Objects First With Java
A Practical Introduction Using BlueJ

Designing applications

Main concepts to be covered

- Discovering classes
- CRC cards
- Designing interfaces
- Patterns

Analysis and design

- So far: design classes.
- Now: find proper classes – i.e. analyze the problem and design a solution.
- A large and complex area – start from the scratch for a new software system.
- The verb/noun method is suitable for relatively small problems.
- CRC cards support the design process.

The verb/noun method

- The nouns in a description refer to ‘things’.
  - A source of classes and objects.
- The verbs refer to actions.
  - A source of interactions between objects.
  - Actions are behaviour, and hence methods.

A verbal problem description

“The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers’ telephone numbers.”

Identify nouns and verbs

Cinema booking system
Stores (seat bookings)
Stores (telephone number)

Theatre
Has (seats)

Movie

Customer
Reserves (seats)
Is given (row number, seat number)
Requests (seat booking)

Time

Date

Seat booking

Show
Is scheduled (in theatre)

Seat

Seat number

Telephone number

Row

Row number

Nouns and verbs

- Not an exact method – just an approach to a first design.
- Additional classes and methods might be found later.
- Some of the classes and methods may turn out later to be not needed.
- Understanding/Discribing the desired functionality (to create the “text”) is crucial and complicated!

Using CRC cards

- First described by Kent Beck and Ward Cunningham.
- Deals with next step of design process: explore interactions between classes.
- Each index card records:
  - A *class* name.
  - The class's *responsibilities*.
  - The class's *collaborators* (other classes which this class uses).

