Hugbúnaðarverkefni 1 / Software Project 1

12. Design Patterns

HBV501G – Fall 2018

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz Prep

- Please prepare a scrap of paper with the following information:

- ID: ____@hi.is   Date: ______
- a) ________________________
- b) ________________________
- c) ________________________
- d) ________________________

- During class, I'll show you questions that you can answer very briefly
  - No elaboration necessary
- Hand in your scrap at the end of class
- All questions in a quiz weigh same
- All quizzes (ca. 10 throughout semester) have the same weight
  - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam

Recap: Assignment 4: Code Review – Schedule

- ✓ By **Sun 11 Nov,** make your **project artifacts** available to your partner team:
  - Your project vision and design model from Assignments 1 & 3 (incl. fixes of severe issues)
  - A current snapshot of your source code (does not need to be the finished product)
- ✓ Take **one week** to **review** the other team’s code and **document** your findings:
  - Comment on clarity of design, quality of implementation, readability of code, tech choices
  - State what you like and make suggestions for improvements
- By **Sun 18 Nov,** submit your **review report** to Ugla
  - 1-2 pages in PDF
- On **Thu 22 Nov, discuss** your findings with the other team and your tutor.

- **Grading criteria:**
  - Quality of constructive feedback on other team’s design and code (80%)
  - Design and technology issues identified in your own system (10%)
  - Coding style / clarity issues identified in your own system (10%)

Assignment 5: Final Product – Schedule

- On **Thu 29 Nov**, demonstrate and explain your product in class:
  1. **Product:** What does your system do? Demonstrate the key use cases of your product.
  2. **Architecture:** How does your product work? Explain architecture & key design decisions.
  3. **Process:** How did you build the system? Review and discuss challenges you faced.
  - Demonstrate #1 live; prepare a few slides for #2 and #3 (the order of #1–#3 is up to you).
  - Please strive to be present for the whole time (08:30-11:30) that Thursday
    - so you can see other teams' work and learn from their experiences
    - so your classmates have an audience as well

- By **Sun 2 Dec**, submit in Ugla:
  - The **slides** of the presentation you gave in class
  - The **source code** of your final product, including everything required to build and run it, i.e.:
    - All server- and client-side code
    - SQL statements to create the required database schema and test data
      - unless auto-generated by the Java Persistence API
    - Any necessary instructions for building and running the project
      - e.g. Maven/Gradle scripts or manual instructions for obtaining 3rd party components (e.g. the database)

Assignment 5: Final Product – Presentation

- **On Thu 29 Nov**, each team presents their product to their tutor and his other teams
  - 20 minutes per presentation (~5 minutes for each of the following parts, 5 minutes for questions)
- Your presentation must cover the following parts (in an order of your choice):
  - **A live demonstration of your final software**
    - Should include product’s key use cases
  - **An overview of your system architecture**
    - What are the key components, and how do they communicate?
    - What aspects are client and server responsible for?
    - How are you storing and accessing data?
    - Any particular aspects of your design you would like to highlight?
  - **A retrospective on your project work**
    - What went well? What difficulties did you encounter?
    - How did you plan to structure / manage your work? How did that turn out?
    - What would you do differently next time?
    - How would you avoid any difficulties you encountered?
- You can choose which teammates give which parts of the presentation.

No reading off slides
or note cards!

Recap: Presentation Tips

- **Engage your audience**
  - Speak freely, don't read off slides/cards
  - Interact with the audience and the slides
  - Excite yourself about the topic
- **Tailor your talk to your audience**
  - What do they know already?
  - What don't they know yet?
  - What do they need and want to know?
- **Be clear**
  - High-contrast colors, readable font size
  - Keep your slides simple
    - just a few text bullets
    - even better: clear illustrations
- **Use a style you're comfortable with**
  - Visualize what you want to explain in a way that's natural for you to talk about
    - Slides / clean hand-drawn sketches / …
- **Be prepared**
  - Practice to get a feeling for your talk
    - Pacing, time limits vs. amount of material
  - Rehearse any sticky parts
    - Intro, wrap-up, topic transitions, sketches
  - Ensure the live demo works on your PC
- **Relax**
  - Avoid stress of switching computers
  - Don't try to memorize it all – get fluent in the topic, and you'll be fluent in your talk

Assignment 5: Grading Criteria

1. **Product Demonstration (80%)**
   - ✓ Software runs smoothly
   - ✓ Key use cases are working
2. **Architecture & Design (10%)**
   - ✓ Architecture described, illustrated clearly
   - ✓ Key design decisions described and illustrated clearly
   - ✓ Room for improvement discussed critically
3. **Software Process (10%)**
   - ✓ Design & development process over the course of the semester described clearly
   - ✓ Handling of technical / methodical / collaboration challenges discussed critically

- **Submitted code**
  - Not graded explicitly, but checked for conformity with presented prototype and architectural requirements
    - Inconsistency of submitted code with presented product may lead to reduction of Product Demonstration grade
    - Completely messy code or code not using server-side Spring MVC components may lead to reduction of Architecture & Design grade
- **Presentation Grade**
  - ✓ Presenter shows initiative, explains things clearly, shows understanding of the project, can answer questions competently

Preview: Final Exam

- **Date & Time: 13 Dec 2018, 13:30-16:30**

- **Focus:** Understanding of software engineering concepts and methods

- **Scope:** Lecture slides (i.e. contents of Námsefni folder)
  - Note: The spoken part is relevant too!

