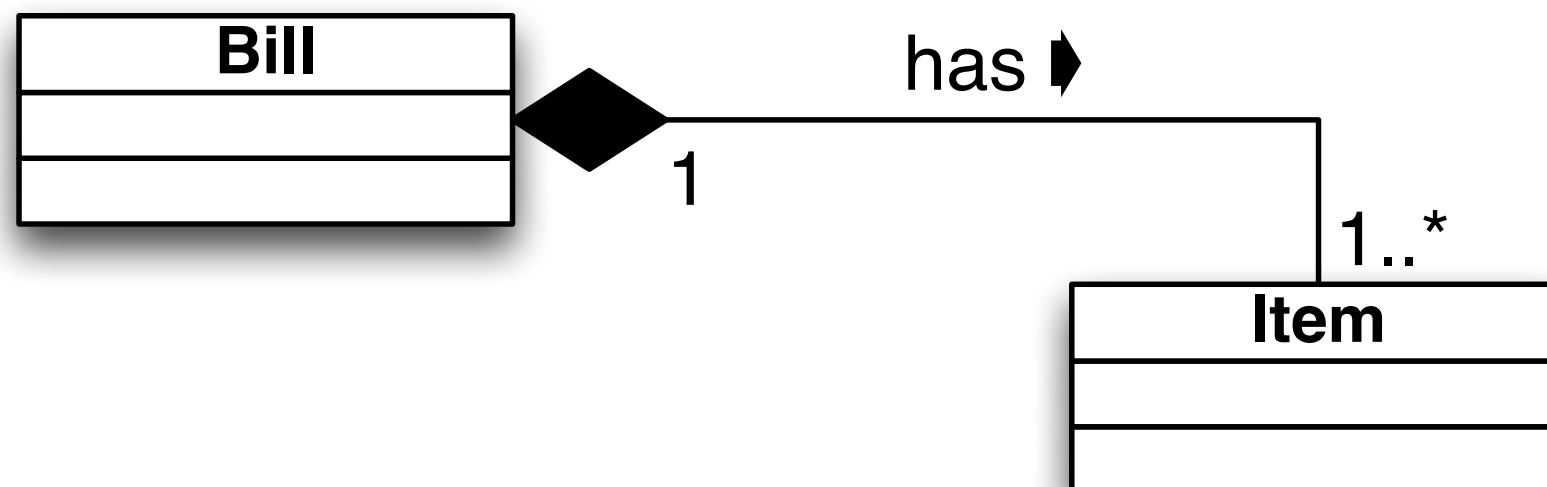
A special case of the aggregation, the *composition*, is marked with ♦

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



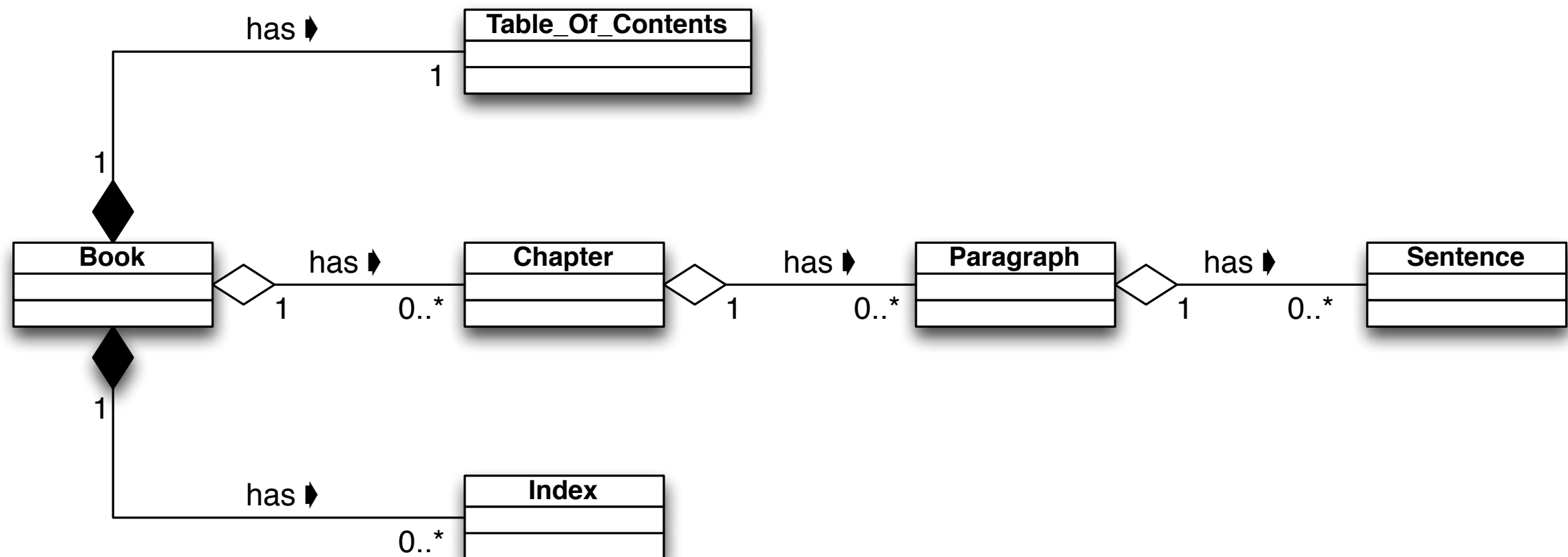
# Example: Bill Item

A bill item always belongs to a bill.



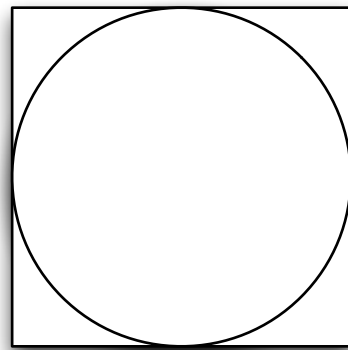
# Example: Book

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



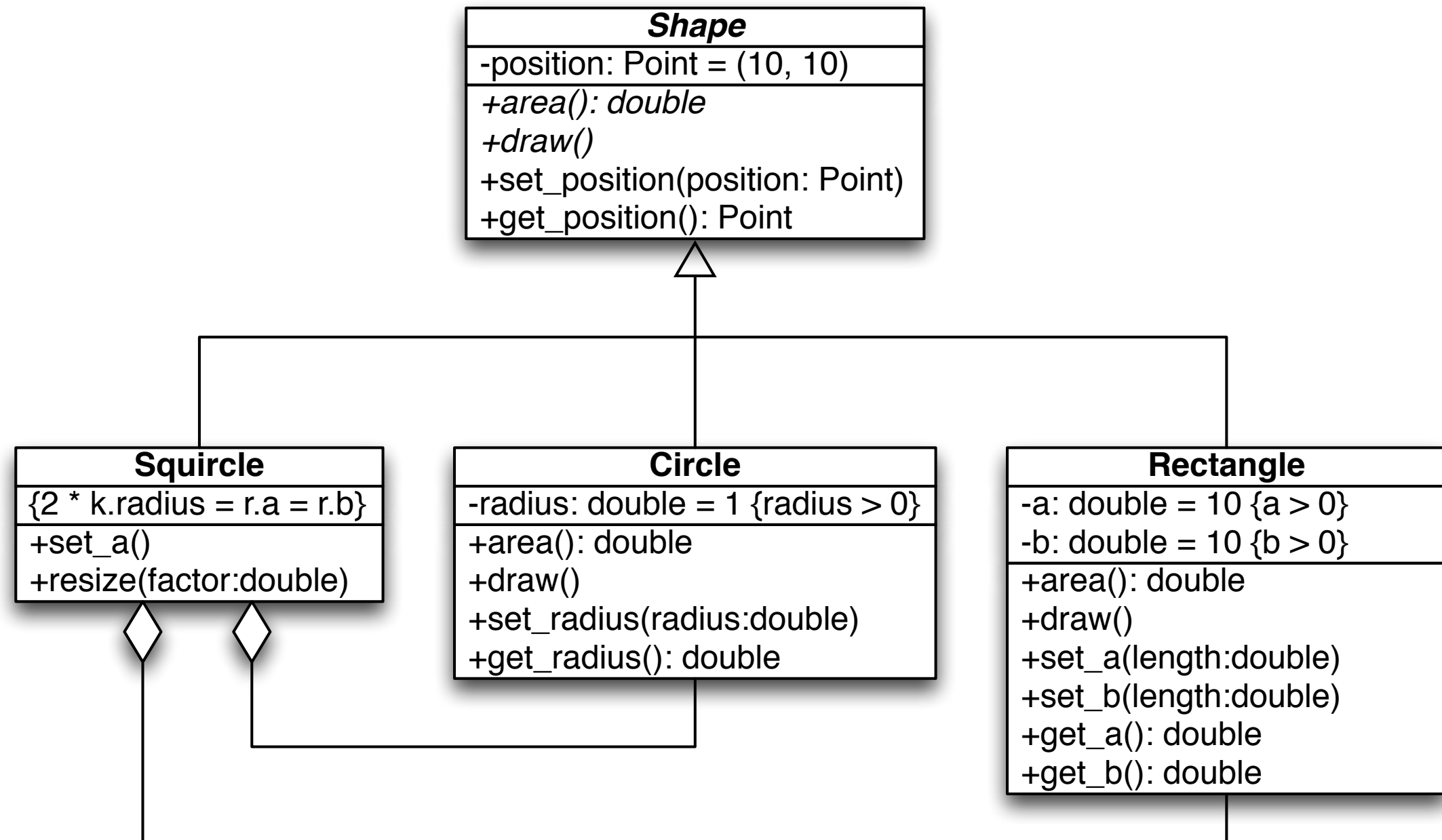
# Example: Squircle

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

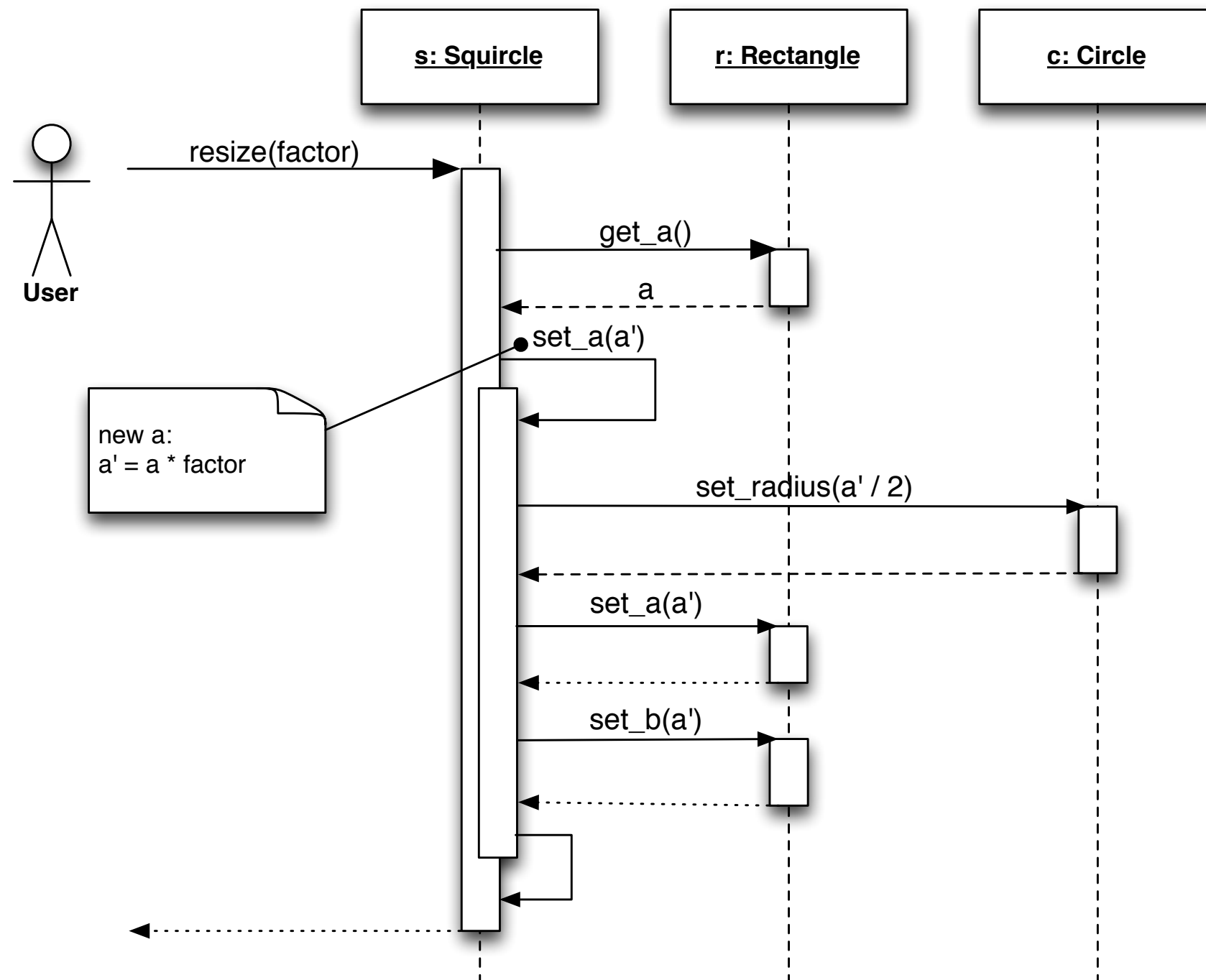
A *sequence diagram* reflects the flow of information between individual objects with an emphasis on *chronological order*.

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

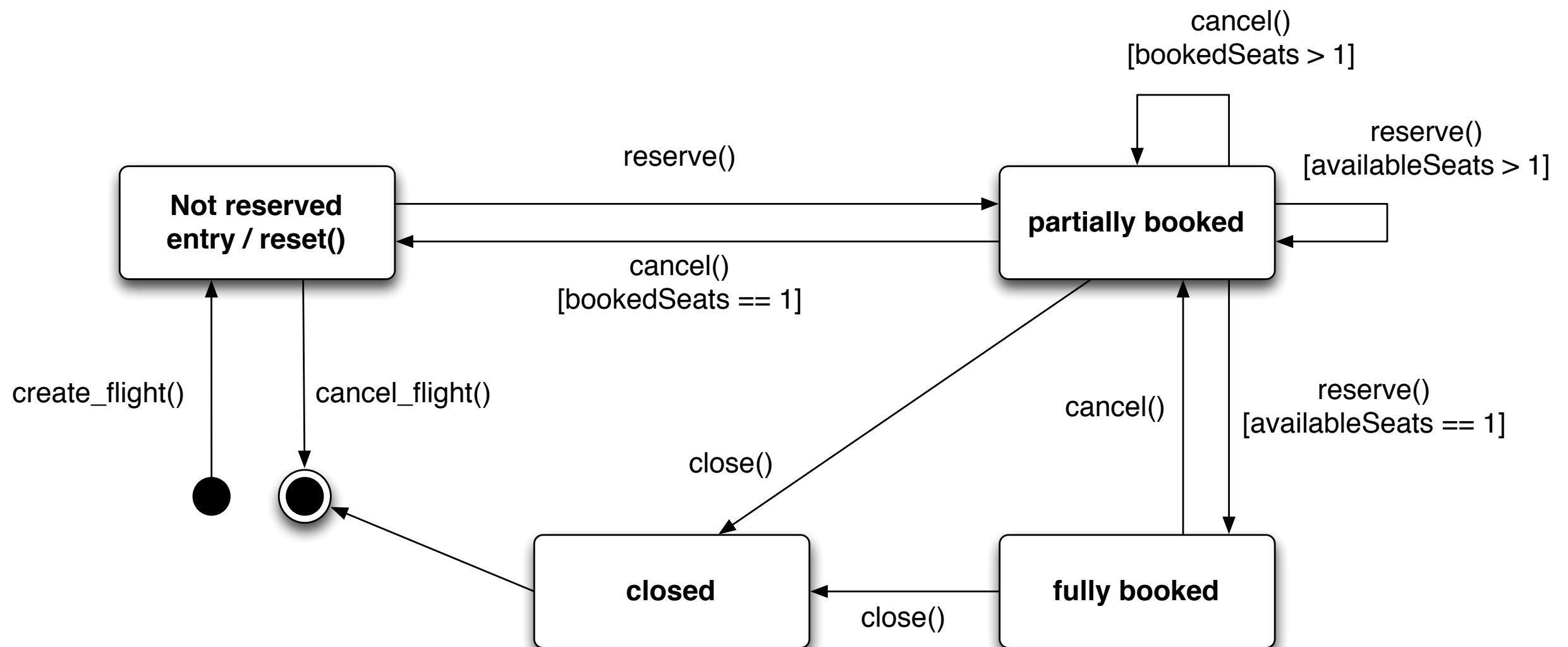
States can also be annotated with *actions*:

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.



