Class and Interface Design

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP C03

Design of

- Use-client interfaces
- Class implementations
- Inheritance interfaces

Objects and Classes

- The **modeling** perspective
  - An object is the representation of a phenomenon from a domain
  - A class is a description of the commonalities of similar objects
- The **technology** perspective
  - An object is an encapsulation of some program state
  - A class is the implementation of how to change that state
- Here, we will focus on the technology perspective

Classes and Interfaces

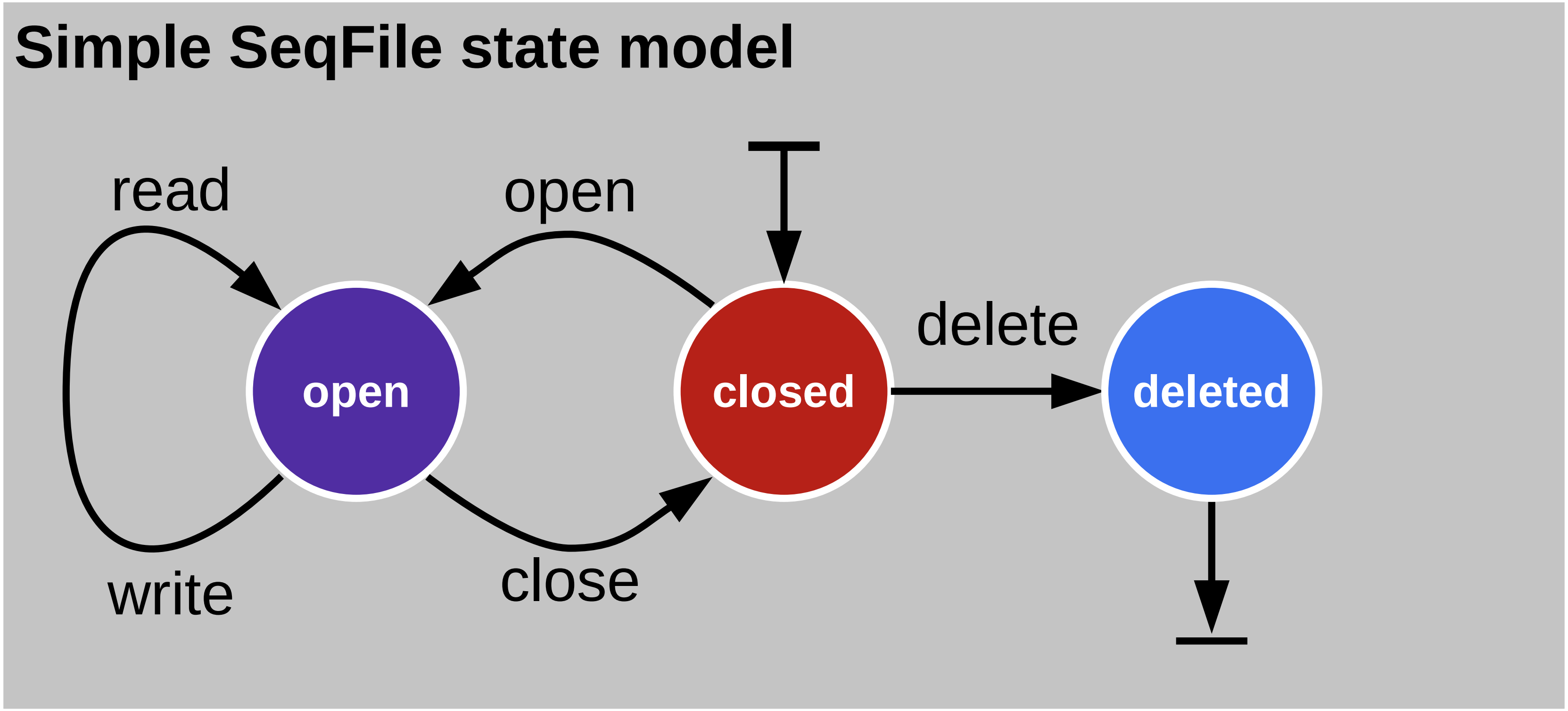
- **Interface**
  - The abstract description of some object behavior
    - Includes an abstract state model and state transitions
  - To be implemented by abstract and/or concrete classes
- **Abstract class**
  - A partial implementation of an interface’s behavior
    - Sets up algorithmic scaffolding for concrete subclasses
  - Implements an interface, to be extended by subclasses
- **Implementation class**
  - A concrete (complete) implementation of an interface’s behavior
    - Includes implementation state (Java fields)
  - Directly implements an interface or extends an abstract class