- **Style:** Written exam
  - Write into given spaces on exam sheets
  - Mark exam sheets only with your exam number, *not* your name

- **Weight:** 30-70% of final course grade

- **Tools:**
  - One sheet of handwritten material allowed
    - i.e. blank A4 sheet with only your own ink (double-sided use ok)
    - no photocopied notes or printed material
  - Dictionary allowed (in book form)
  - No electronic devices allowed
- **Questions:**
  - Explain / argue / discuss / calculate...
  - No optional questions (except quiz result)
  - But answers that exceed expectations can make up for deficiencies elsewhere
- **Answers:**
  - in English, in your own words
  - short paragraphs of whole sentences
  - possibly small models

Teaching Assistants Wanted

- I am looking for TAs for next semester’s courses:
  - **HBV401G Software Development**
  - **HBV601G Software Project 2**
- If you are interested
  or know someone who is,
  please contact me at **book@hi.is**.

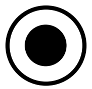
- **Tasks**
  - Advising student teams
  - Scoring assignments
- **Opportunities**
  - Help to shape the course
  - Brush up your CV
- **Requirements**
  - Successful completion of respective course
  - Ability to advise and evaluate student teams

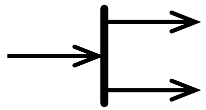
Quiz #9 Solution: UML Diagrams

- **Decide which of the following UML diagram types (a-d) are (1) <u>structure</u> and which are (2) <u>behavior/interaction</u> diagrams:**

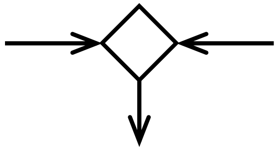
a) UML state machine diagram **(2)**
b) UML package diagram **(1)**
c) UML sequence diagram **(2)**
d) UML object diagram **(1)**

- **What is the meaning (5-8) of the following UML symbols (e-h)?**

e)  **(6)**

f)  **(8)**

g)  **(5)**

h)  **(7)**

5. Action
6. Final state
7. Merge node
8. Fork pseudo state

Design Patterns

see also:

- Larman: Applying UML and Patterns, Ch. 26
- Freeman, Robson: Head First Design Patterns

Recap: Domain Models vs. Design Models

- We create **domain models** in order to **understand the application domain**
  - So we can talk to the business stakeholders about their requirements on their own terms
  - ➢ Key challenge is understanding all of the domain's entities and their relationships, i.e. **not missing anything important / valuable / risky**
  - Supported by: Strategies for identifying classes, guidelines for expressing relationships, value-oriented perspective

- We create **design models** in order to express **how our solution shall work**
  - So we can talk to fellow developers about what the best technical solutions would be
  - ➢ Key challenge is engineering a solution that enables efficient implementation, execution and maintenance, i.e. **creating an efficient design**
  - Supported by: Object-oriented design guidelines, **design patterns**

Focus of this chapter

From Design Heuristics to Design Patterns

- The **design heuristics** discussed in Chapter 8 of this course are low-level guidelines for the design of individual classes.
  - Examples: High Cohesion, Low Coupling, Protected Variations, Indirection, …

- **Design patterns** are proven ways of satisfying the above principles while solving common design problems. They often involve multiple collaborating classes.
  - Examples: Singleton, Factory, Adapter, Proxy, … *(→ HBV401G, Chapter 12)*

Recap from HBV401G: Design Patterns

- Solving new problems by applying solutions that have been proven in the past
  - How is the new problem similar to a previous problem?
  - How was the previous problem solved?
  - Is that solution generalizable?
  - How can the general solution be applied to the concrete problem?

- Description of the generic solutions as **patterns**
  - Originally a concept from the domain of architecture, only later applied to software

- Advantages:
  - **Pattern collections** (“pattern languages”) bundle comprehensive domain experience
  - **Explicit description** of characteristics and impacts of design decisions reduces risks
  - **Common vocabulary** simplifies communication and prevents misunderstandings