# Case Study: Spreadsheet

C11 (L) TOTAL C125

	A	B	C	D
	ITEM	NO.	UNIT	COST
	---	---	---	---
	MUCK RAKE	43	12.95	556.85
	BUCKZ CUT	15	6.75	101.25
	TONER	250	49.95	12487.50
	EYEF SNUFF	2	4.95	9.90
				-----
			SUBTOTAL	13155.50
		9.75% TAX		1282.66
				-----
			TOTAL	14438.16

The VisiCalc spreadsheet program – the first “killer app”

# Case Study: Spreadsheet

*A spreadsheet consists of  $m \times n$  cells.*

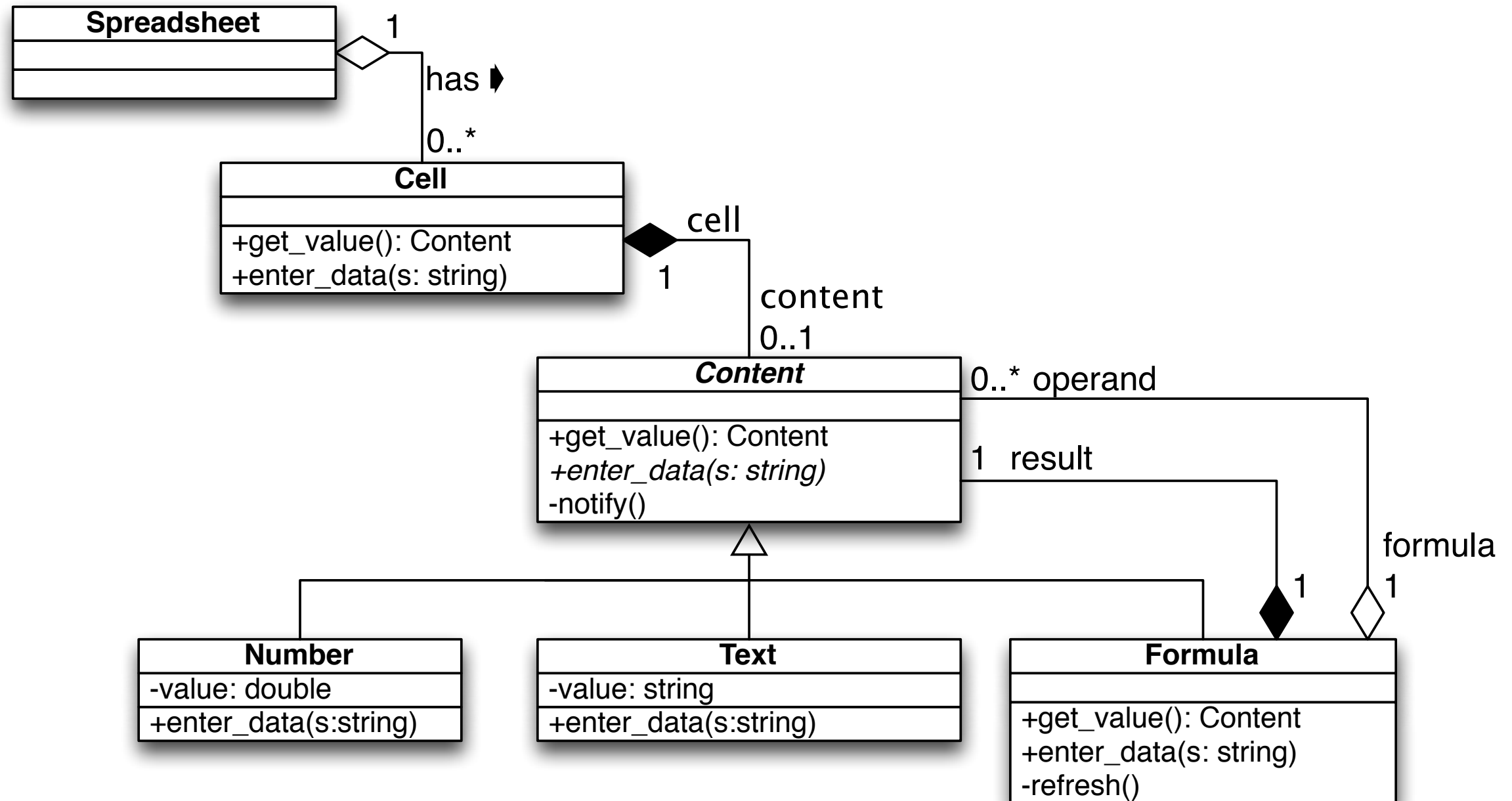
*Cells are either empty or they have content.*

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

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

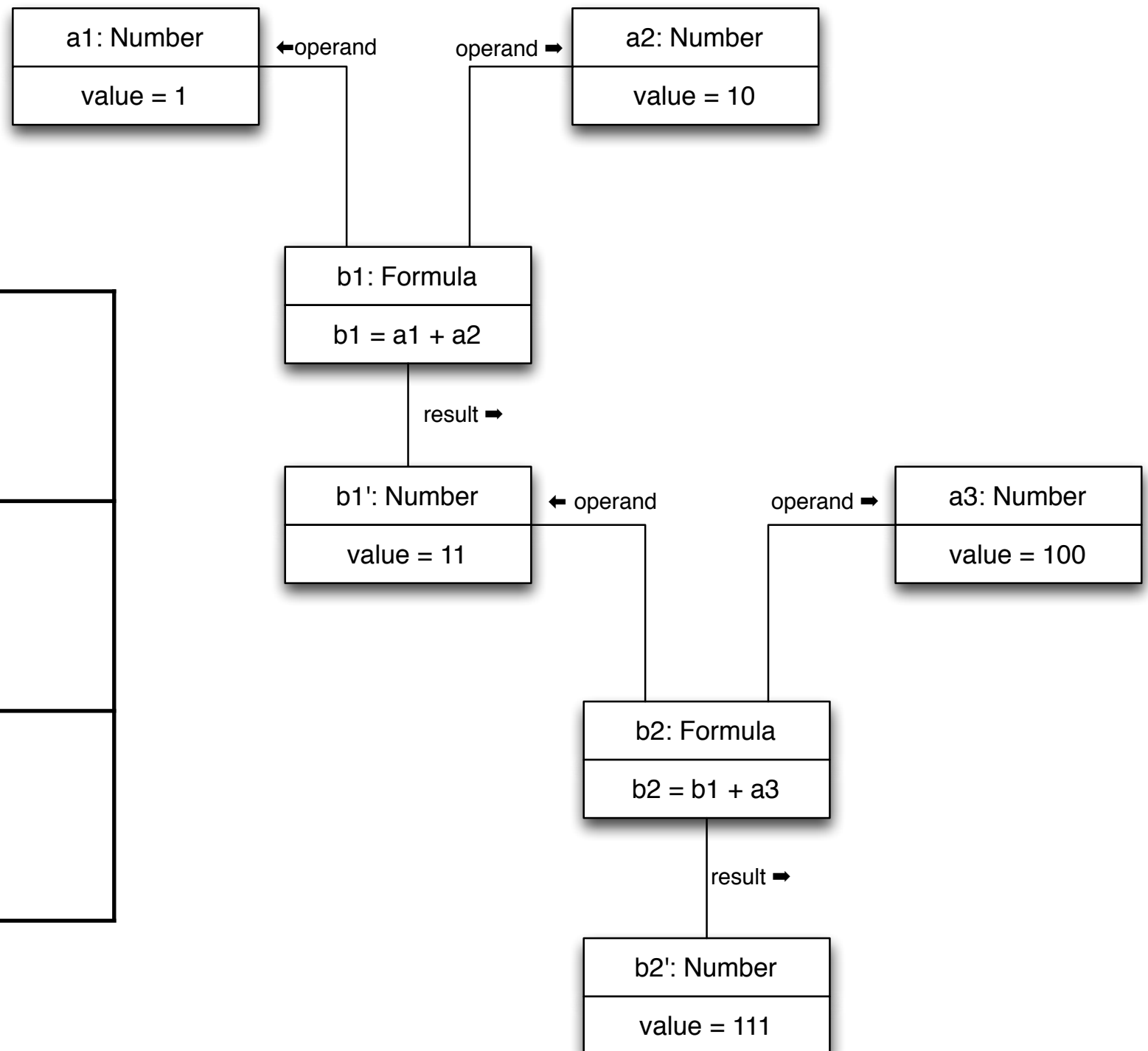
*There are multiple contents for a formula  
(that serve as operands)*

# Object Model



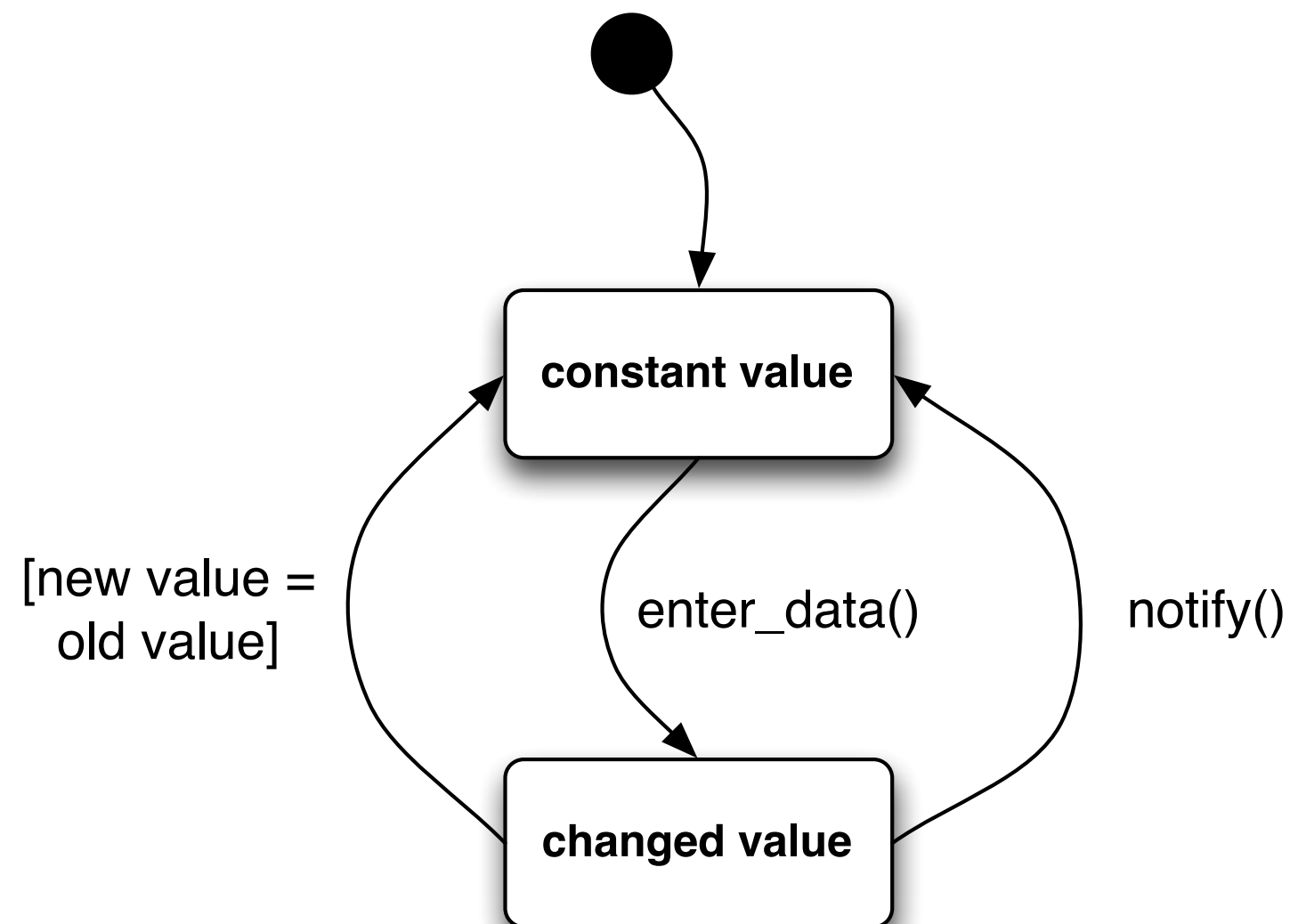
# Relationships between Objects

	A	B
I	I	II
2	10	III
3	100	



# State Chart

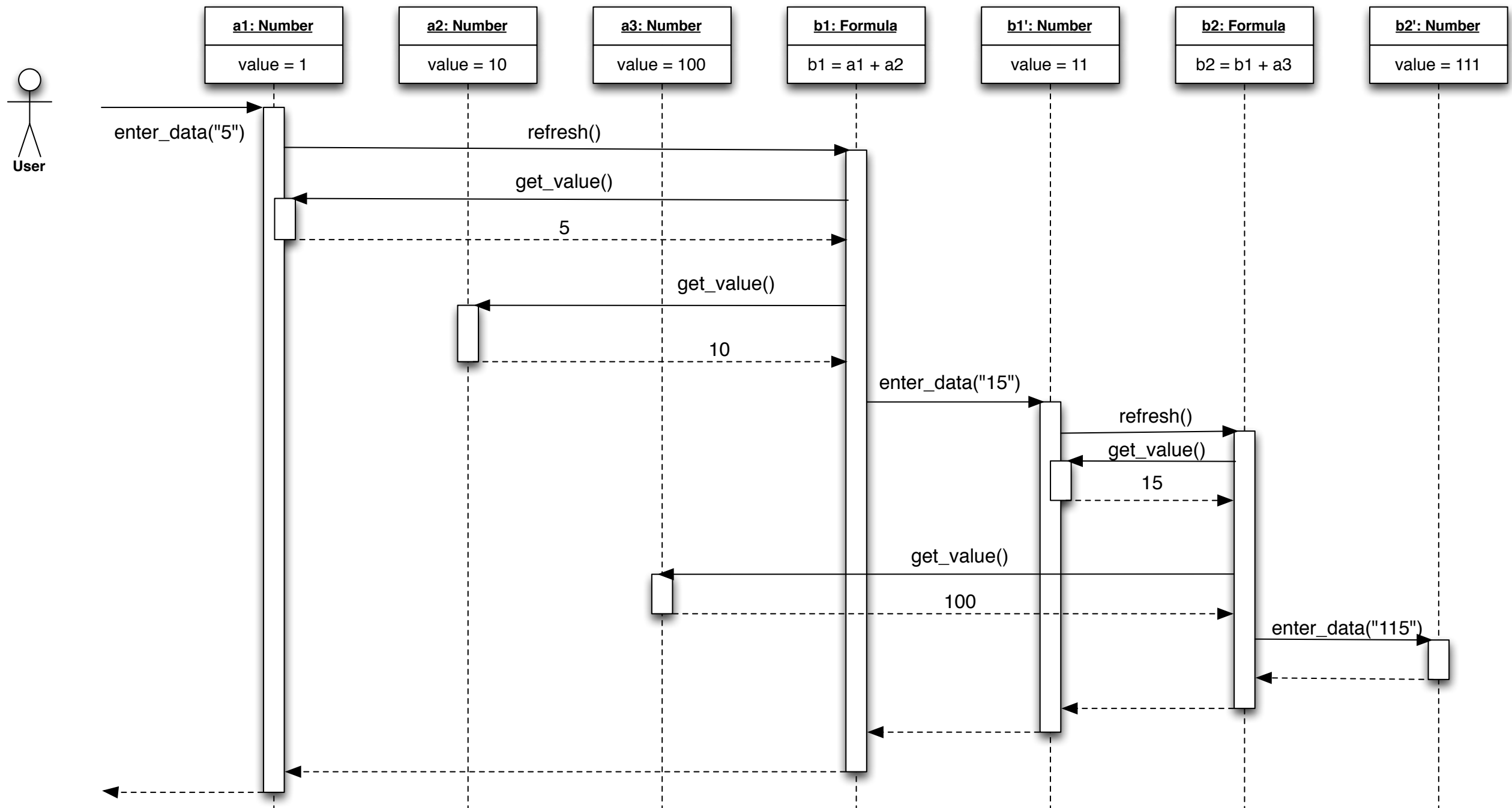
The method *enter\_data()* of the *Content* class examines whether the actual value has changed. If so, every *Formula* that has this *Content* as an operand is notified by means of the method *notify()*.





# Sequence Diagram

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



# Design Patterns

In this chapter we will examine typical usage scenarios of object-oriented programming - the so called *design patterns*.