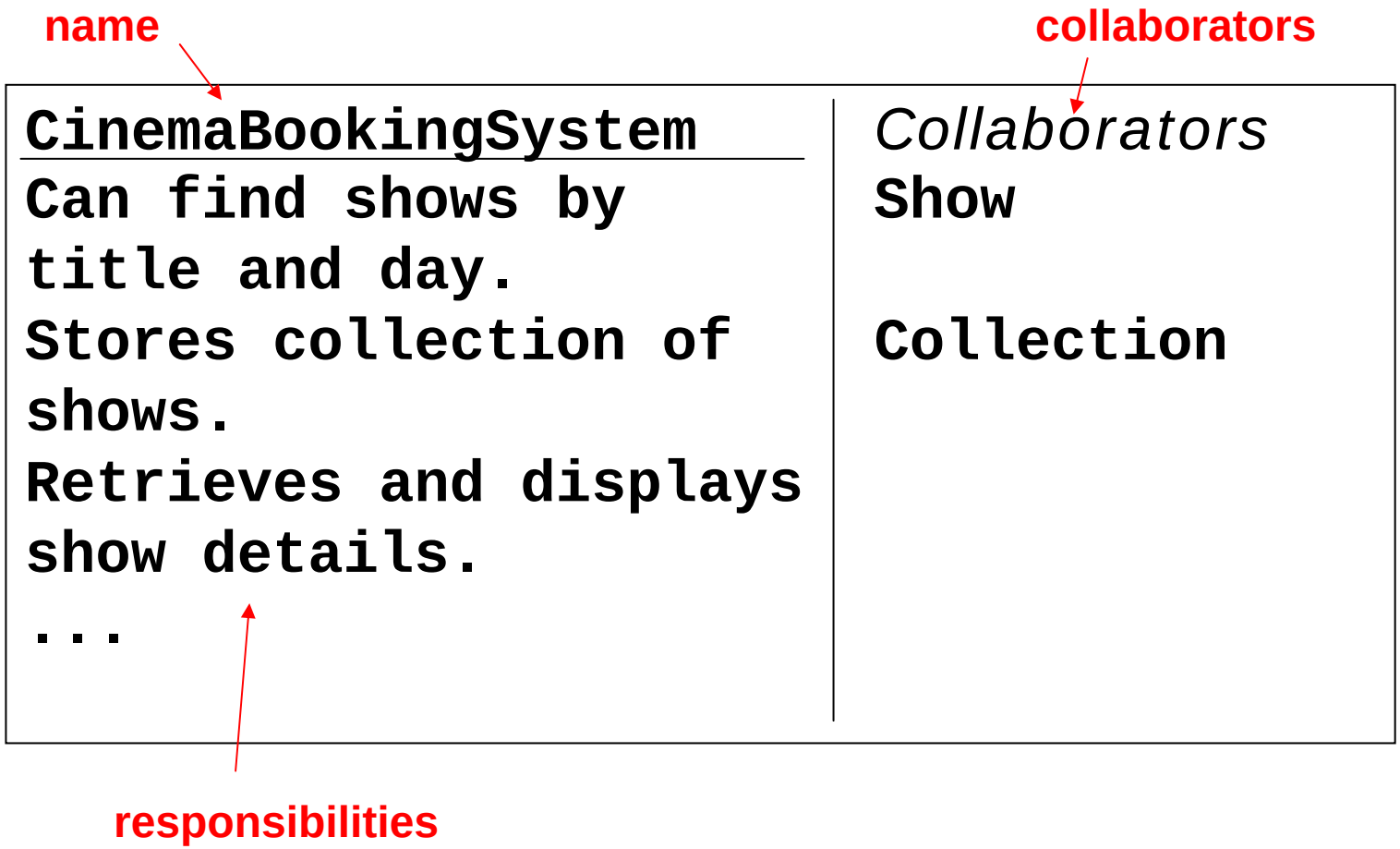
A CRC card

Class name	**Collaborators**
Responsibilities	

Scenarios

- An activity that the system has to carry out or to support.
  - Sometimes known as *use cases*.
- Used to discover and record object interactions (collaborations).
- Can be performed as a group activity (each member “playing” one class).

A partial example



Scenarios as analysis

- Scenarios serve to check whether the problem description is clear and complete.
- Sufficient time should be taken over the analysis.
- The analysis will lead to a consolidated design.
  - Spotting errors or omissions here will save considerable wasted effort later.

Class design

- Scenario analysis helps to clarify the application structure – this prepares the next big step from CRC cards to Java classes.
- Each card maps to a class.
- Collaborations reveal class cooperation and object interaction.
- Responsibilities reveal public methods.
  - And sometimes fields; e.g. “Stores collection …”

Designing class interfaces

- Replay the scenarios in terms of method calls, parameters and return values.
- Note down the resulting signatures.
- Create outline classes with public-method stubs.
- Careful design is a key to successful implementation – and typically takes at least as much time.

Documentation

- Write class comments.
- Write method comments.
- Describe the overall purpose of each.
- Documenting now ensures that:
  - The focus is on *what* rather than *how*.
  - That it doesn't get forgotten!
- Force yourself to do it!!! (You will NOT do it later, despite all good pledges!)

Cooperation

- Team-working is likely to be the norm, not the exception (whether you like it or not).
- Documentation is essential for team working.
- Clean O-O design, with loosely-coupled components, also supports cooperation.

Prototyping

- Supports early investigation of a system.
  - Early system understanding.
  - Early problem identification.
- Incomplete components can be simulated.
  - E.g. always returning a fixed result.
  - Avoid random behaviour which is difficult to reproduce.

Software growth

- Waterfall model – a classic!
  - Analysis
  - Design
  - Implementation
  - Unit testing
  - Integration testing
  - Delivery
- In case of phase failure, go back one step.
- But: no provision for iteration (loops).

Waterfall model

- Main drawbacks:
  - Designers / developers must understand the system's functionality in detail from the start.
  - System must not change after delivery.

Iterative development

- Use early prototyping.
- Frequent client interaction.
- Iteration over:
  - Analysis
  - Design
  - Prototype
  - Client feedback
- That is: software cycle, stepwise refinement.
- A growth model is the most realistic.

Using design patterns

- Inter-class relationships are important, and can be complex.
- Some relationships recur in different applications.
- Design patterns describe common problems and general solutions.
- They
  - help clarify relationships;
  - provide and document good solutions;
  - promote reuse of (class) structures.