Recap: Design Patterns in OOD

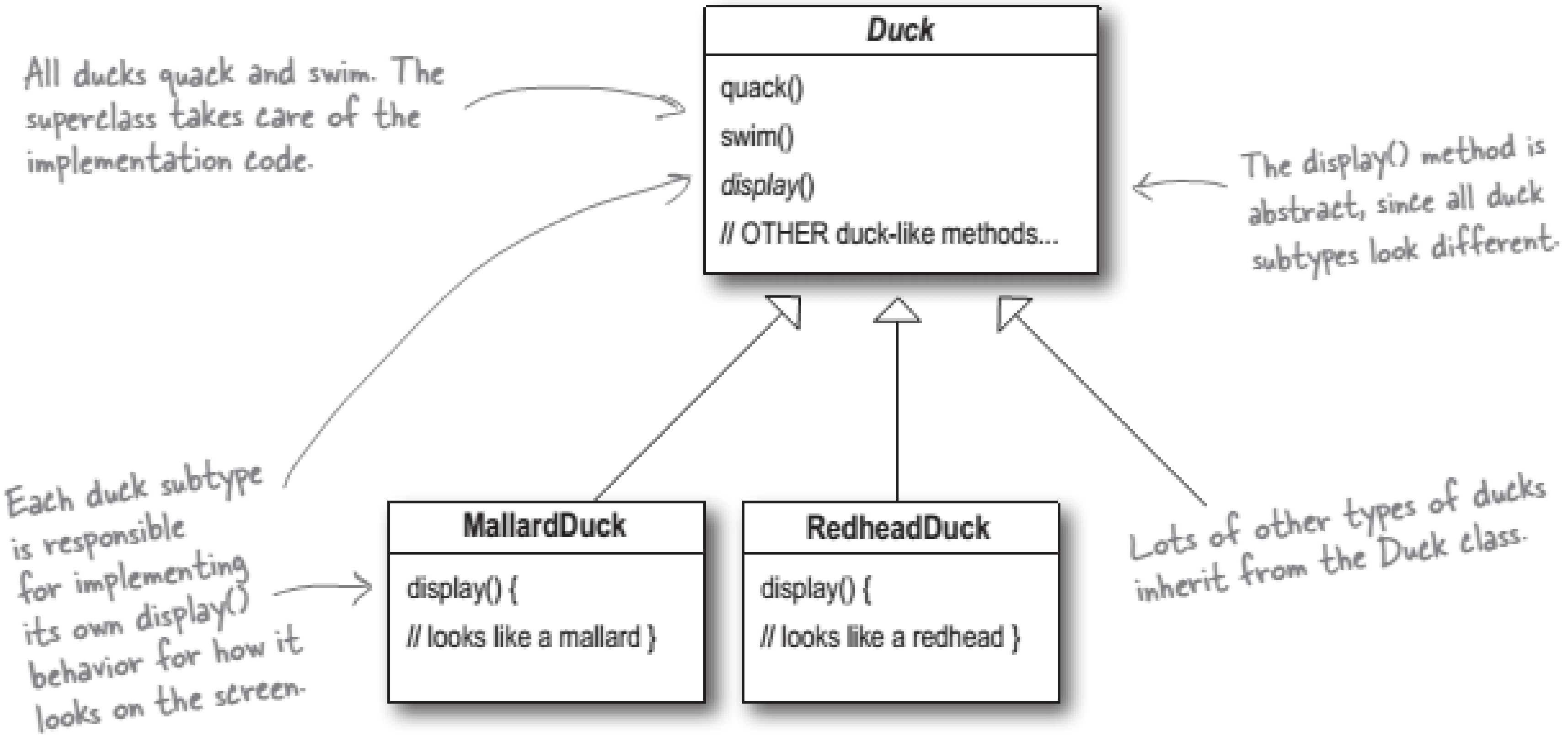
In object-oriented design, there are proven solutions for a number of frequently found requirements or modeling challenges, the so-called **design patterns**:

- For creating particular object structures
  - e.g. Singleton, Factory, Composite, ...
- For the interaction of objects with particular behaviors
  - e.g. Iterator, Observer, Transaction, ...
- For the cooperation and decoupling of system components
  - e.g. Adapter, MVC, Strategy, ...