The name *design pattern* was coined by the architect Christopher Alexander:

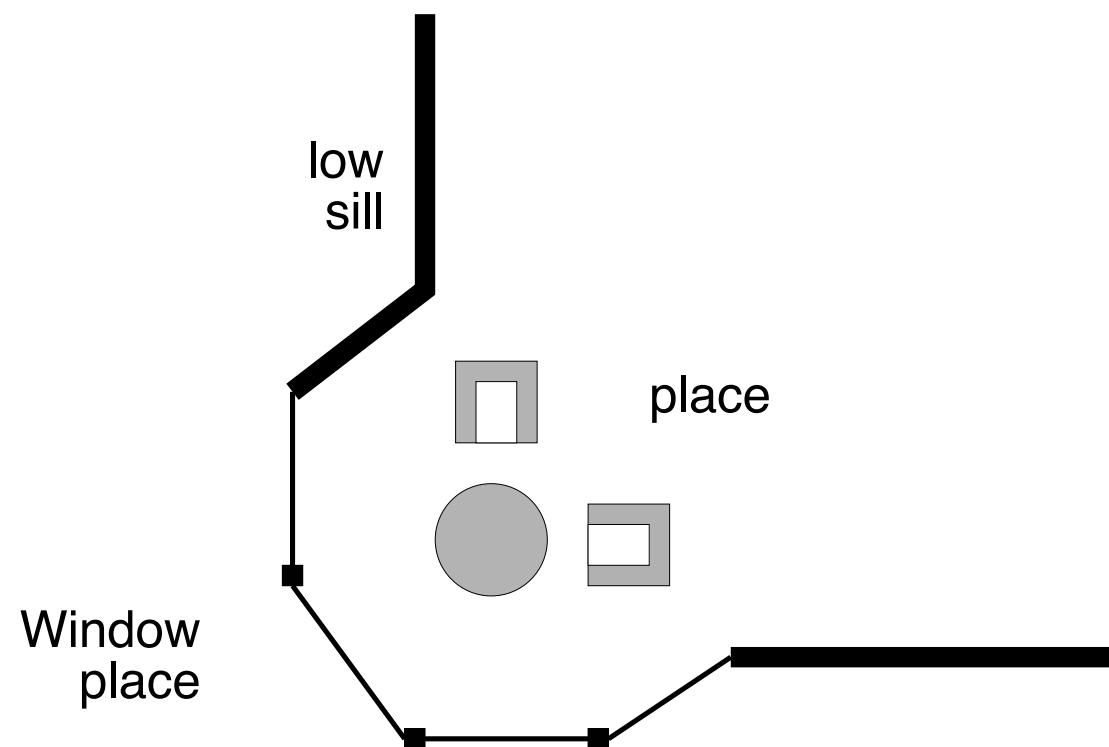
“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

A pattern is a *template* that can be used in many different situations.

# Patterns in Architecture: *Window Place*

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them

In every room where you spend any length of time during the day, make at least one window into a “window place”



# Patterns in Software Design

In our case design patterns are

descriptions of communicating objects and classes, that have been adapted to solve a design problem in a specific context.

# The Elements of a Pattern

Patterns are usually combined into *catalogues*: manuals that contain patterns for future reuse.

Every pattern can be described by at least four characteristics:

- Name
- Problem
- Solution
- Consequences

# The Elements of a Pattern (2)

The name of the pattern is used to describe the design problem and its solution in one or two words.

it enables us to

- design things on a higher level of abstraction
- use it under this name in the documentation
- speak of it

The problem describes when the pattern is used.

- it describes the problem and its context
- can describe certain design problems
- can contain certain *operational conditions*

# The Elements of a Pattern (3)

The **solution** describes the parts the design consists of, their relations, responsibilities and collaborations - in short, the *structure* and *participants*:

- not a description of a concrete design or an implementation
- but rather an *abstract description* of a design problem, and how a general interaction of elements solves it

The **consequences** are results, benefits and drawbacks of a pattern:

- *assessments of resource usage* (memory usage, running time)
- influence on *flexibility*, *extendiblility*, and *portability*

# Case Study: Text Editor *Lexi*

Let's consider the design of a "what you see is what you get" ("WYSIWYG") text editor called *Lexi*.

*Lexi* is able to combine text and graphics in a multitude of possible layouts.

Let's examine some design patterns that can be used to solve problems in *Lexi* and similar applications.



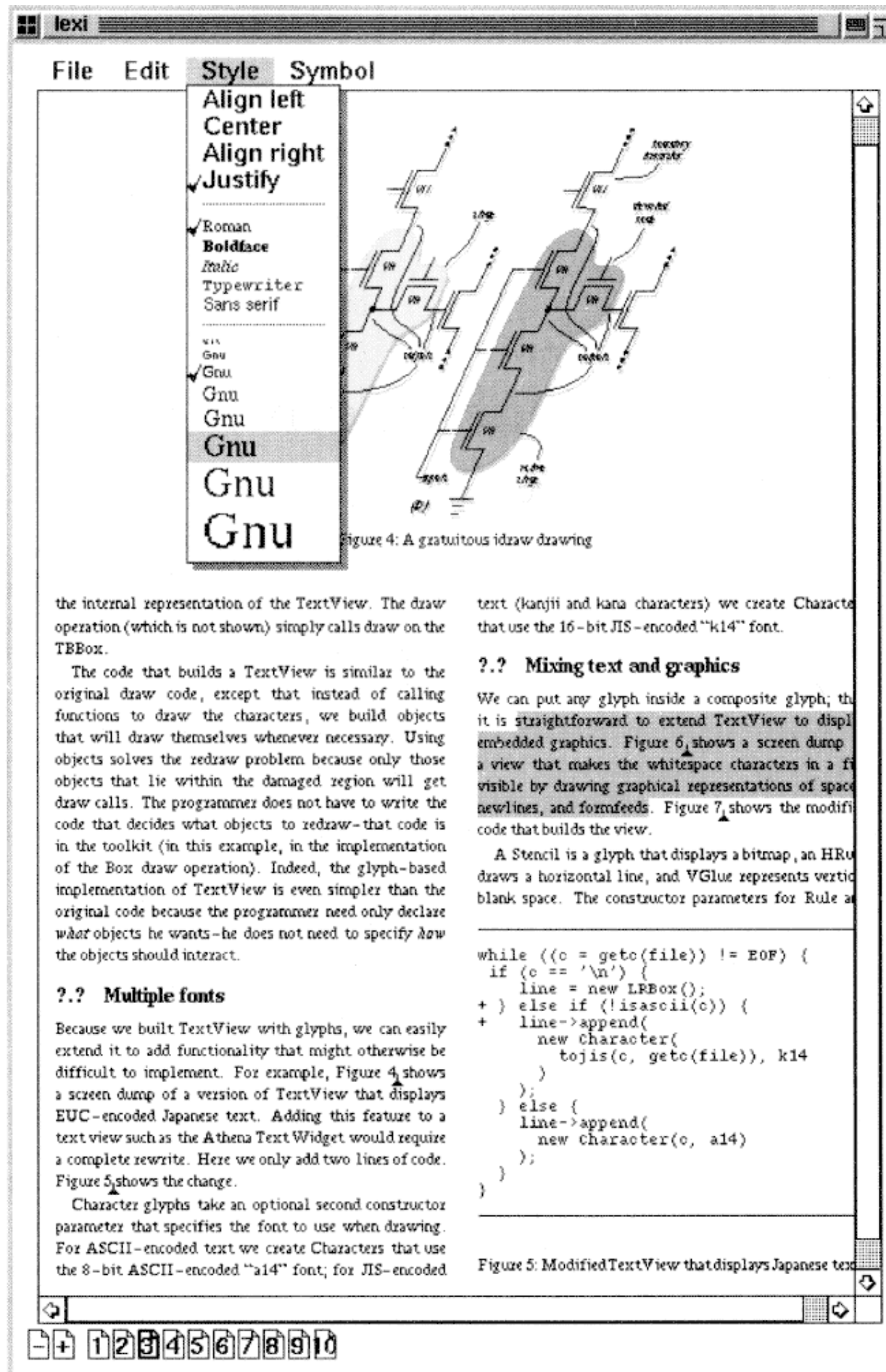


Figure 4: A gratuitous idraw drawing

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw—that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare what objects he wants—he does not need to specify how the objects should interact.

## 2.2 Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena TextWidget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded "a14" font; for JIS-encoded

text (kanji and kana characters) we create Characters that use the 16-bit JIS-encoded "k14" font.

## 2.2 Mixing text and graphics

We can put any glyph inside a composite glyph; thus it is straightforward to extend TextView to display embedded graphics. Figure 6 shows a screen dump of a view that makes the whitespace characters in a file visible by drawing graphical representations of space, newlines, and formfeeds. Figure 7 shows the modified code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRule draws a horizontal line, and VGlue represents vertical blank space. The constructor parameters for Rule are

```
while ((c = getc(file)) != EOF) {
  if (c == '\n') {
    line = new LRBox();
  } else if (!isascii(c)) {
+   line->append(
      new Character(
        tojis(c, getc(file)), k14
      )
    );
  } else {
    line->append(
      new Character(c, a14)
    );
  }
}
```

Figure 5: Modified TextView that displays Japanese text

# Challenges

**Document structure.** How is the document stored internally?

**Formatting.** How does *Lexi* order text and graphics as lines and polygons?

**Support for multiple user interfaces.** *Lexi* should be as independent of concrete windowing systems as possible.

**User actions.** There should be a unified method of accessing *Lexi*'s functionality and undoing changes.

Each of these design problems (and their solutions) is illustrated by one or multiple design patterns.

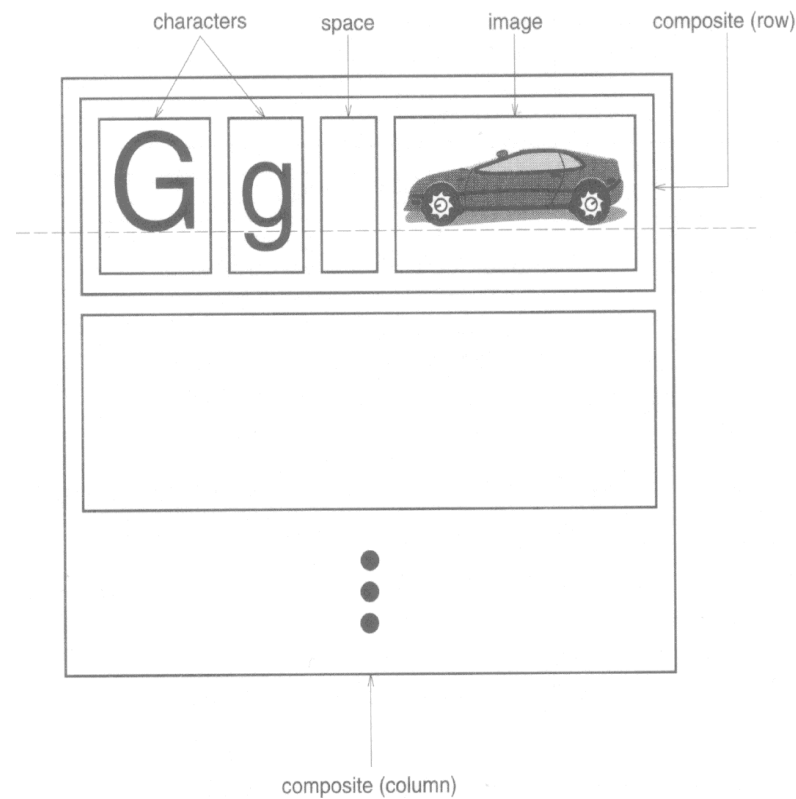
# Displaying Structure - Composite Pattern

A *document* is an arrangement of basic graphical elements like glyphs, lines, polygons etc.

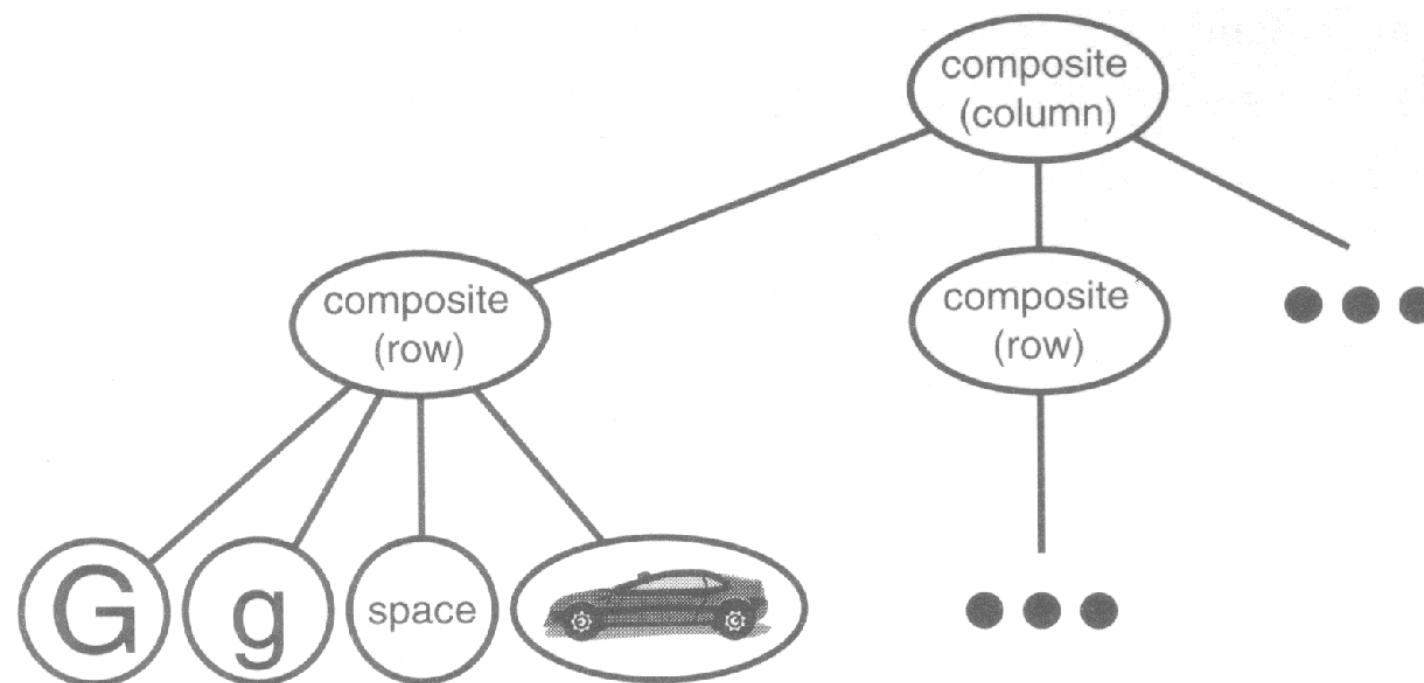
These are combined into *structures* - rows, columns, figures, and other substructures.

Such hierarchically ordered information is usually stored by means of *recursive composition* - simpler elements are combined into more complex ones.

# Elements in a Document

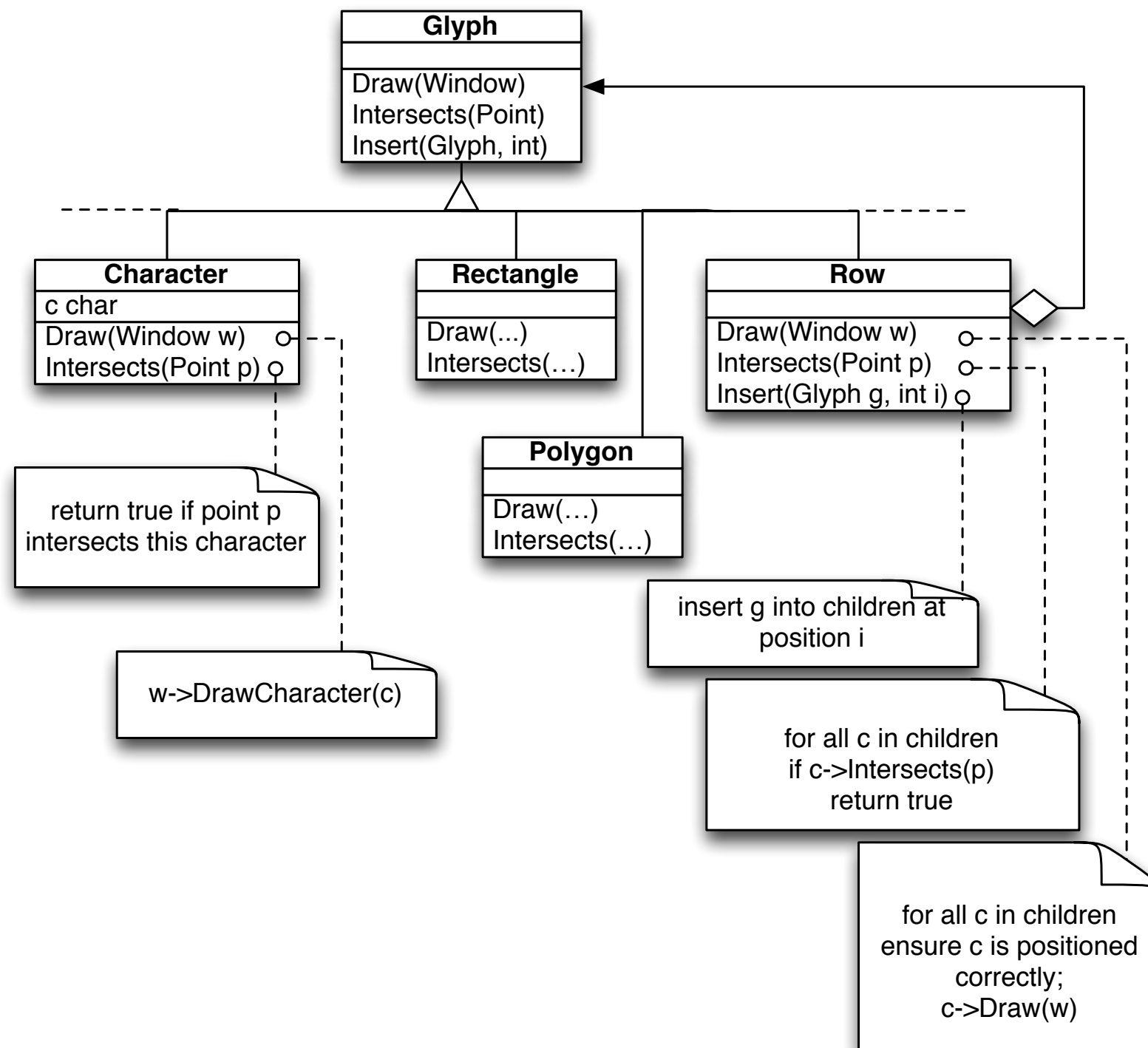


For each important element there is an individual object.