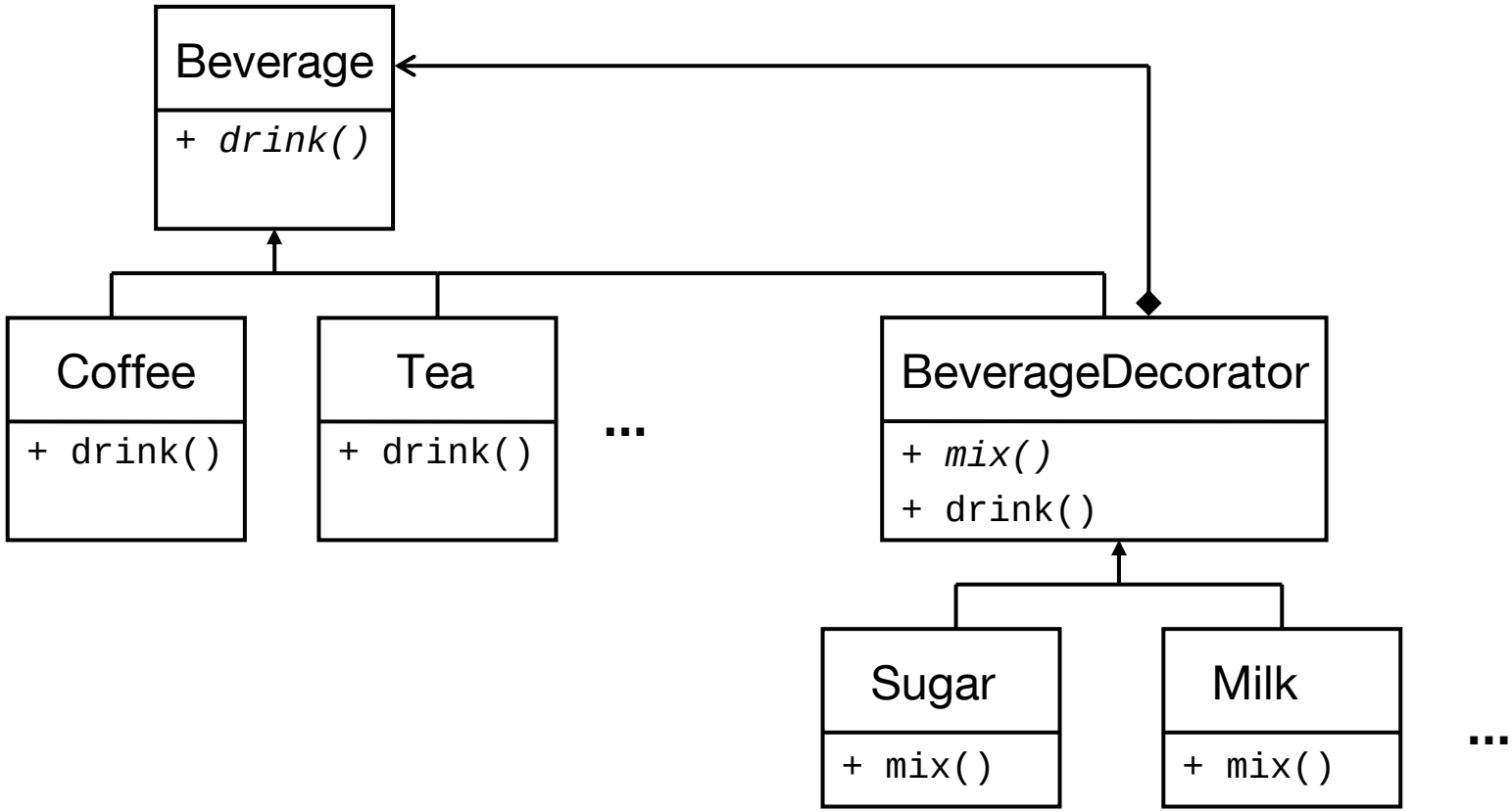
Pattern structure

- A pattern name.
- The problem addressed by it:
  - Intent, motivation, applicability.
- How it provides a solution:
  - Structures, participants, collaborations.
- Its consequences.
  - Results, trade-offs.

Example 1: Decorator

- Augments the functionality of an object.
- Decorator object wraps another object.
  - The Decorator has a similar interface.
  - Calls are relayed to the wrapped object ...
  - ... but the Decorator can interpolate additional actions.
- Example: `java.io.BufferedReader`
  - Wraps and augments an unbuffered `Reader` object.
- Similar to inheritance, but: new behaviour only at runtime for *individual* objects.

Example 1: Decorator (II)



Example 2: Singleton

- Ensures only a single instance of a class exists.
  - All clients use the same object.
- Constructor is private to prevent external instantiation.
- Single instance obtained via a static `getInstance` method.
- Example: `Canvas` in *shapes* project.