➢ Goal: Effective use of OO concepts on the **component level**

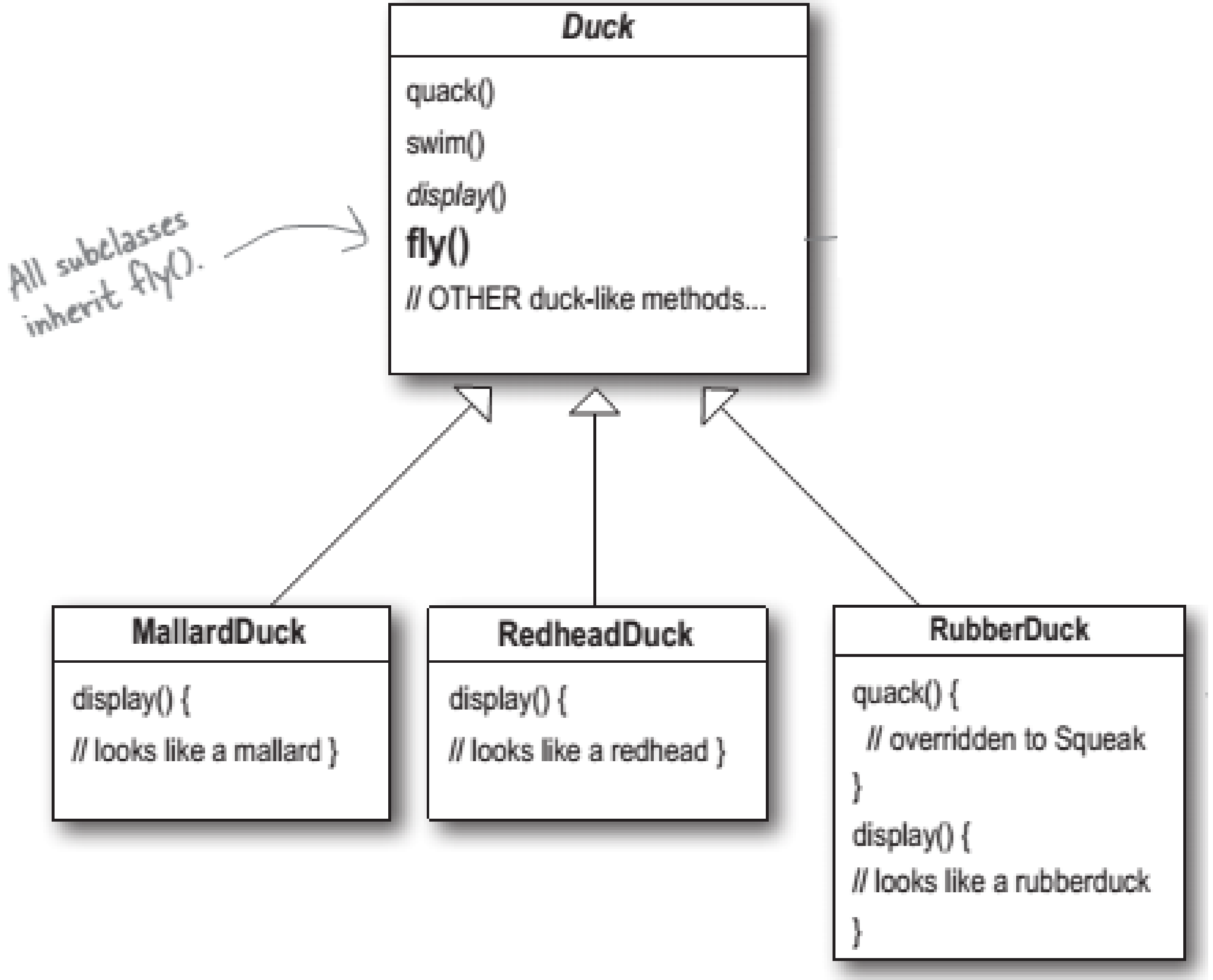
Strategy Pattern
Motivation: Modeling "Similar" Classes

- Imagine we are supposed to simulate the behaviors of different kinds of ducks.
- Our usual approach: Express generic and specific class properties with inheritance.



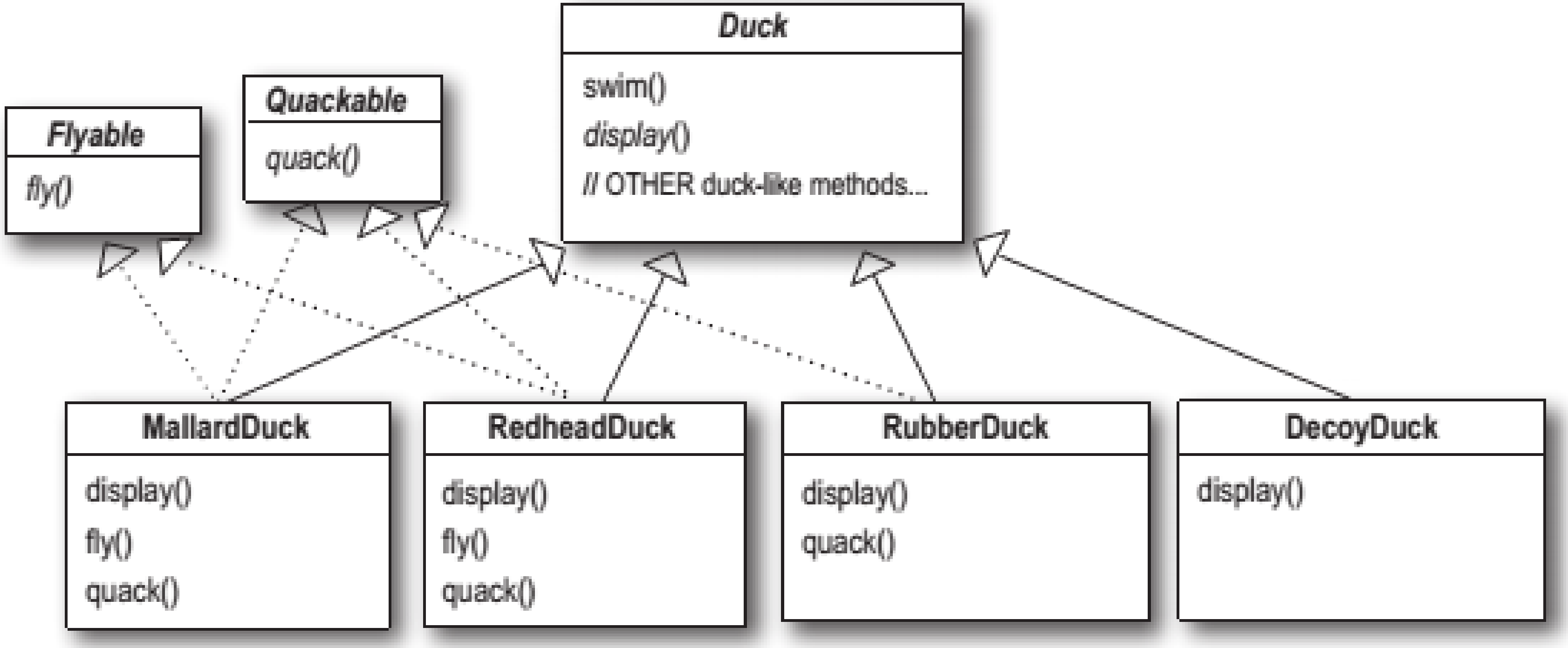
Strategy Pattern
Motivation: Inheritance May Not Be Precise Enough