# Glyphs

We define an abstract superclass Glyph for all objects that can occur in a document.



# Glyphs (2)

Each glyph knows

- how to draw itself (by means of the Draw() method). This abstract method is implemented in concrete subclasses of Glyph.
- how much space it takes up (like in the Intersects() method).
- its children and parent (like in the Insert() method).

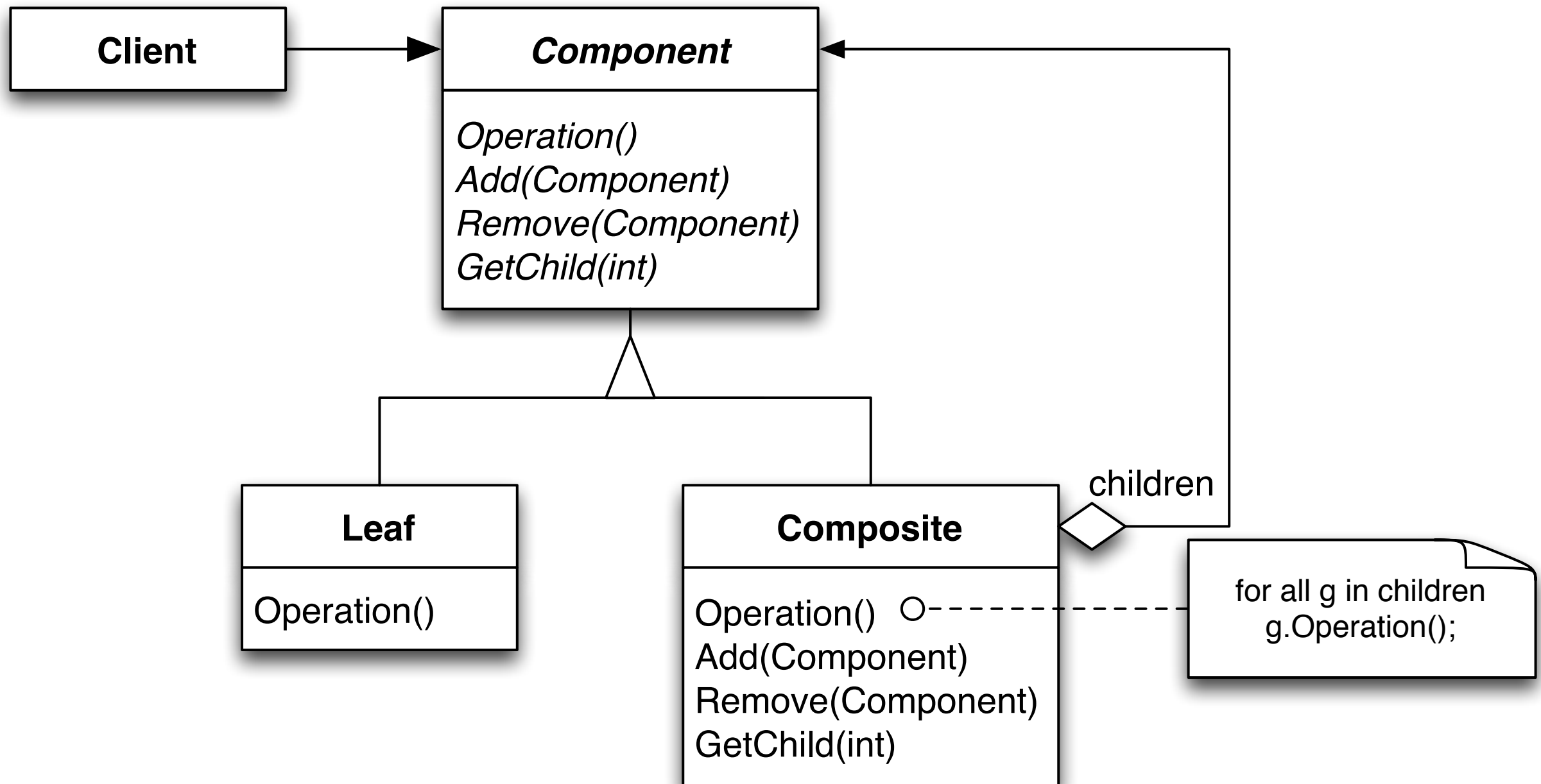
The class hierarchy of the *Glyph* class is an instance of the *composite* pattern.

# The Composite Pattern

**Problem**      Use the composite pattern if

- you want to express a part-of-a-whole hierarchy
- the application ignores differences between composed and simple objects

# Structure





# Participants

## Component (Glyph)

- defines the interface for all objects (simple and composed)
- implements the default behavior for the common interface (where applicable)
- defines the interface for accessing and managing of subcomponents (children)

## Leaf (e.g. rectangle, line, text)

- provides for basic objects; a leaf doesn't have any children
- defines common behavior of basic elements

# Participants (2)

## Composite (e.g. picture, column)

- defines common behavior of composed objects (those with children)
- stores subcomponents (children)
- implements methods for accessing children as per interface of *Component*

## Client (User)

- manages objects by means of the *Component* interface

# Consequences

## The composite pattern

- defines class hierarchies consisting of composed and basic components
- simplifies the user: he can use basic and composed objects in the same way; he doesn't (and shouldn't) know whether he is handling a simple or complex object.

# Consequences (2)

## The composite pattern

- simplifies adding of new kinds of elements
- can *generalize* the design too much: for example, the fact that a certain composed element has a fixed number of children, or only certain kinds of children can only be checked at runtime (and not at compile time).  
⇐ *This is a drawback!*

Other known fields of application: expressions, instruction sequences

# Encapsulating of Algorithms - Strategy Pattern

*Lexi* has to wrap the text in rows and combine rows into columns - as the user wishes it.

This is the task of the formatting algorithm.

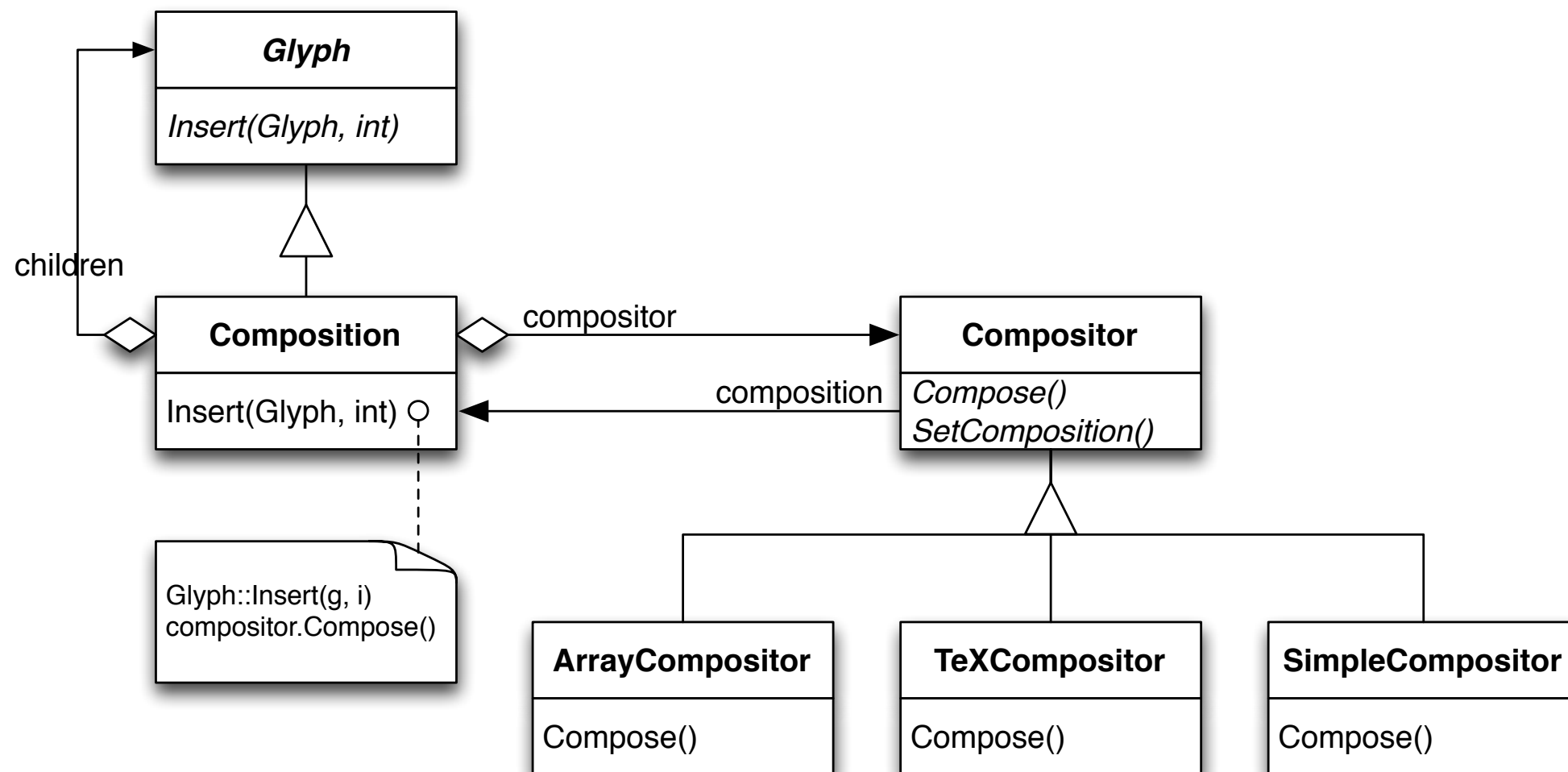
*Lexi* must support multiple formatting algorithms e.g.

- a fast, imprecise ("quick-and-dirty") algorithm for the WYSIWYG view
- a slow and precise one for printing

In accordance with the separation of interests, the formatting algorithm must be independent of the document structure.

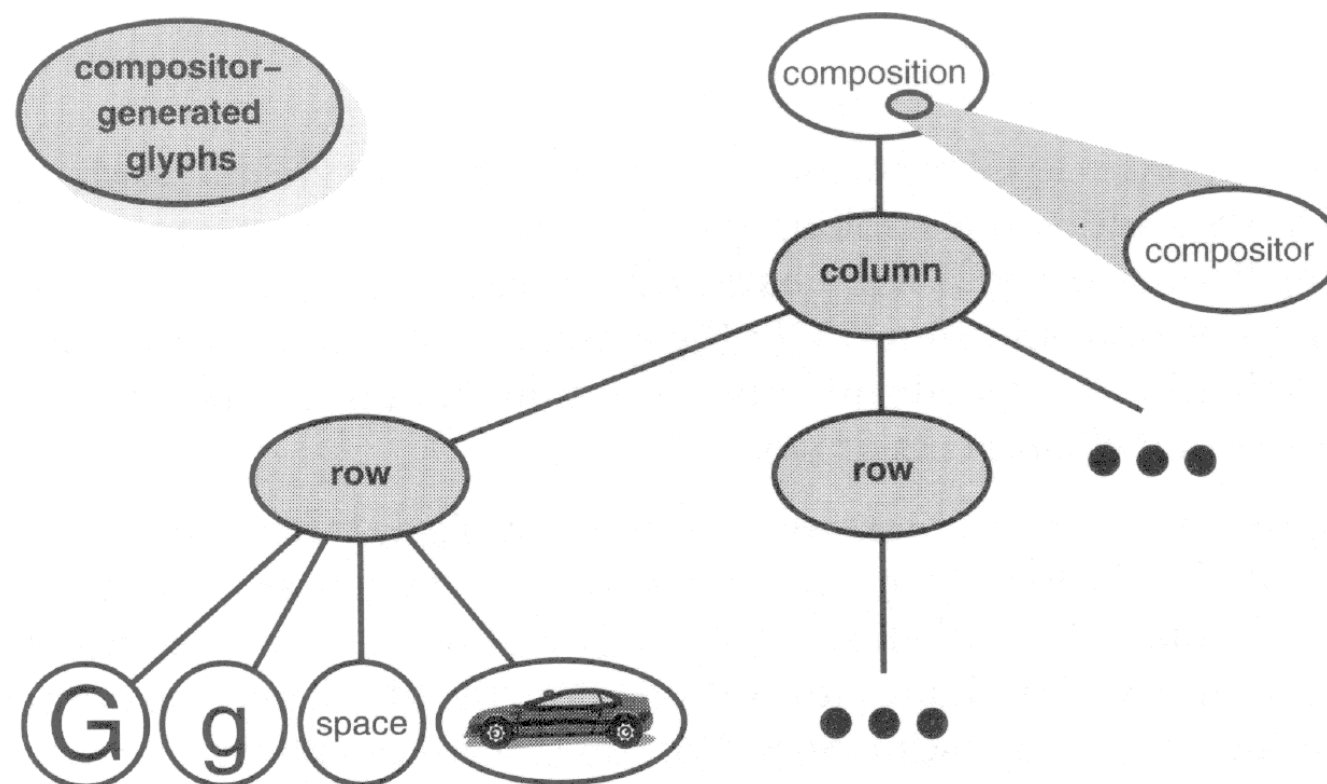
# Formatting Algorithms

We define a separate class hierarchy for objects that encapsulate certain formatting algorithms. The root of this hierarchy is the *Compositor* abstract class with a general interface; every subclass implements a concrete formatting algorithm.



# Formatting Algorithms (2)

Every Compositor traverses the document structure and possibly inserts new (composed) Glyphs:



This is an instance of the strategy pattern.

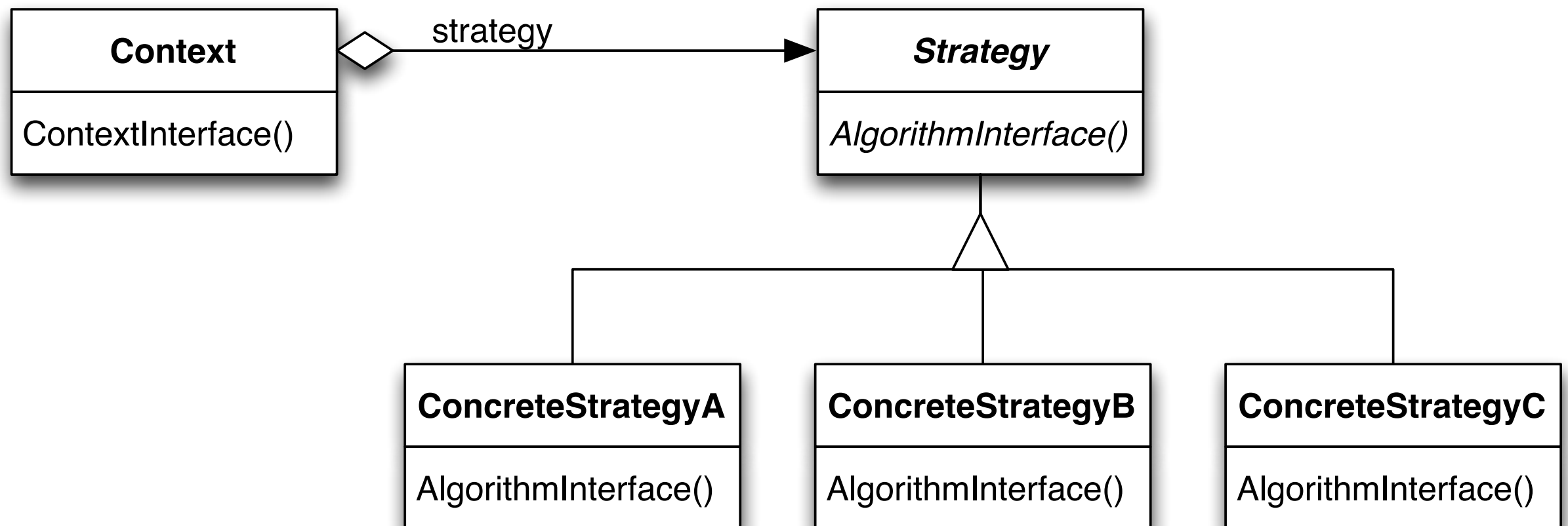
# Strategy Pattern

**Problem**      Use the strategy pattern if

- multiple connected classes differ only in behavior
- different variants of an algorithm are needed
- an algorithm uses data that shall be concealed from the user



# Structure



# Participants

## **Strategy** (Compositor)

- defines a common interface for all supported algorithms

## **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)

- implements the algorithm as per Strategy interface

## **Context** (Composition)

- is configured with a ConcreteStrategy object
- references a Strategy object
- can define an interface that makes data available to Strategy

# Consequences

## The strategy pattern

- makes conditional statements unnecessary (e.g. if simple-composition then... else if tex.composition...)
- helps to identify the common functionality of all the algorithms
- enables the user to choose a strategy...
- ... but burdens him with a choice of strategy!
- can lead to a communication overhead: data has to be provided even if the chosen strategy doesn't make use of it

Other fields of application: code optimization, memory allocation, routing algorithms

# User Actions - Command Pattern

*Lexi's* functionality is accessible in multiple ways: you can manipulate the WYSIWYG representation (enter text, move the cursor, select text), and you can choose additional actions via menus, panels, and hotkeys.