Java Classes and Interfaces

- A Java interface is an interface
- A Java class has an interface
  - This interface is conceptually separate from the implementation
  - Cf. C++ where classes are split in header and implementation files
- A Java class can be an abstract class
  - By declaration using “abstract”
  - By not providing a public constructor
  - By having abstract methods

Abstract State Model

- An interface expresses an abstract state model



- You cannot (should not) call delete on an open file

Interface and Implementation of SeqFile

```
public class SeqFile {
  public boolean isOpen() {
    ...
  }

  public boolean isClosed() {
    ...
  }

  public byte[] read() {
    assertIsOpen();
    ...
  }

  public void delete() {
    assertIsClosed();
    ...
  }

  ...
```

Abstract State vs. Implementation State

Abstract State

```
public interface SeqFile {

  public boolean isEmpty();
  public boolean isOpen();
  ...

}
```

Implementation State

```
public class ByteFile
  implements SeqFile {

  protected byte[] data;
  protected int length;

  public boolean isEmpty() {
    return length == 0;
  }

  ...
}
```

Program to an Interface Principle 1 / 2

Program to an interface,

not an implementation

Program to an Interface Principle 2 / 2

- Program to an abstract state model
  - Do not rely on any implementation details
- Do not rely on what is not in the interface
  - Do not expect implementation side-effects
  - Do not rely on specific performance unless guaranteed
- When can you not use the interface?

Class Implementation

- A concrete implementation class
  - has an interface that is a superset of any interfaces it implements
  - is a complete implementation of that class
  - implements an abstract state model (interface) using primitives
- Design by primitives
  - The implementation state is covered by primitive methods
  - All other methods should utilize these primitive methods
- Corollaries
  - Do not change any fields outside the primitive methods
  - Prepares for implementation evolution

Implementation with Design by Primitives

```
public void insert(int i, String c) {
  assertIsValidIndex(i, getNoComponents() + 1);
  assertIsNonNullArgument(c);
  int oldNoComponents = getNoComponents();
  doInsert(i, c);
  assert (oldNoComponents + 1) == getNoComponents() : "pc failed";
}
```

```
protected void doInsert(int index, String component) {
  int newSize = getNoComponents() + 1;
  String[] newComponents = new String[newSize];
  for (int i = 0, j = 0; j < newSize; j++) {
    if (j != index) {
      newComponents[j] = components[i++];
    } else {
      newComponents[j] = component;
    }
  }
  components = newComponents;
}
```

Quiz: Implementing Primitives

- Would you implement doSetComponent as shown?
  If so, under which circumstances? If not, why not?

```
public void setComponent(int i, String c) {
  assertIsValidIndex(i);
  doSetComponent(i, c);
}

protected void doSetComponent(int i, String c) {
  doInsert(i, c);
  doRemove(i + 1);
}
```

Inheritance Interface

- Inheritance Interface
  - The description of an abstract state space underlying the state model
  - The use-client interface specifies the state model including all constraints
  - The inheritance interface specifies the full state space and is unprotected [1]
- Design by primitives
  - The abstract state space should be represented by a set of primitive methods
  - Typically, these primitive methods are hook methods
  - They may have default implementations
- Narrow inheritance interface principle
  - The interface should be minimal to allow fast and simple implementation
  - This may imply inefficient implementations in the abstract superclass

[1] In a larger system design context, you may want to protect superclasses against abuse, but in the one-system context here this is not of concern

AbstractName Inheritance Interface

```
public abstract class AbstractName implements Name {
  ...
  public abstract int getNoComponents();
  protected abstract String doGetComponent(int i);
  protected abstract void doSetComponent(int i, String component);
  protected abstract void doInsert(int index, String component);
  protected abstract void doRemove(int index);
}
```

```
public class StringName
  extends AbstractName {

  protected int noComponents;
  protected String name;

  ...

}
```

```
public class StringArrayName
  extends AbstractName {

  protected int noComponents;
  protected String[] components;

  ...

}
```