- If we want to add a new behavior to all subclasses, we just need to add it to the superclass.
  - here e.g.: giving all ducks the ability to fly
- But what if the behavior is not relevant for all the subclasses?
  - here e.g.: some ducks are unable to fly, some are unable to quack



Strategy Pattern
Motivation: Interfaces as an Alternative?

- We could
  - pull the properties that are only supported by some subclasses out of the superclass, and
  - model them as interfaces that subclasses can elect to implement
- However, then
  - the subclasses cannot inherit the shared behavior
  - but all need to implement it themselves ☹



The Big Picture
Implementing Shared Behaviors

- **Where to implement shared behaviors to ensure low coupling of classes?**
- In our example:
  - All ducks can **swim**, and they all do it in the same way.
    - ➢ The one implementation is placed in the superclass, and inherited by all subclasses.
  - All ducks can be **displayed** in the simulator, but they *all* look differently.
    - ➢ The abstract method in the superclass ensures that all ducks will be displayable, but how they are displayed is implemented individually in each subclass.
  - All ducks can **quack**, but *some* do it in different ways.
    - ➢ The most common implementation is placed in the superclass and inherited by all subclasses, which can override it with specialized implementations if desired.
  - Some ducks can **fly**, but some can not (and should not look like they could).
    - Inheriting from a superclass would confer undesired abilities to some subclasses ☹
    - Implementing an interface would require re-implementing the ability in each eligible subclass ☹
    - ➢ Use composition and delegation instead of inheritance to add behavior to classes selectively.

Encapsulation of Change as a Driver of Good Design

One of the most fundamental modular / object-oriented design principles:

Encapsulate what varies.

- If you can foresee that something will change…
  - an algorithm, a data structure, a technical component, a business process, etc.
- …encapsulate that part of your system
  - so other parts of the system will remain unaffected by any internal changes.
  - Various encapsulation techniques exist on different levels:
    - **Programming language:** Methods, classes, visibilities
    - **Class structure:** Abstract supertypes (i.e. `abstract` classes or `interfaces`), polymorphism
    - **Class collaboration:** Object-oriented design patterns
    - **System architecture:** Component-based design, layered architectures, communication protocols

The Big Picture

Encapsulation of Change as a Driver of Good Design

Two important corollaries to the encapsulation principle:

- **As a module user: Program to an interface, not an implementation.**
  - When you are working with something encapsulated, treat it as a black box and make no assumptions about how it works inside.
    - i.e. do not rely on knowledge/assumptions about things like data structures, sorting orders, side effects, thread safety etc., as they may change (unless they are explicitly specified)

- **As a module provider: You can change an implementation, but never an interface.**
  - Changing an interface will break any outside code relying on it.
    - Changes include adding abstract methods to supertypes (i.e. classes or Java `interfaces`), changing method signatures, and even changing a method's documentation if it promises certain properties such as sorting order, thread safety etc.

Strategy Pattern

Solution: Encapsulating What Varies

- The behaviors that vary between duck types are pulled into individual classes encapsulating each behavior.