We don't want to bind any action to a specific user interface because

- there may be multiple ways to initiate the same action (you can navigate to the next page via a panel, a menu entry, and a keystroke)
- maybe we want to change the interface at some later time

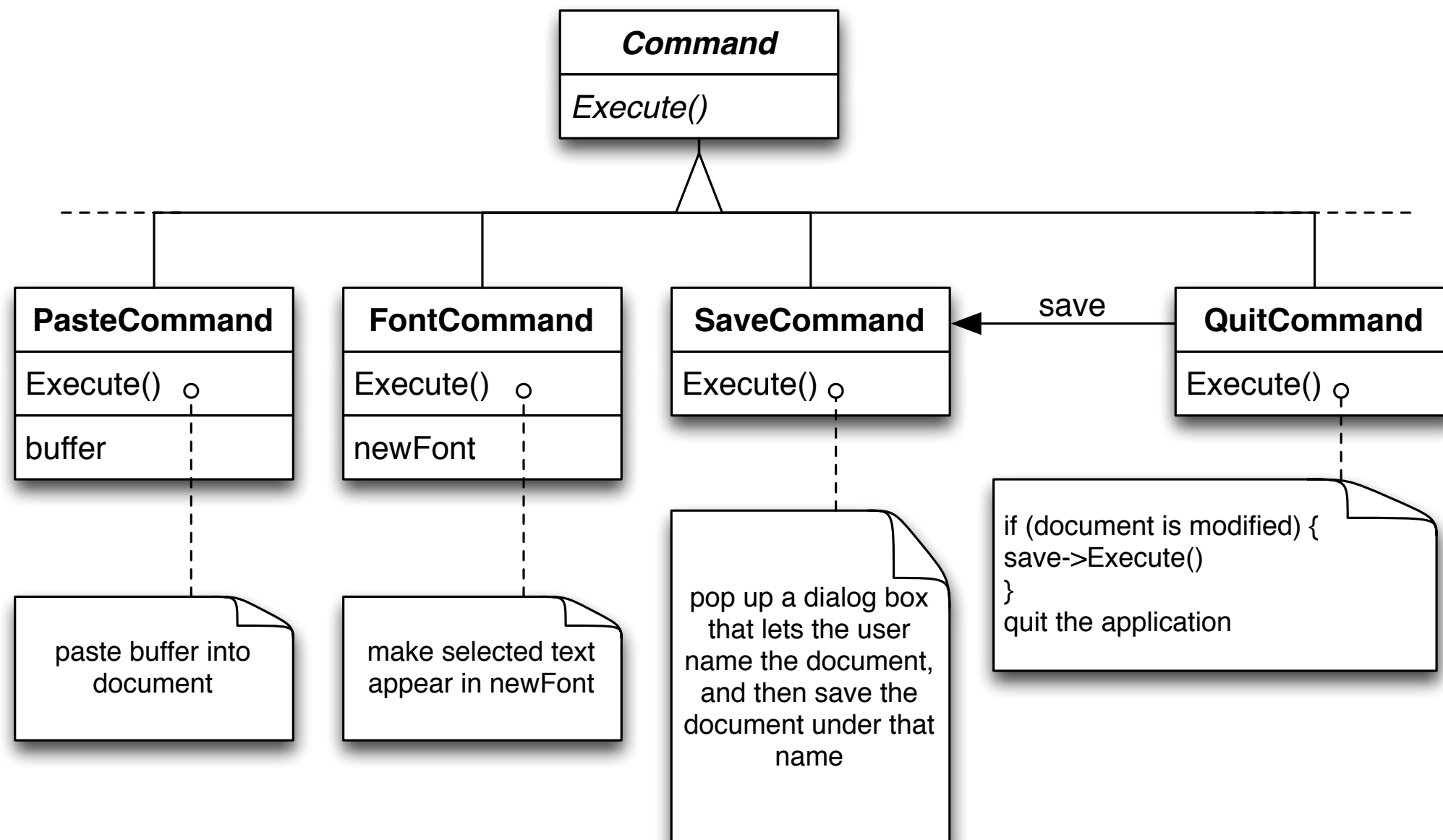
# User Actions (2)

To complicate things even more, we want to enable *undoing* and *redoing* of multiple actions.

Additionally, we want to be able to record and play back *macros* (instruction sequences).

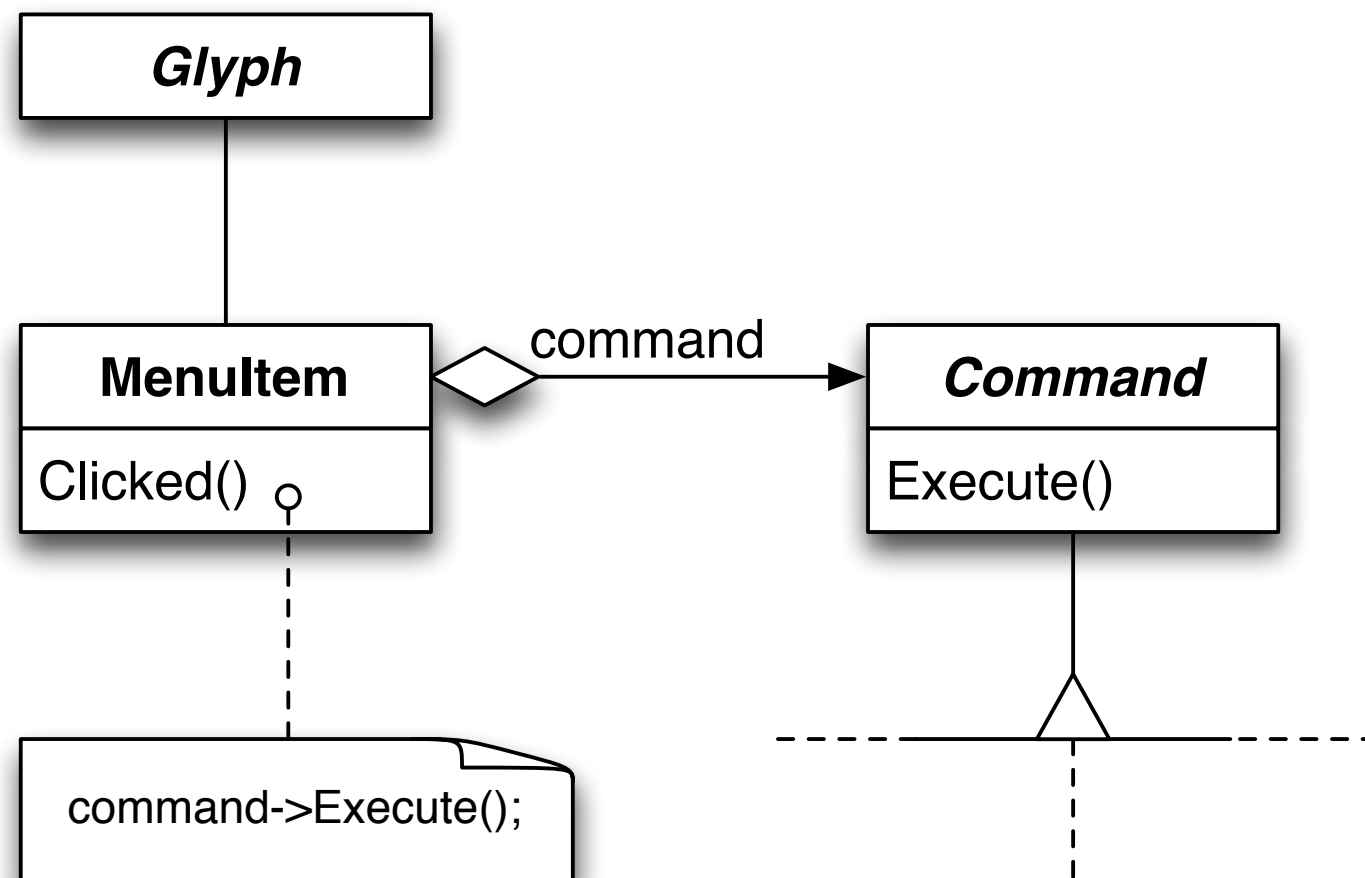
# User Actions (3)

Therefore we define a *Command* class hierarchy, which encapsulates the user actions.



# User Actions (4)

Specific glyphs can be bound to user actions; they are executed when the glyph is activated.



This is an instance of the *command* pattern.

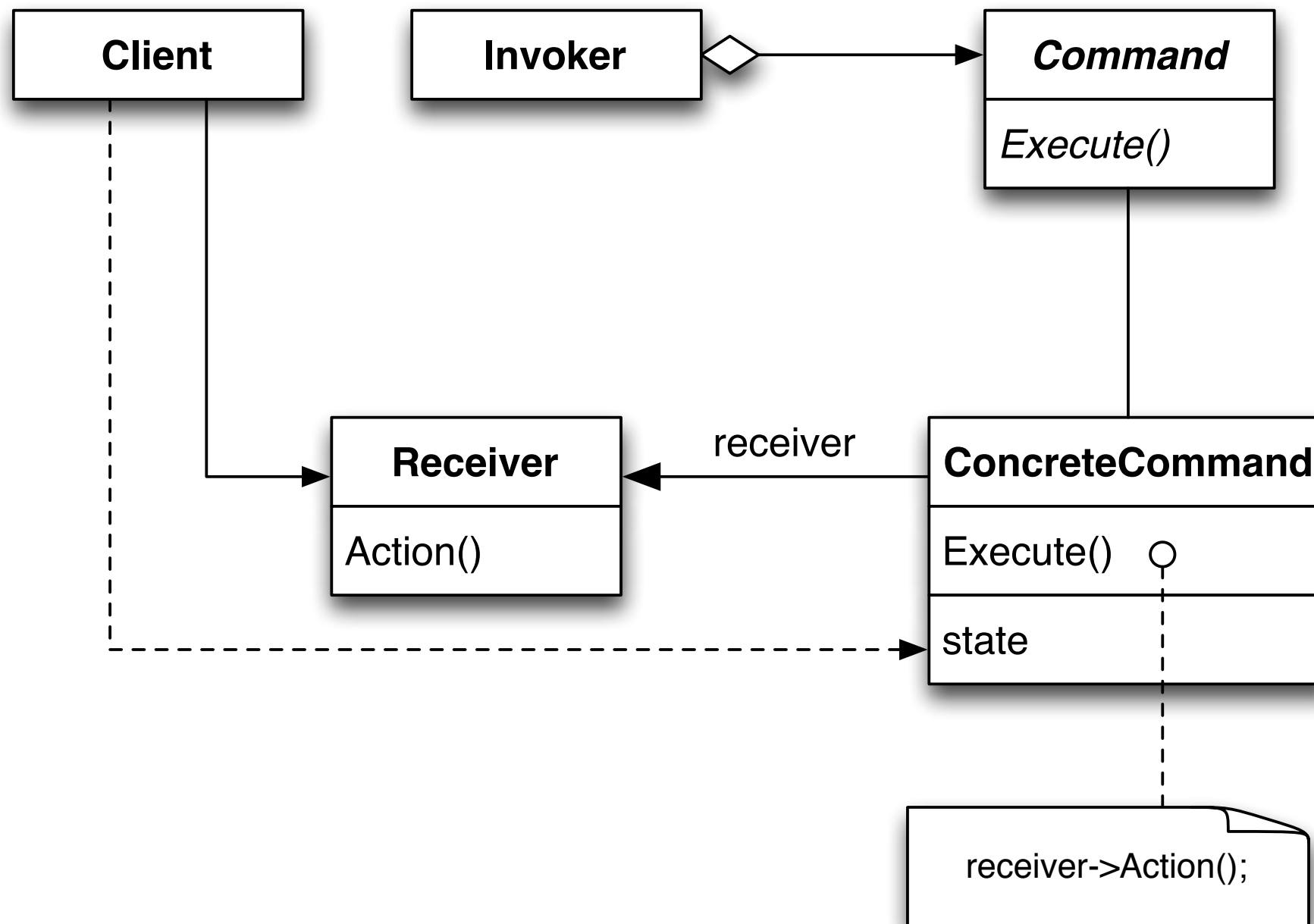
# Command Pattern

**Problem**      Use the command pattern if you want to

- parameterize objects with the action to be performed
- trigger, enqueue, and execute instructions at different points in time
- support undoing of instructions
- log changes to be able to restore data after a crash



# Structure



# Participants

## **Command**

- defines the interface to execute an action

## **ConcreteCommand** (PasteCommand, OpenCommand)

- defines a coupling between a receiving object and an action
- implements Execute() by calling appropriate methods on the receiver

## **Client** (User, Application)

- creates a ConcreteCommand object and sets a receiver

# Participants (2)

## Invoker (Caller, MenuItem)

- ask the instruction to execute its action

## Receiver (Document, Application)

- knows how the methods, that are coupled with an action, are to be executed. Any class can be a receiver.

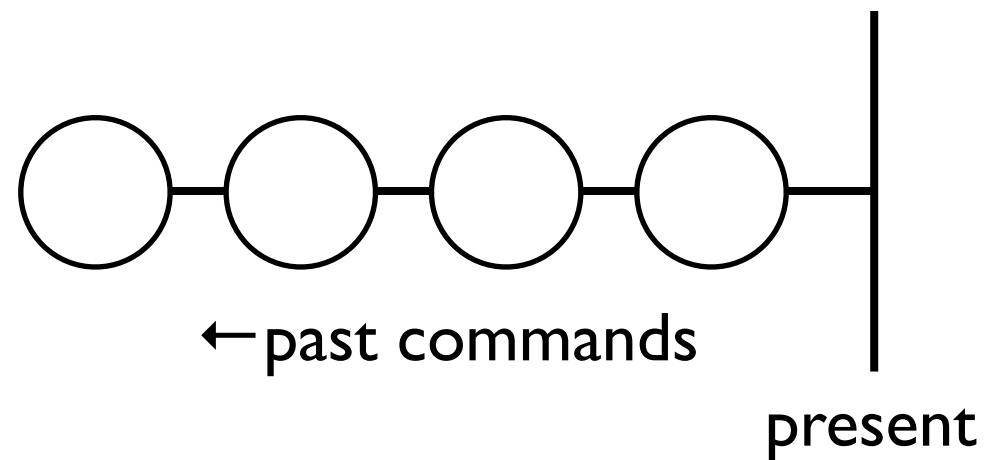
# Consequences

## The command pattern

- decouples the object that triggers an action from the object that knows how to execute it
- implements Commands as *first-class* objects that can be handled and extended like any other object
- allows to combine Commands from other Commands
- makes it easy to add new Commands because existing classes don't have to be changed

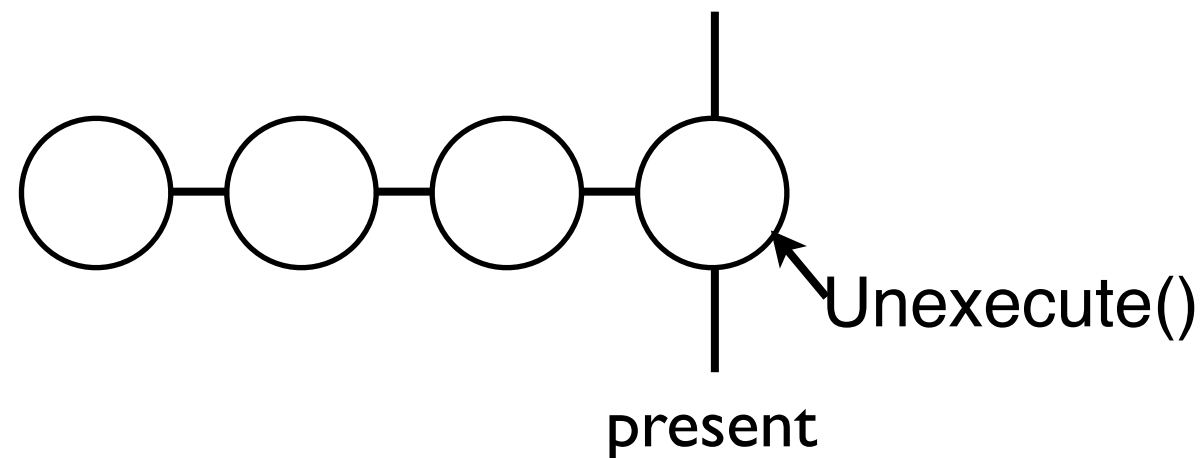
# Undoing Commands

With the help of a Command-Log we can easily implement command undoing. It looks like this:



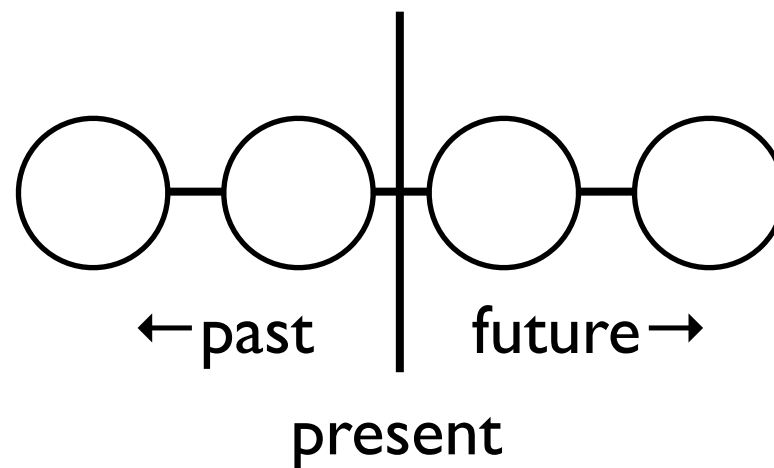
# Undoing Commands (2)

To undo the last command we call `Unexecute()` on the last command. This means that each command has to store enough state data to be able to undo itself.



