CS 4500
Software Development

[Objects, Data Structures, Interfaces]

Ferdinand Vesely

September 24, 2019

Information Hiding

- Principle: separation of design decisions subject to change
- Module has knowledge of a design decision
    - hides from the rest of the system (secret)
- Separation of interface and implementation
- Interface to reveal as little as possible

Information Hiding

- Continuity criterion:
  - module changes
  - changes apply only to its secret elements
  - public ones untouched
  - then: clients of module will not be affected
- Smaller public part – changes more likely to be in secret part

Information Hiding

Technical Requirement

> *It should be impossible to write client modules whose correct functioning depends on secret information.*

- Language support, e.g.:
  - Java/C++: **`private`**/**`public`**/**`protected`**
  - ML/OCaml modules: abstract types
  - Haskell: constructor hiding (somewhat weak)
  - others?

Data Abstraction

```java
public class Point {
  public double x;
  public double y;
}
```

vs.

```java
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Data Abstraction

```java
public class Point {
  public double x;
  public double y;
}
```

- Exposes implementation
- No access policy – coordinates manipulated individually

Data Abstraction

Is

```java
public class Point {
  private double x;
  private double y;

  public void setX(double x) { this.x = x; }
  public void setY(double y) { this.y = y; }

  public double getX() { return this.x; }
  public double getY() { return this.y; }
}
```

Any better?

Data Abstraction

```java
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

- Hides implementation
- Representation: is it polar or Cartesian?
- Access policy: coordinates must be set at once

Information Hiding $\leftrightarrow$ Data Abstraction

- Hiding
  - NOT: variables private, access through getters/setters
  - Abstraction!
- Abstract interfaces – manipulate *essence* of data
- Without knowledge of implementation

Data Structure vs. Object

Object

- hide data behind abstractions
- expose functions operating on data

Data Structure (Records)

- expose data
- no meaningful functions

Data Structures (Records)

```java
public class Square {
  public Point topLeft;
  public double side;
}
public class Circle {
  public Point center;
  public double radius;
}
```

```java
public class Geometry {
  public final double PI = 3.1415926535;

  public double area(Object shape)
      throws NoSuchShapeException {
    if (shape instanceof Square) {
      Square s = (Square) shape;
      return s.side * s.side;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle) shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Data Structures (Records)

Advantages?

- If new operation added to `Geometry`:
  - ⇒ no change to shape classes
  - ⇒ no change to classes dependent on shapes

However:

- If new shape added:
  - ⇒ all functions in `Geometry` need to be changed

Objects

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() { return side * side; }
}

public class Circle implements Shape {
  private Point center;
  private double radius;
  private final double PI = 3.141592653589793;

  public double area() { return PI * radius * radius; }
}
```

Objects

- No “centralized” `Geometry` class necessary
- If new shape added:
  ⇒ no existing function affected
- If new function added:
  ⇒ all Shapes need changing

Data Structures vs. Objects

Takeaway

"Everything is an object" = myth

Sometimes a transparent data structure is appropriate

The Law of Demeter

aka *The Principle of Least Knowledge*

Any method f of class C should only call the methods of

(a) *C itself*
(b) *an object created by f*
(c) *an object passed as an argument to f*
(d) *object held as an instance variable of C*

The Law of Demeter

- In particular: do not invoke methods of objects resulting from invocations in (a)-(d)
- “Talk to friends, not strangers”

Rationale

Avoid “train wrecks”, e.g.:

```
String outputDir =
        ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- Lots of knowledge for one method

Train Wrecks

```
String outputDir =
        ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- `Options` and `ScratchDir` seem to expose their internals
- Are they objects or data structures?
- If latter, why not simply:

```
String outputDir = ctxt.options.scratchDir.absolutePath;
```

- “Beans”

Hiding Structure

- If `Context` is an object, we should be telling it to do something, not querying its internals
- Why do we query the absolute path in the first place?

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

TL;DR: Creating a new scratch file of a given name.