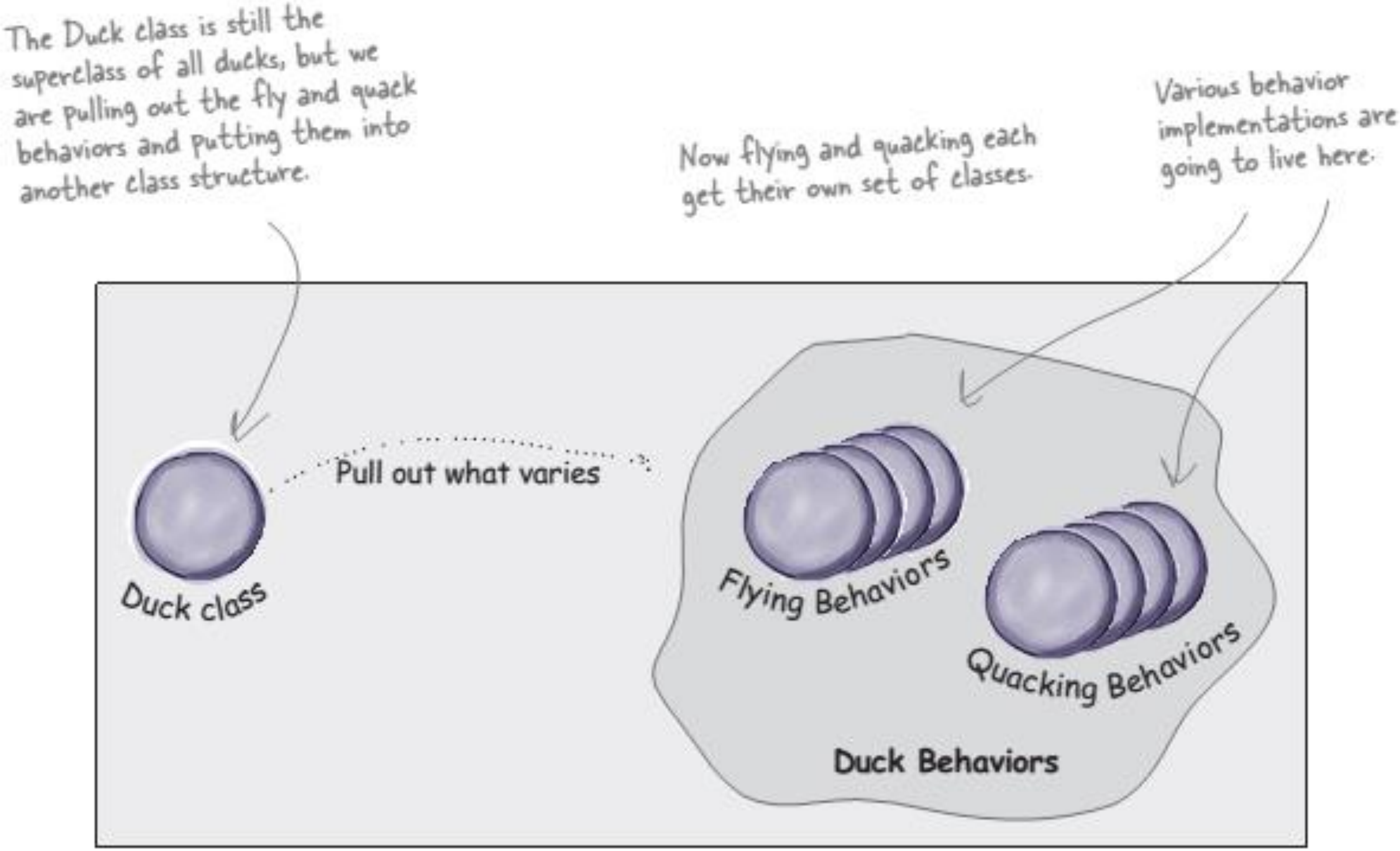
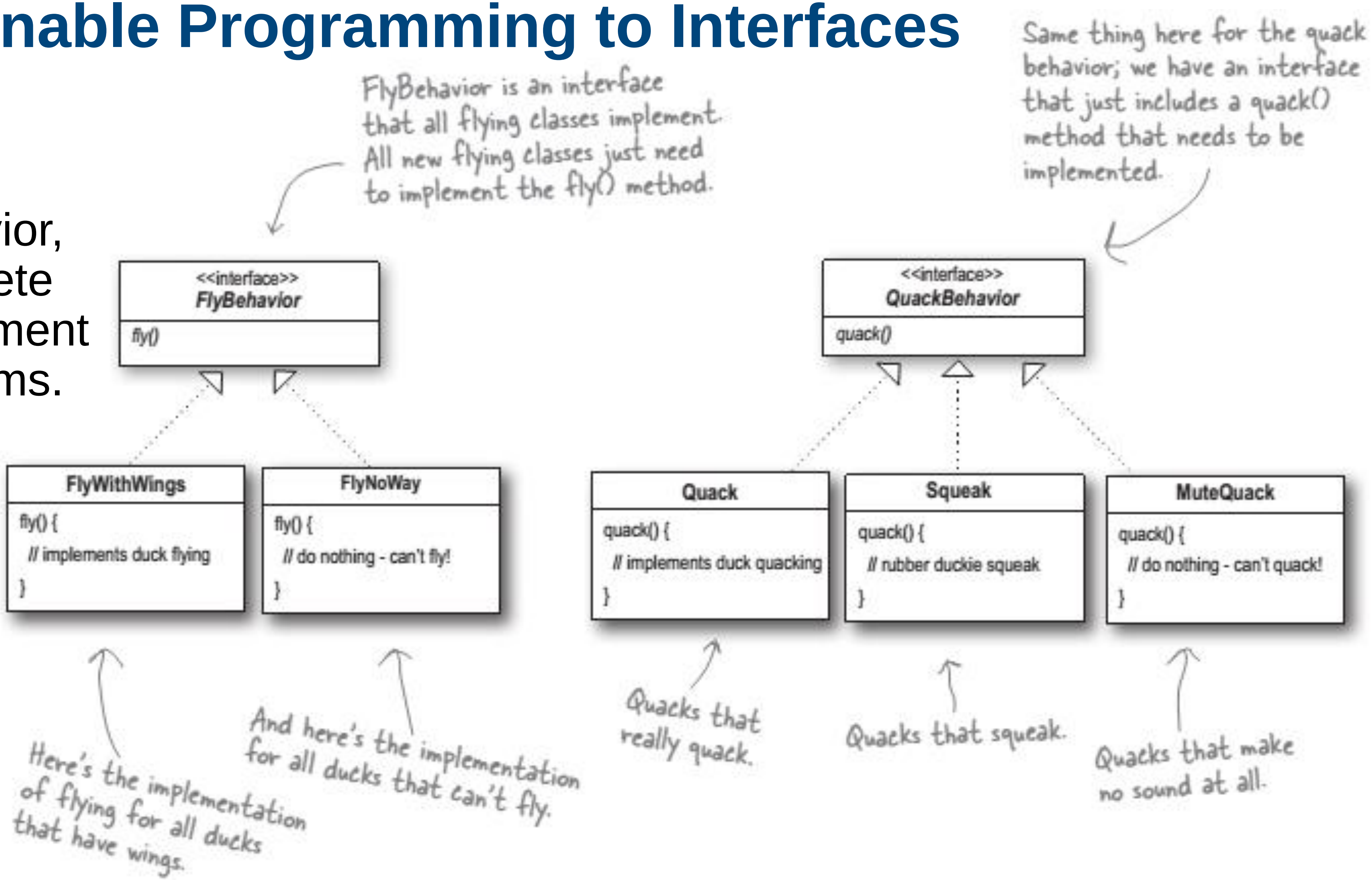