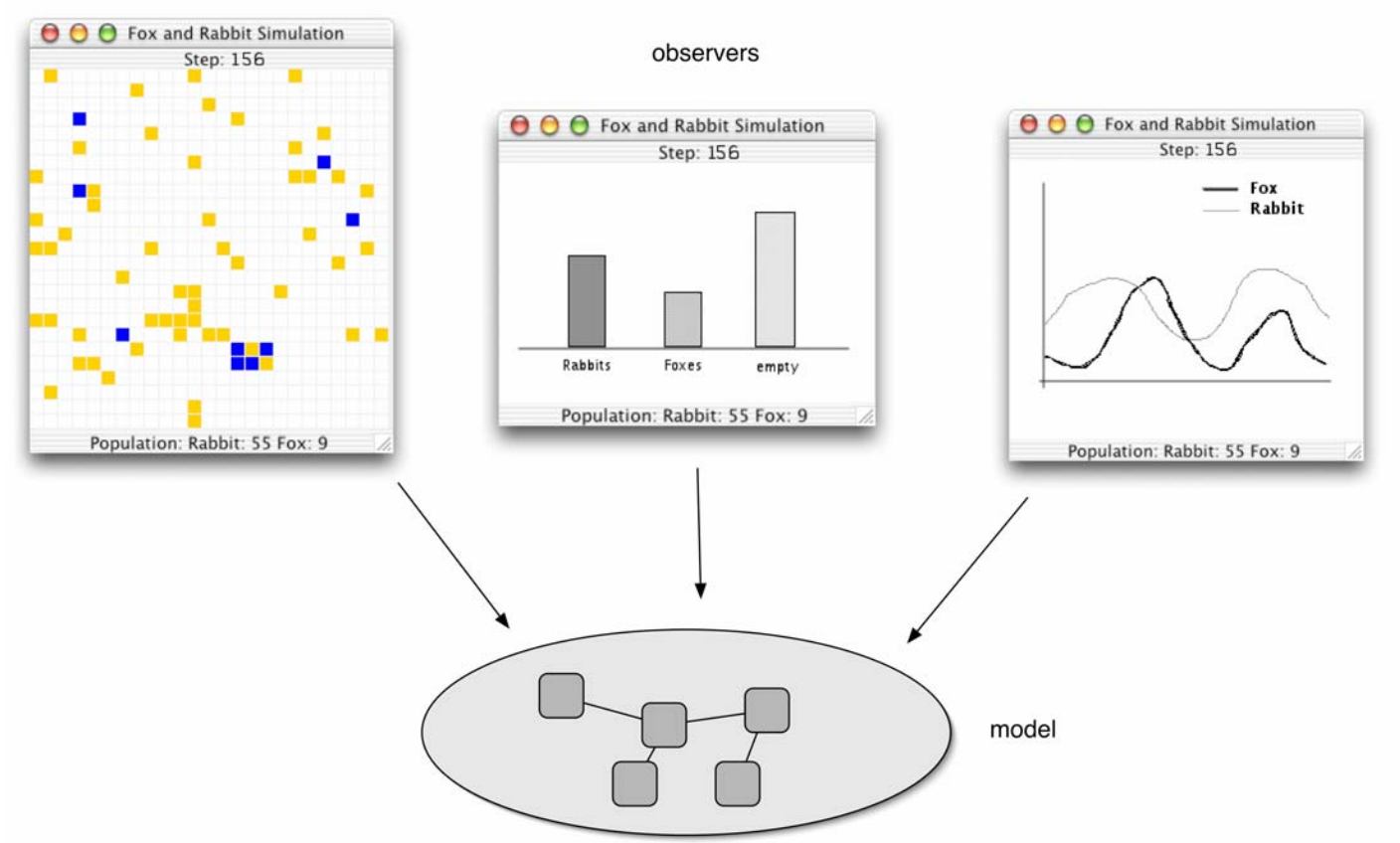
Example 3: Factory method

- A creational pattern.
- Clients require an object of a particular interface type or superclass type.
- A factory method is free to return an implementing-class object or subclass object.
- Exact type returned depends on context.
- Example: `iterator` methods of the Collection classes.

Example 4: Observer

- Supports separation of internal model from a view of that model.
- Observer defines a one-to-many relationship between objects.
- The object observed notifies all Observers of any state change.
- Example `SimulatorView` in the *foxes-and-rabbits project*.

Observers – an example



Review

- Class collaborations and object interactions must be identified early.
  - CRC analysis supports this.
- An iterative approach to design, analysis, and implementation can be beneficial.
  - Regard software systems as entities that will grow and evolve over time.

Review

- Work in a way that facilitates collaboration with others.
- Design flexible, extendible class structures.
  - Being aware of existing design patterns will help you to do this.
- Continue to learn from your own and others' experiences.