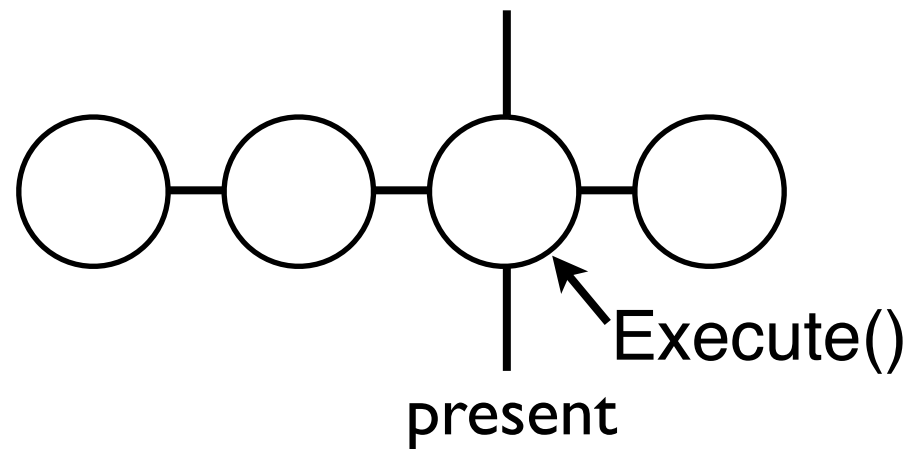
# Undoing Commands (3)

After undoing, we move the "Present-Line" one command to the left. If the user chooses to undo another command we end up in this state:



# Undoing Commands (4)

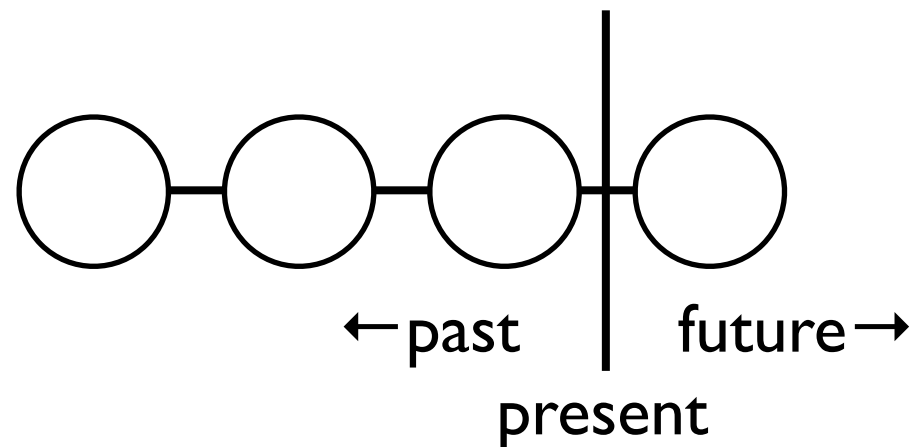
To redo a command, we simply have to call `Execute()` on the current command...





# Undoing Commands (5)

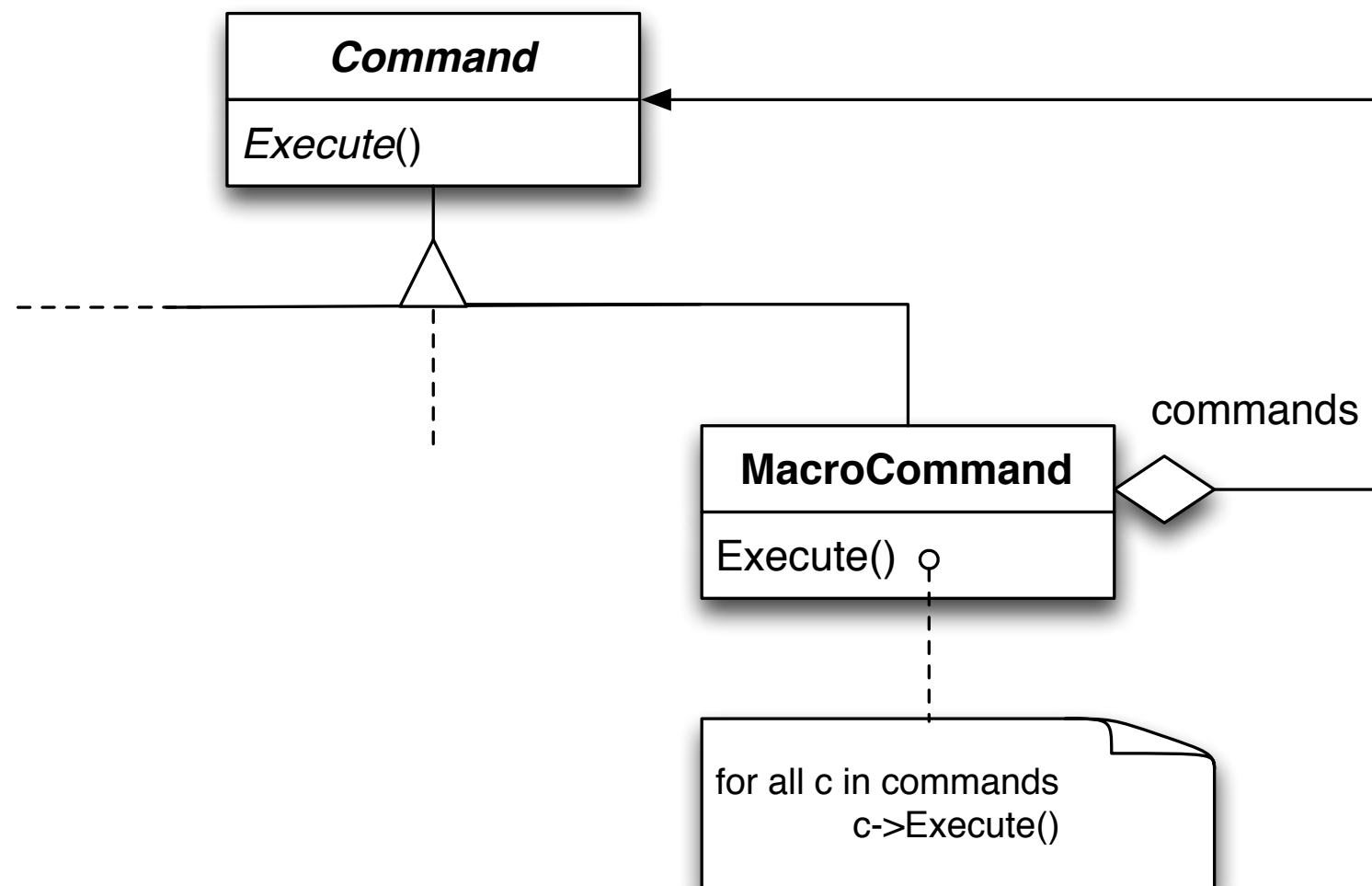
... and move the "Present-Line" one command to the right, so the next call to `Execute()` will redo the next command.



This way the user can navigate back and forth in time depending on how far he has to go to correct an error.

# Macros

Lastly, let's consider an implementation of macros (instruction sequences). We use the command pattern and create a MacroCommand class that contains multiple command and can execute them successively:



If we add an `Unexecute()` method to the **MacroCommand** class, then we can undo macros like any other command.

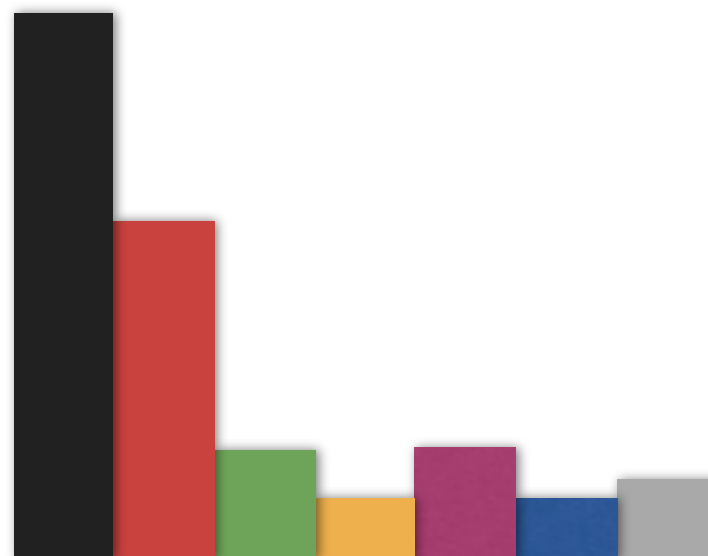
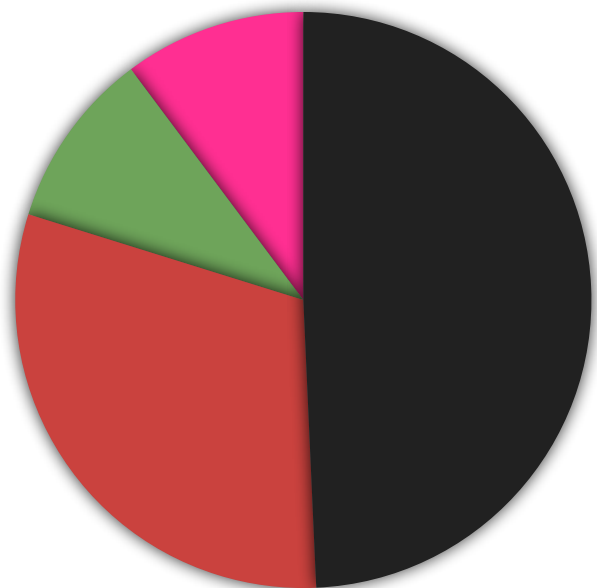
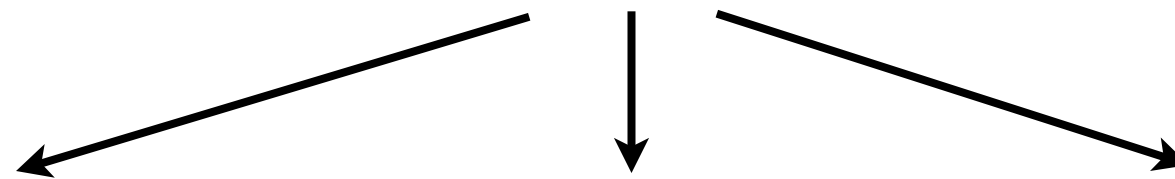
# Model-View-Controller

The *Model-View-Controller* pattern is one of the best known and most common patterns in the architecture of *interactive* systems.

# Example: Election Day

CDU/CSU	41.5%
SPD	25.7%
GRÜNE	8.4%
FDP	4.8%
LINKE	8.6%
AfD	4.7%
Sonstige	6.2%

Bundestagswahl  
22.09.2013



CDU/CSU	0.415
SPD	0.257
GRÜNE	0.084
FDP	0.048
LINKE	0.086
PIRATEN	0.047
Sonstige	0.062

# Problem

User interfaces are most frequently affected by changes.

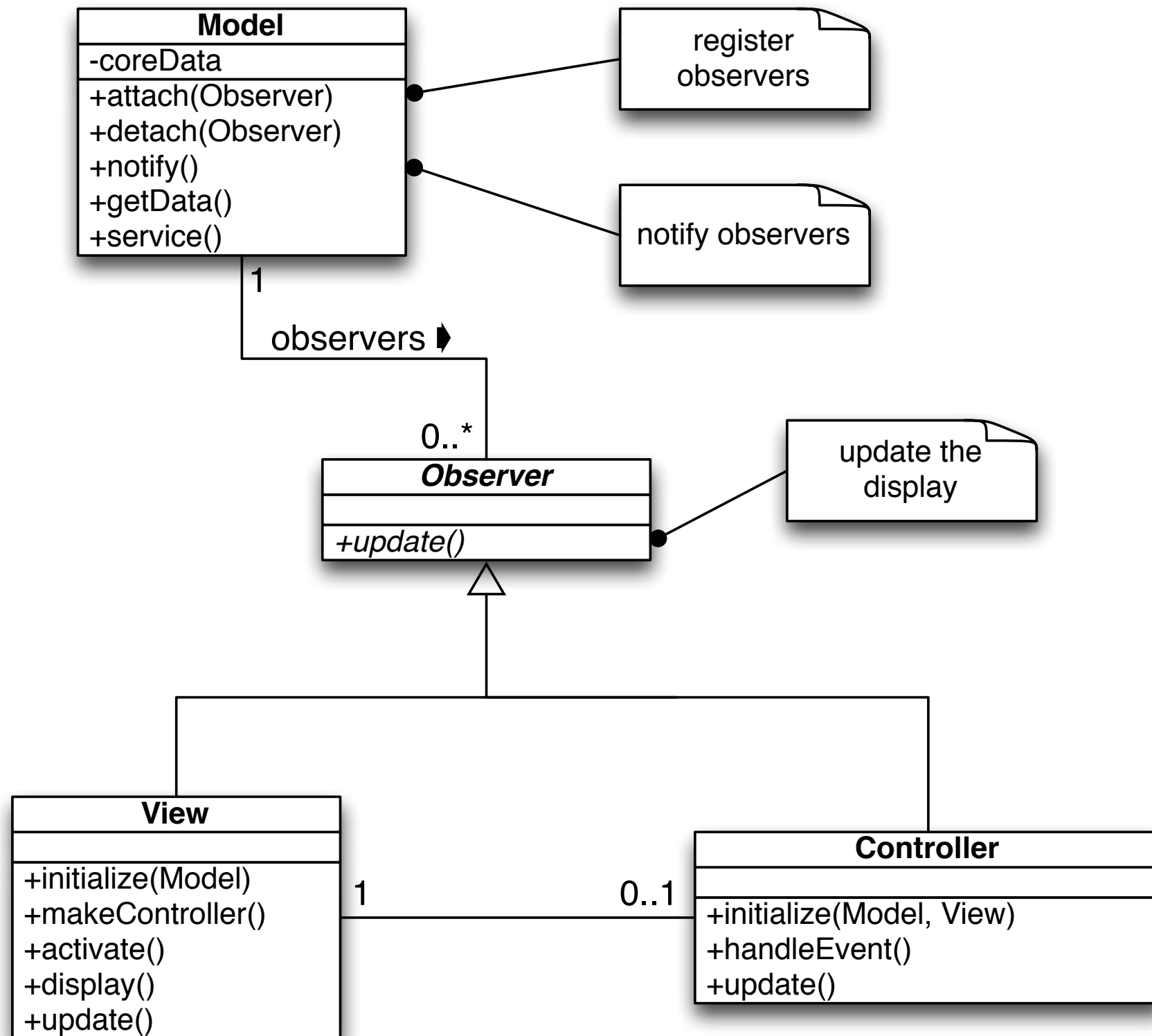
- How can I represent the same information in different ways?
- How can I guarantee that changes in the dataset will be instantly reflected in all views?
- How can I change the user interface? (possibly at runtime)
- How can I support multiple user interfaces without changing the core of the application?