Figure: Freeman/Robson, Head First Design Patterns

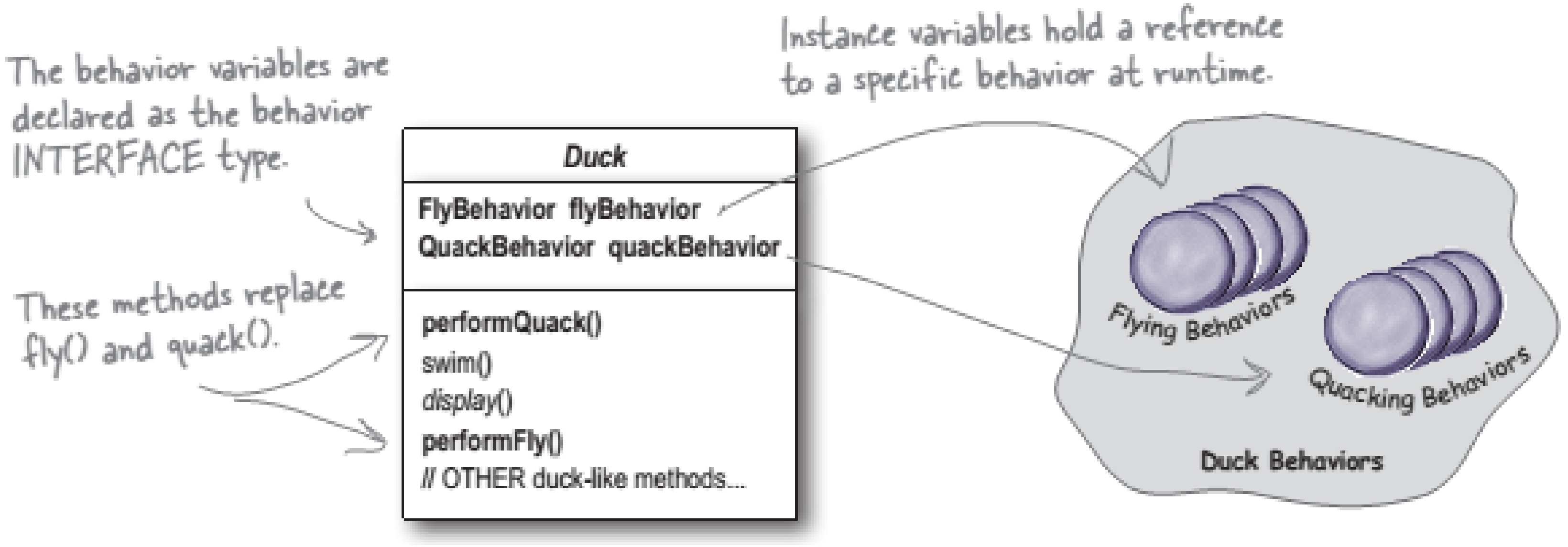
Strategy Pattern
Solution: Enable Programming to Interfaces

- The interface describes the general behavior, and the concrete classes implement its specific forms.



Strategy Pattern
Solution: Program to Interfaces

- Duck instances can now **delegate** the actual flying and quacking behavior to specialized objects implementing particular variations of that behavior.
- And we can even configure which behavior to use at runtime!



Strategy Pattern
Example Implementation

- Implementation of specific behavior classes

```
public interface FlyBehavior {
   public void fly();
}
```

The interface that all flying behavior classes implement.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
      System.out.println("I'm flying!!");
   }
}
```

Flying behavior implementation for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
      System.out.println("I can't fly");
   }
}
```

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
      System.out.println("<< Silence >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
      System.out.println("Squeak");
   }
}
```

Strategy Pattern
Example Implement.

- Implementation of generic client class

```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {
    }

    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.