The Narrow Inheritance Interface Principle

```
public abstract class AbstractName implements Name {
  ...

  protected void doSetComponent(int i, String c) {
    doInsert(i, c);
    doRemove(i + 1);
  }
  ...
```

```
public class StringArrayName extends AbstractName {

  protected int noComponents;
  protected String[] components;

  protected void doSetComponent(int i, String c) {
    components[i] = c;
  }
  ...
```

Bottom-up or Top-down

- Thinking the class hierarchy bottom-up
  - A subclass calls methods from the superclass
    - These are either helper methods
    - Or the superclass has a different (domain) type
- Thinking the class hierarchy top-down
  - A superclass delegates implementation to subclasses
    - The superclass provides the algorithm while
    - the subclasses provide the implementation of the primitive steps

The Open / Closed Principle

- The Open / Closed Principle
  - “A class should be open for extension, but closed for modification”
  - A moniker for one of the many rather vague “principles” of good design
- A (somewhat) useful interpretation
  - A class should be open for extension (that does not violate its interface)
  - A class should be closed to modification (that would violate the interface)
- To be replaced by two clearer rules
  - Liskov substitutability principle [LW93]
  - The abstract superclass rule [H94]

Quiz: Interface Structure

1. The methods of your class or interface are spread around and you want it to be more readable. How should you order your methods?
    1. By method type (getter, setter, ...)
    2. By method visibility (public, ...)
    3. By method purpose
    4. By client needs
        1. By use-client needs
        2. By inheritance-client needs
    5. Some or all of the above

2. Where to the implementation fields of your class go?
    1. Above the methods (start of class)
    2. Below the methods (end of class)
    3. Close to the methods using them
    4. Does not matter

Class and Interface Evolution

1. Simple class
2. Interface separation
3. Implementation classes
4. Abstract superclass

1. Simple Class

(Class) Interface of Name Class

```
public class Name {

  public String asString() { /* ... */ }
  public String asString(char delimiter) { /* ... */ }
  public String[] asStringArray() { /* ... */ }

  public String getComponent(int i) { /* ... */ }
  public void hasComponent(String c) { /* ... */ }
  public void setComponent(int i, String c) { /* ... */ }
  public Iterator<String> iterator() { /* ... */ }
  public void insert(int i, String c) { /* ... */ }
  public void remove(int i) { /* ... */ }

  ...

}
```

Implementation of a Simple Name Class

```
public class Name {
  protected String[] components;
  protected int length;

  public void remove(int index) {
    assertIsValidIndex(index);
    doRemove(index);
  }

  protected void doRemove(int i) {
    System.arraycopy(components, i+1, components, i, length-i);
    length -= 1;
  }

  protected void assertIsValidIndex(int index) {
    if ((index < 0) && (index >= length)) {
      throw new IndexOutOfBoundsException(“helpful message”);
    }
  }
  ...
```

How to Get to Methods

- Use-client perspective
  - Two perspectives: Feature implementation and tests
  - Meyer recommends taking a shopping bag approach

- Completeness / internal quality
  - Completeness of protocols
  - Experience with similar situations

How to Group Methods

- Group methods by collaboration purpose
  - Also called “roles” objects play, sometimes “traits” or “protocols”
  - A protocol typically adds a more stringent specification

- Smalltalk method categories (old and bad)
  - Group getters and group setters separately

- Various (browsing) views by IDEs
  - Can only be based on language-level properties

2. Interface/Class Separation

3. Implementation Classes

4. Abstract Superclass

General Types of Classes

- Simple class
  - A class that implements it all
- Interface (class)
  - An interface definition for some classes
- Implementation class
  - A class that implements an interface
- Abstract superclass
  - An abstract class intended to be extended
- Default implementation class
  - An implementation class supposed to be the default choice

Special-Purpose Types of Classes

- Tagging interface
  - An interface indicating a hidden functionality
- Mix-in class (trait class)
  - A class providing partial implementation functionality
- Design purpose
  - Follows design pattern, e.g. Adapter, Factory

Review / Summary of Session

- Classes vs. interfaces
  - Modeling vs. implementation
  - Various types of interfaces
  - Abstract state vs. concrete state

- Basic class design rules
  - Structuring a use-client interface
  - Creating an inheritance interface
  - Design by primitives
  - Evolve, don’t predict

Thank you! Questions?

dirk.riehle@fau.de – **https://oss.cs.fau.de**

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

Credits and License

- Original version
  - © 2012-2021 Dirk Riehle, some rights reserved
  - Licensed under Creative Commons Attribution 4.0 International License
- Contributions
  - None yet