# Solution

The *Model-View-Controller* pattern splits the application into three parts:

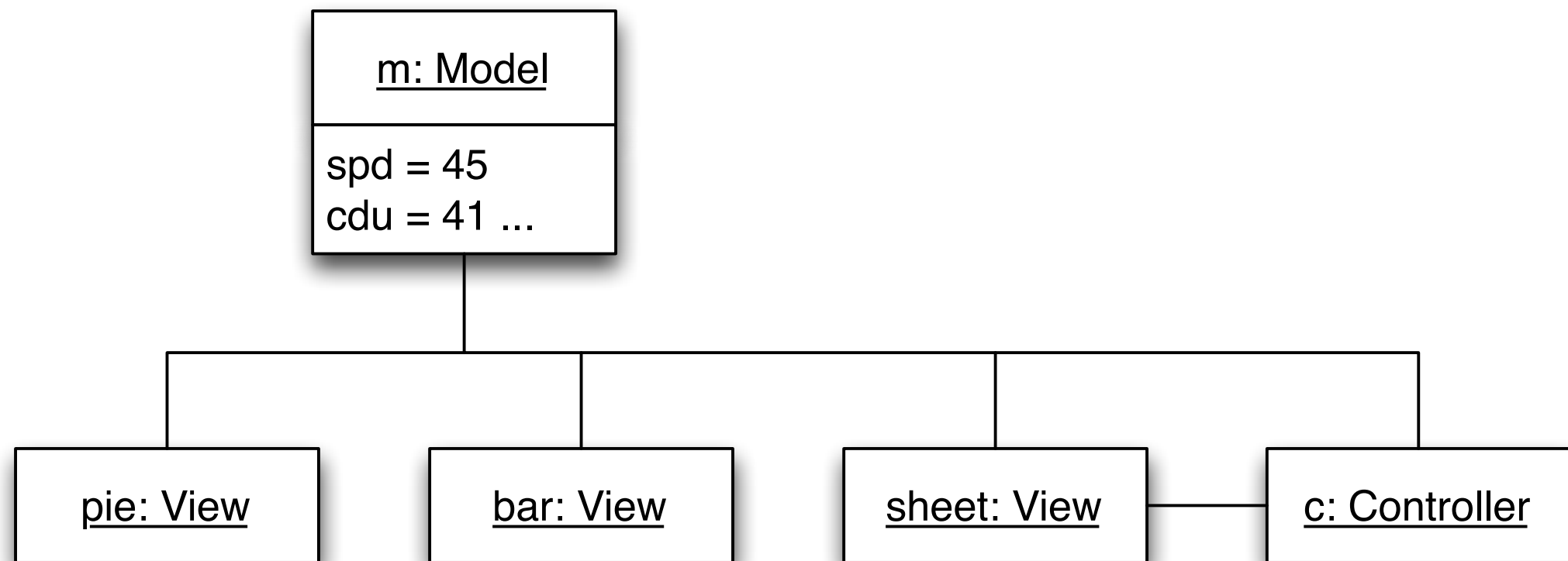
- The *model* is responsible for processing,
- The *view* takes care of output,
- The *controller* concerns itself with input

# Structure



# Structure (2)

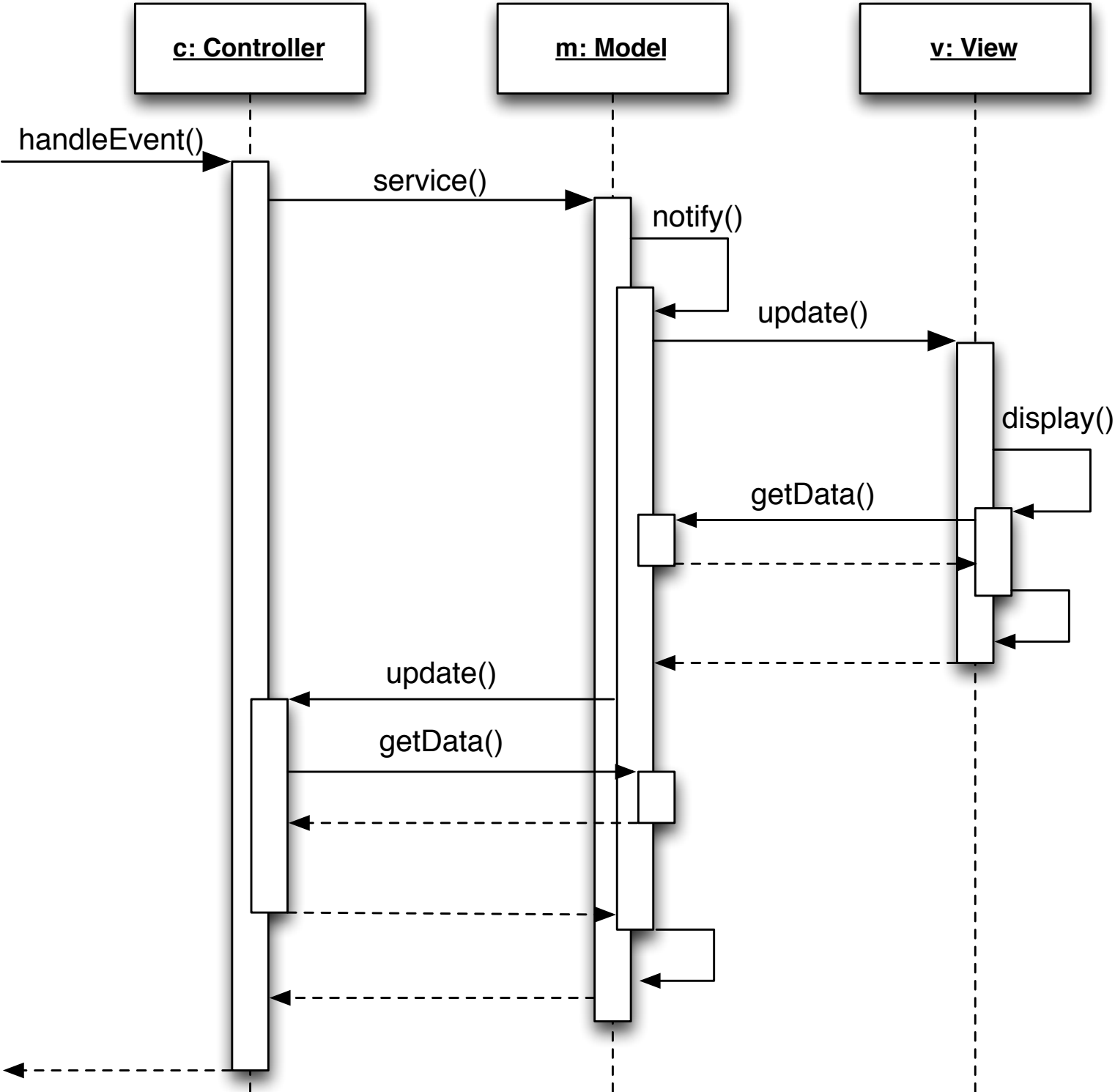
Each model can *register* multiple *observers* (= views and controllers).



As soon as the model changes, all registered observers are *notified*, and they update themselves accordingly.



# Dynamic behavior



# Consequences of the Model-View-Controller Pattern

## Benefits

- multiple views of the same system
- synchronous views
- attachable views and controllers

## Drawbacks

- increased complexity
- strong coupling between Model and View
- Strong coupling between Model and Controllers (can be avoided by means of the command pattern)

Known applications: GUI libraries, Smalltalk, Microsoft Foundation Classes

# In Summary

In summary, design patterns offer:

**A common design vocabulary.** Design patterns offer a common design vocabulary for software engineers for communicating, documenting, and exchanging design alternatives.

**Documentation and learning help.** The most large object-oriented systems use design patterns. Design patterns help to understand such systems.

**An extension of existing methods.** Design patterns concentrate the experience of experts - independently of the design method.

*“The best designs will use many design patterns that dovetail and intertwine to produce a greater whole.”*

# Anti-Patterns



If the following  
patterns occur in your  
software project,  
you're doing it wrong!

# The Blob



# The Golden Hammer





# Copy and Paste Programming



# Anti-Patterns: Programming

**The Blob.** (aka “God Class”) One object("blob") has the majority of the responsibilities, while most of the others just store data or provide only primitive services.

*Solution:* refactoring

**The Golden Hammer.** A favorite solution ("Golden Hammer") is applied to every single problem: With a hammer, every problem looks like a nail.

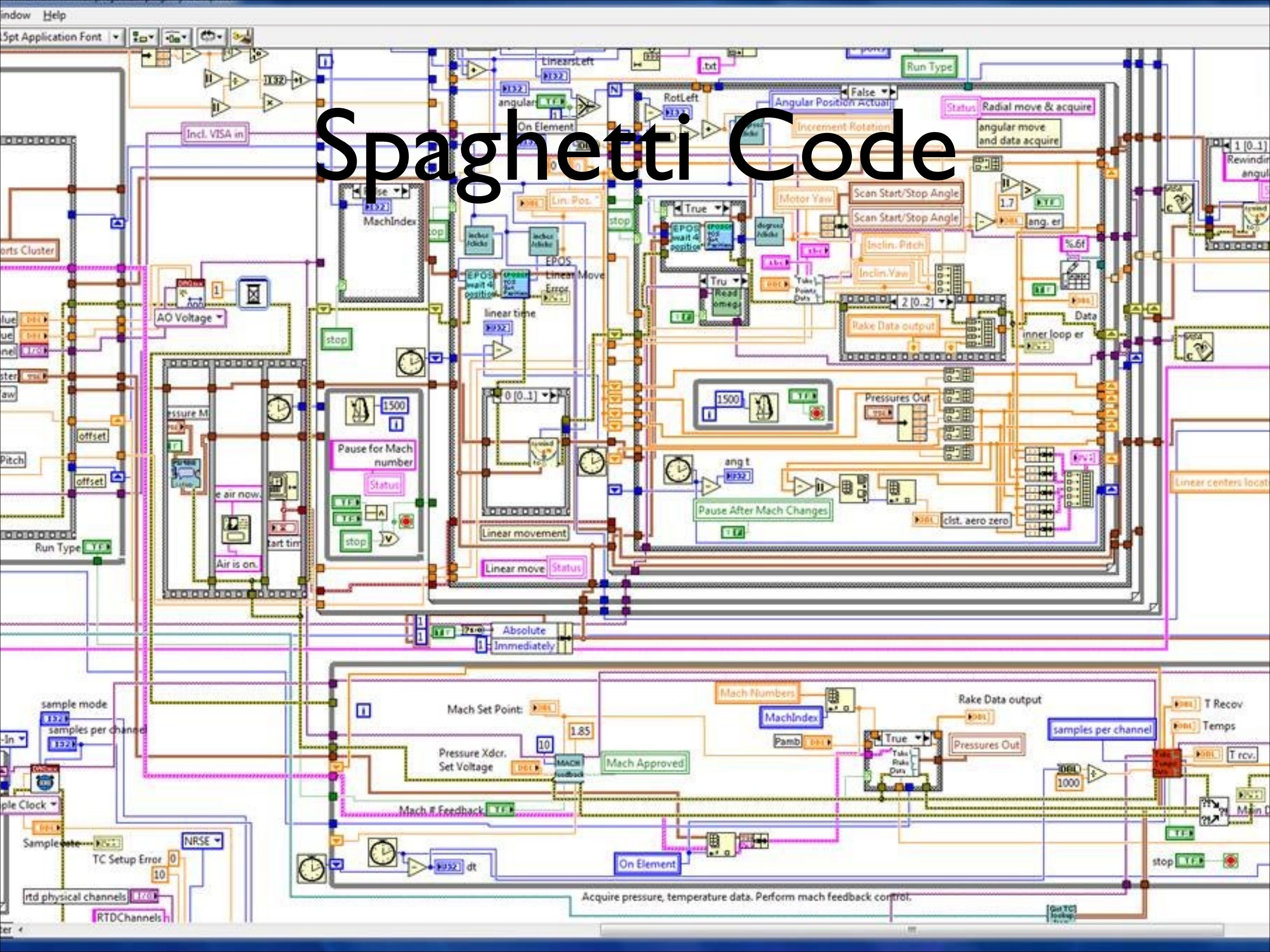
*Solution:* improve level of education

**Copy-and-Paste Programming.** Code is reused in multiple places by being copied and adjusted. This causes a maintenance problem.

*Solution:* Black box reuse, identifying of common features.



# Spaghetti Code





# Mushroom Management



# Anti-Patterns: Programming (2)

**Spaghetti Code.** The code is mostly unstructured; it's neither particularly modular nor object-oriented; control flow is obscure.

*Solution:* Prevent by designing first, and only then implementing. Existing spaghetti code should be refactored.

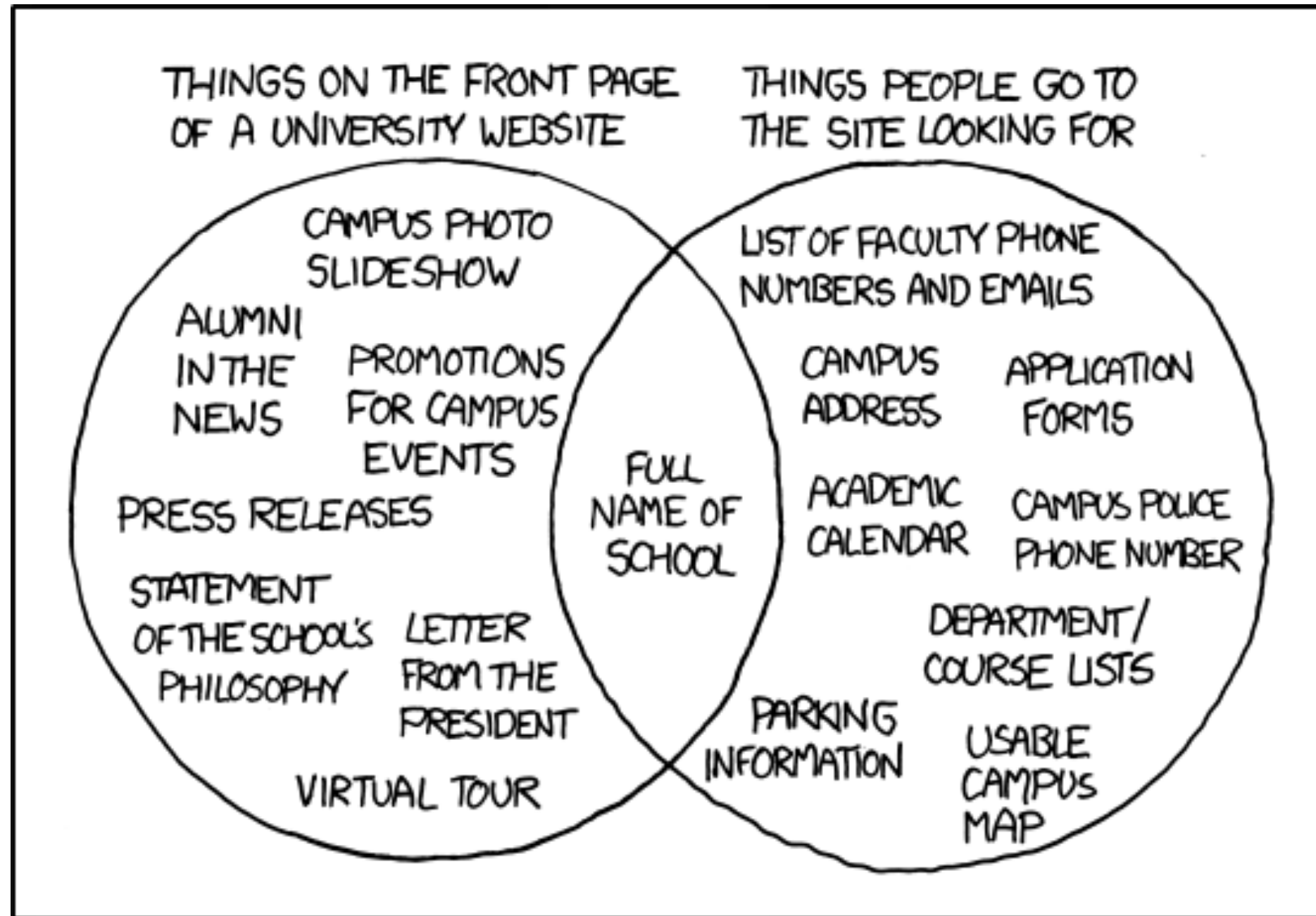
**Mushroom Management.** Developers are kept away from users.

*Solution:* Improve contacts.

# Vendor Lock-In



# Design by Committee





# Uni Saar App



[View In iTunes](#)

This app is designed for both iPhone and iPad

**Free**

Category: [Education](#)

Released: 07 November 2013

Version: 1.0

Size: 24.5 MB

Languages: English, German

Developer: Universitaet des Saarlandes

© 2013 Universität des Saarlandes

**Rated 4+**

**Compatibility:** Requires iOS 5.0 or later. Compatible with iPhone, iPad, and iPod touch. This app is optimized for iPhone 5.

## Customer Ratings

We have not received enough ratings to display an average for the current version of this application.

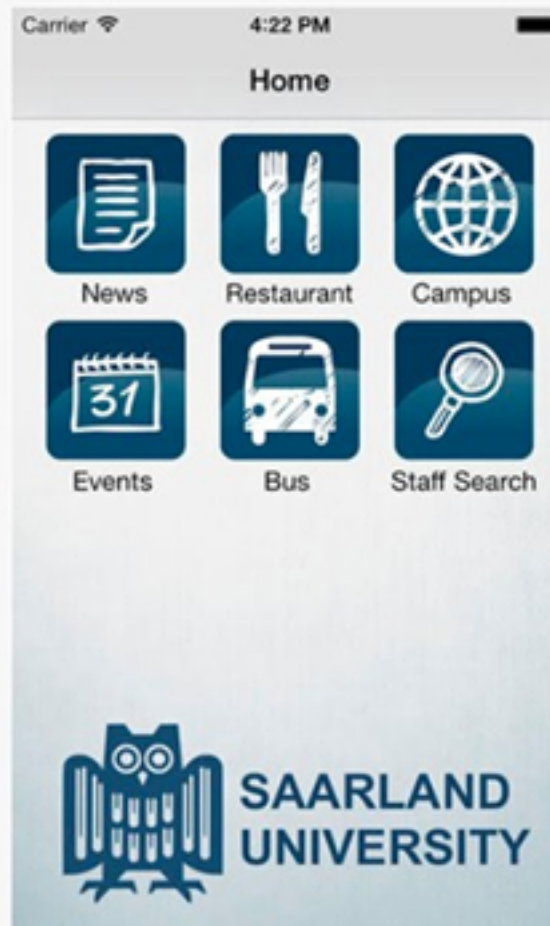
## Description

This app provides useful information concerning live and studies at Saarland University. You can browse the campus map, check out current food in the Mensa and other restaurants, look for your bus home, view current events and search for staff members.