Strategy Pattern
Example Implementation

- Implementation and configuration of a particular client type

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
         System.out.println("I'm a real Mallard duck");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

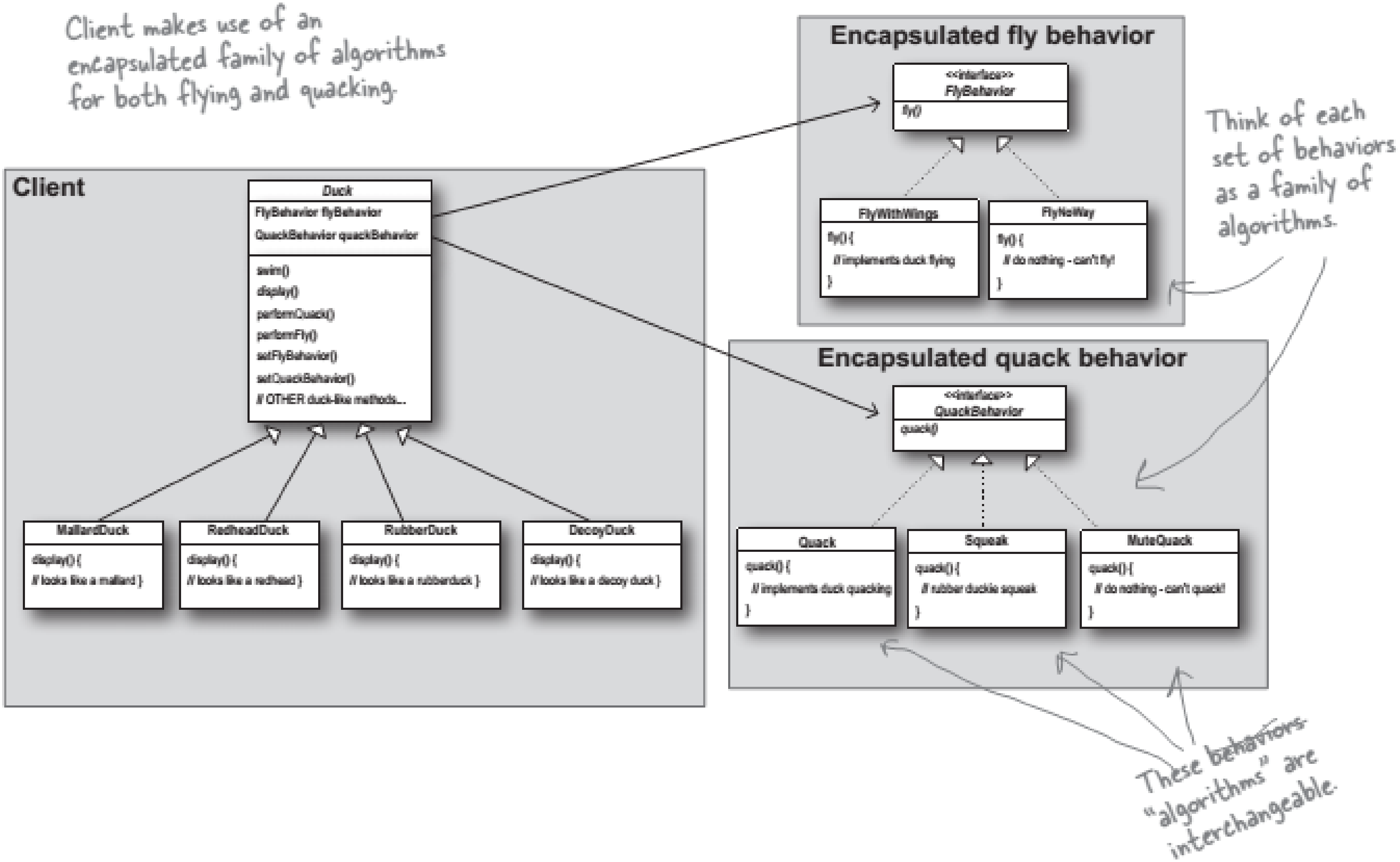
And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

Strategy Pattern
Summary

- The **Strategy pattern**
  - defines a family of algorithms,
  - encapsulates each one,
  - and makes them interchangeable.

- The Strategy pattern lets the algorithm vary independently from clients that use it.



In-Class Quiz #10: Encapsulation and Coupling

How (1-4) should shared behaviors be implemented to ensure low coupling of classes in these scenarios (a-d)?

a) Several classes share a behavior, but some perform it in different ways than others.

b) Several classes share a behavior, but they all perform it in different ways.

c) Among several classes, some share the same behavior, but some do not (and should not look as if they did).

d) Several classes share a behavior, and they all perform it in the same way.

1. The behavior is added to those classes who should perform it by composition and delegation.
2. The behavior is implemented in a superclass, and inherited by all subclasses.
3. The most common form of the behavior is implemented in the superclass and inherited by all subclasses, which can override it with individual implementations if desired.
4. An abstract method in a superclass ensures that all classes will provide the behavior, but its form is implemented individually in each subclass.