[Uni Saar Support](#)

## Screenshots

iPhone | iPad



# Anti-Patterns: Architecture

**Vendor Lock-In.** A system is dependent on a proprietary architecture or data format.

*Solution:* Improve portability, introduce abstractions.

**Design by Committee.** The typical anti-pattern of standardizing committees, that tend to satisfy every single participant, and create overly complex and ambivalent designs ("A camel is a horse designed by a committee").

Known examples: SQL and COBRA.

*Solution:* Improve group dynamics and meetings (teamwork)

# Reinvent the Wheel





# Anti-Pattern: Architecture (2)

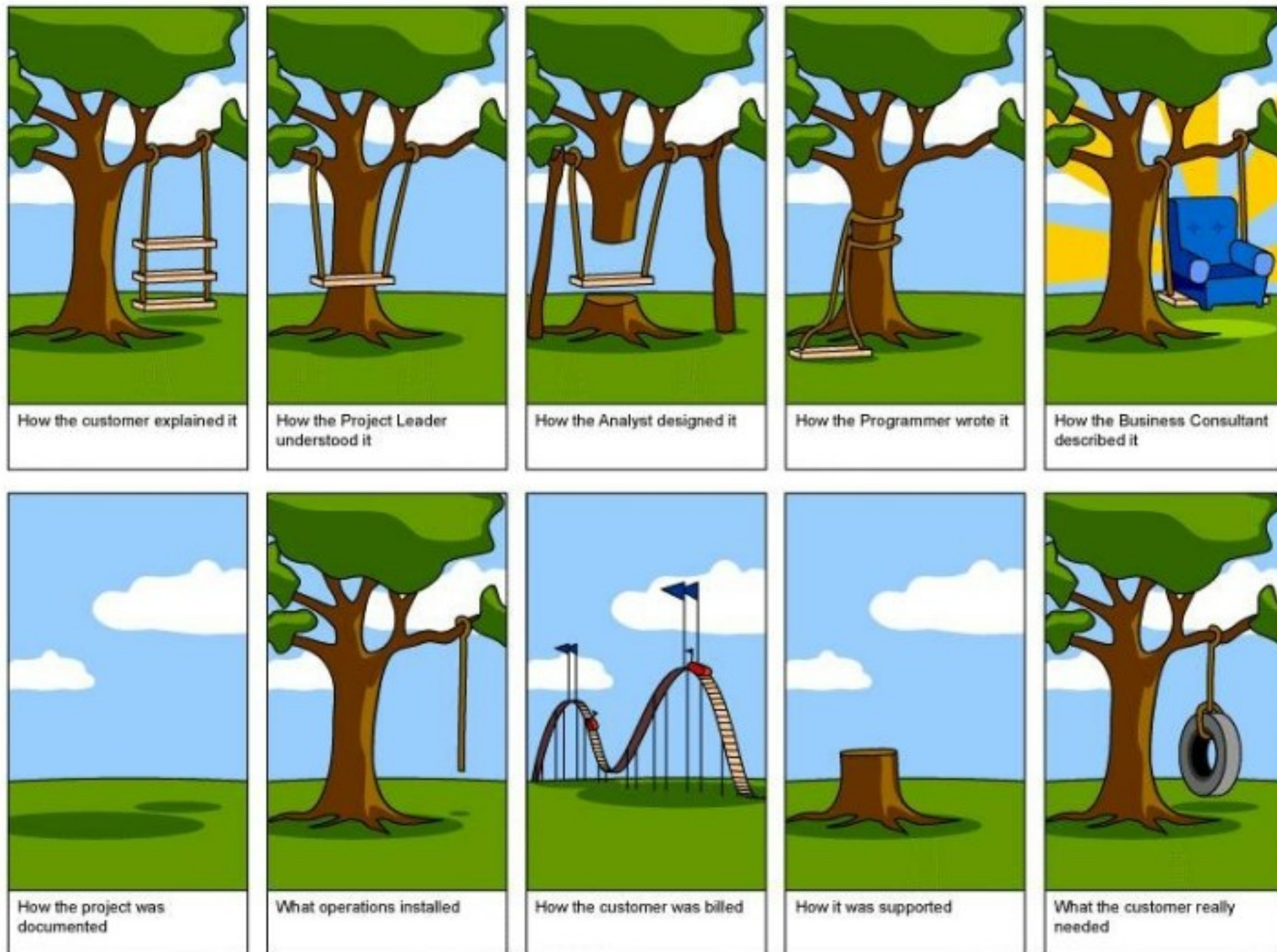
**Reinvent the Wheel.** Due to lack of knowledge about existing products and solutions, the wheel gets reinvented over and over, which leads to increased development costs and problems with deadlines.

*Solution:* Improve knowledge management.

# Intellectual Violence



# Project Mismanagement



# Anti-Patterns: Management

**Intellectual Violence.** Someone who has mastered a new theory, technique or buzzwords, uses his knowledge to intimidate others.

*Solution:* Ask for clarification!

**Project Mismanagement.** The manager of the project is unable to make decisions.

*Solution:* Admit having the problem; set clear short-term goals.

# Other Anti-Patterns

- *Lava Flow* (design changes frequently)
- *Boat Anchor* (a component has no apparent user)
- *Dead End* (a bought component that isn't supported any longer)
- *Swiss Army Knife* (a component that pretends to be able to do everything)



*Two-in-one Gardening Tool*

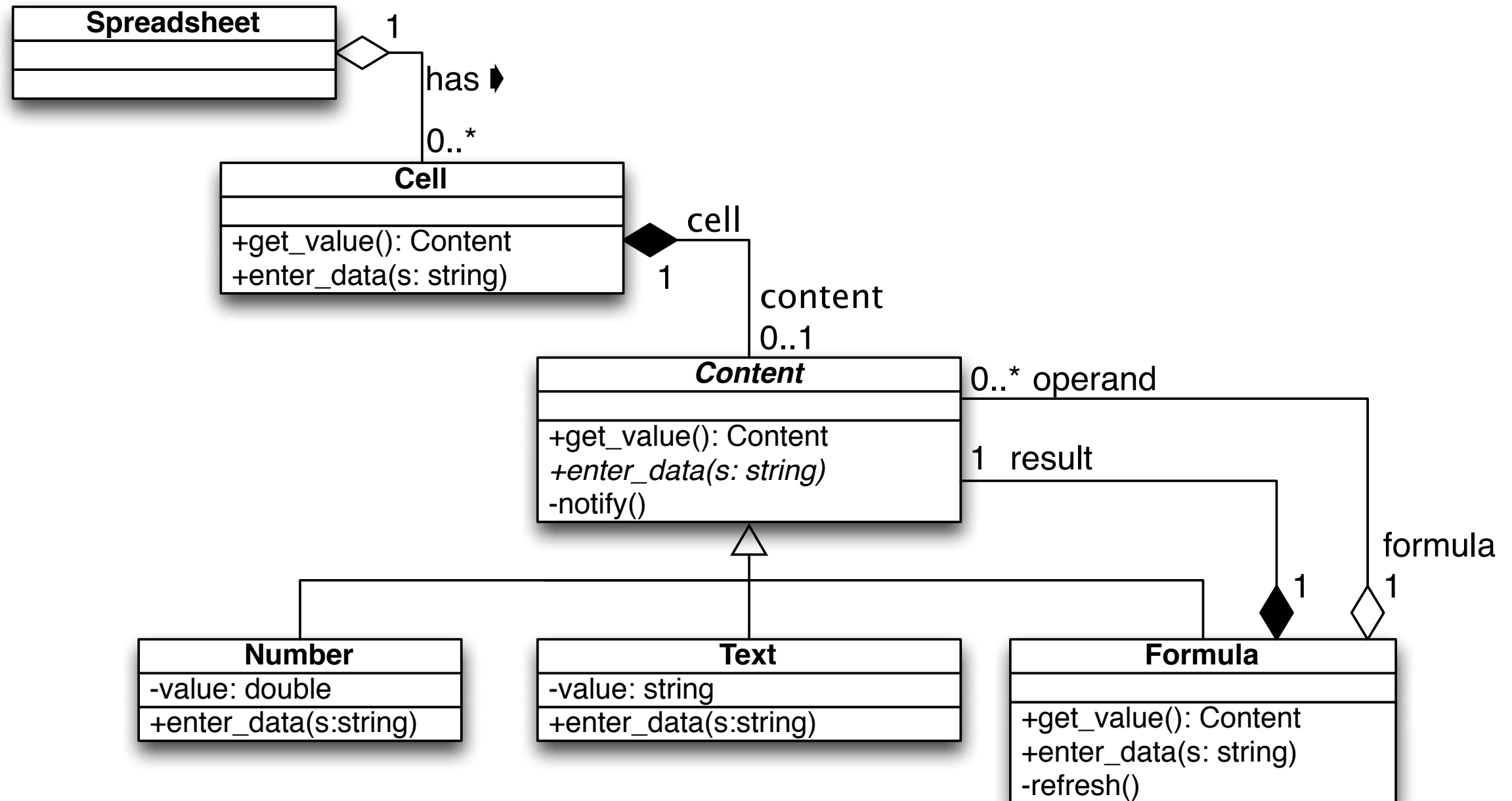
■ *The amateur gardener's do-it-all*

The multi-faceted Swiss Army gardeners is one of the great life-enhancers of modern times. By way of a tribute to the genius of this invention, *Charlie* has never been so far forward, and up in scale. The result is the Two-in-one Gardening Tool. Incorporating a spade of shovel, a hoe, a trowel, a saw, a set of pruning shears and several other gardening implements, all in one moderately compact unit. No more endless trips to and from the shed, and when the day's work is done, you won't have to remember all the tools you've scattered around the garden. However, you may want to think twice about lending one to a nearby neighbor, as everything else will have to go with it.



# Handouts

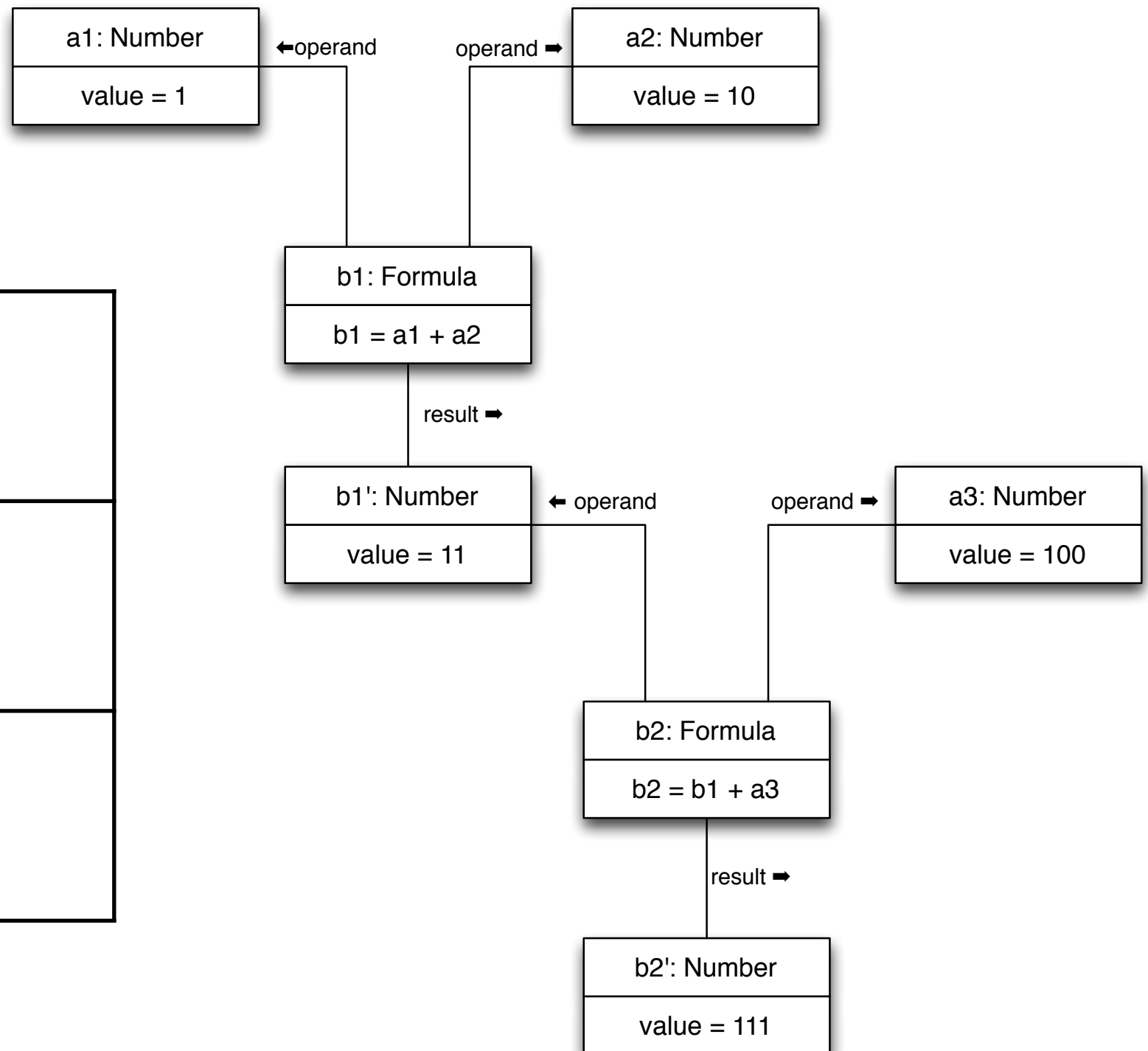
# Object Model





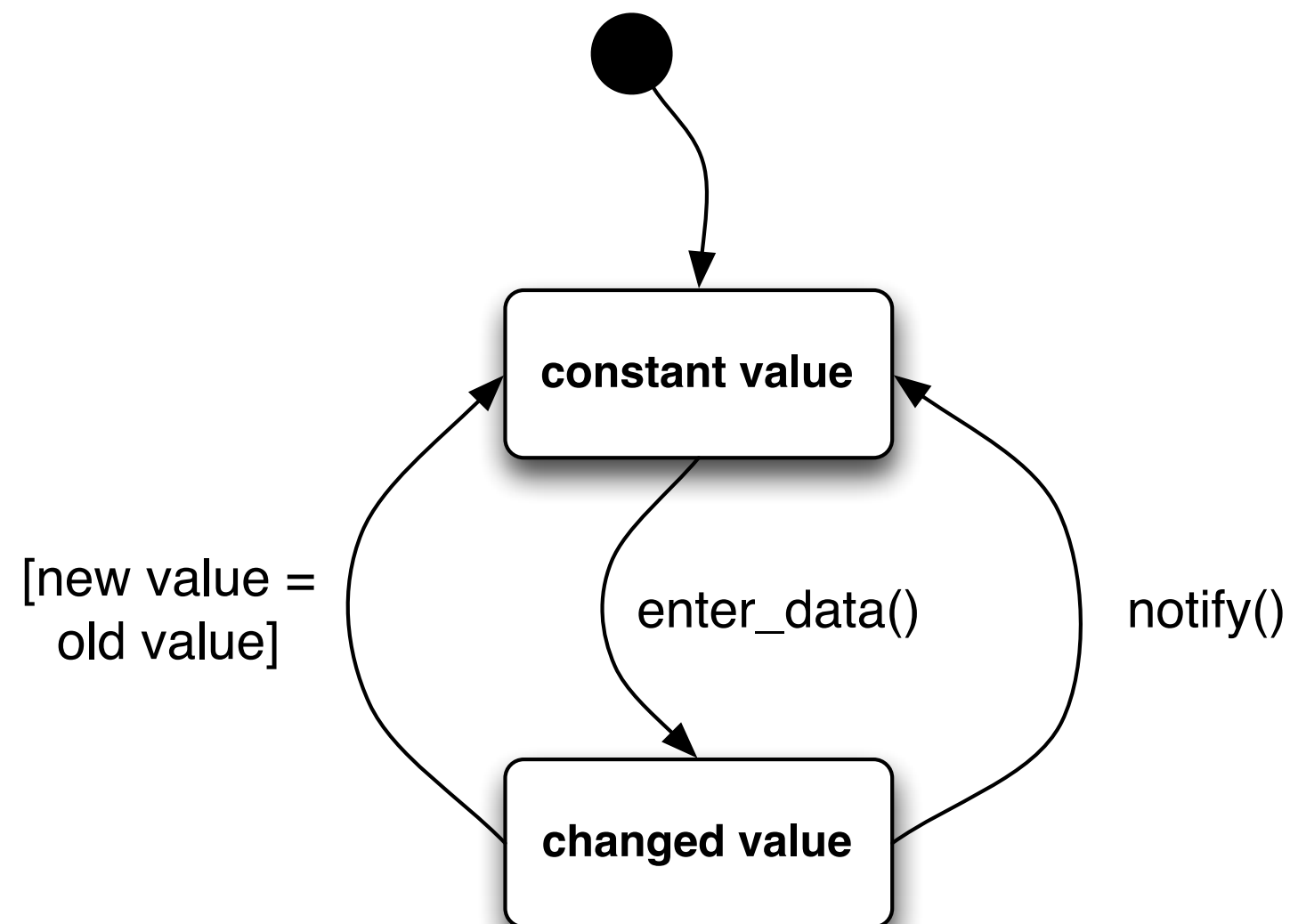
# Relationships between Objects

	A	B
I	I	II
2	10	III
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.