Hiding Structure

- Give the responsibility to `Context`, then ask it to create a file:

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

- `Context` can hide its internals ✔
- The method does not have to navigate through `Options` and `ScratchDir` ✔

Avoid Hybrids

- Half object, half data structure
- Either public variables or setters/getters – exposing internals
- But also significant methods implementing business logic
- Worst of both worlds:
  - hard to add new methods
  - hard to add new data structures

Data Transfer Objects

- Quintessential data structure: no methods, just public vars
- DTOs – usually used for parsing messages from sockets
- Often first in a series of translation stages – raw data to domain objects
- “Bean” variations with accessors/mutators

Data Transfer Objects

```
public class Address {
  public String street;
  public String streetExtra;
  public String city;
  public String state;
  public String zip;
}
```

or

```
public class Packet {
  public short sourcePort;
  public short destinationPort;
  public short length;
  public short checksum;
  public ByteBuffer data;
}
```

Summary

- Objects: expose behavior, hide data
  - Easy: add new classes of objects without changing existing behaviors
  - Hard: add new behaviors to existing objects
- Data structures: expose data, no significant behavior
  - Easy: add new behaviors to existing data structures
  - Hard: add new data structures to existing functions
- Choose the approach that fits the job

Module (Interface) Specifications

Specifying Interfaces – Abstractly

1. Types of Data
2. Operations
3. Axioms
4. Preconditions

Specifying Interfaces Abstractly

Stack

Types

- for any type T, Stack(T)
- Boolean

Specifying Interfaces Abstractly

Stack

Operations

1. $\text{new} : \text{Stack}(T)$
2. $\text{push} : \text{Stack}(T) \times T \rightarrow \text{Stack}(T)$
3. $\text{pop} : \text{Stack}(T) \rightarrow \text{Stack}(T)$
4. $\text{top} : \text{Stack}(T) \rightarrow T$
5. $\text{empty} : \text{Stack}(T) \rightarrow \text{Boolean}$

Specifying Interfaces Abstractly

Stack

Axioms

1. empty(new) = true
2. empty(push(s, x)) = false
3. top(push(s, x)) = x
4. pop(push(s, x)) = s

Specifying Interfaces Abstractly

Stack

Preconditions

1. pop(s) requires empty(s) = false
2. top(s) requires empty(s) = false

Stack Interface

Data Types

1. for any type T, Stack(T)
2. Boolean

Operations

1. new : Stack(T)
2. push : Stack(T) $\times$ $T \rightarrow$ Stack(T)
3. pop : Stack(T) $\rightarrow$ Stack(T)
4. top : Stack(T) $\rightarrow$ T
5. empty : Stack(T) $\rightarrow$ Boolean

Axioms

1. empty(new) = true
2. empty(push(s, x)) = false
3. top(push(s, x)) = x
4. pop(push(s, x)) = s

Preconditions

1. pop(s)
   requires empty(s) = false
2. top(s)
   requires empty(s) = false

Stack Interface Abstractly – Procedural

Data Types

1. Elements: T
2. Boolean – true or false

Operations

1. new : ()
2. push : $T \rightarrow ()$
3. pop : ()
4. top : T
5. empty : Boolean

Pre- and postconditions

1. new
   - PRE: –
   - POST: empty = true
2. push(x)
   - PRE: –
   - POST: empty = false, top = x
3. pop
   - PRE: empty = false
   - POST: –
4. top
   - PRE: empty = false
   - POST: empty = false

Stack Interface – Java Speak

Package: **Stack**

The package Stack provides services of a simple stack. We request that this package be implemented using Java 13 and it should satisfy the following definition...

Data

1. Elements – abstract type referred to as T in this specification
2. Booleans – we use `bool` here

Stack Interface – Java Speak

Operations

```
interface Stack<T> {

  // initialize an empty stack
  // POST: this.empty()
  void Stack<T>();

  // push(x) pushes an element x onto the stack
  // POST: !empty()
  //       top() = x
  void push(T);

...
```

Stack Interface – Java Speak

```
...

  // remove an element from the top of the stack
  // PRE: !empty()
  void pop();

  // return the element on the top of the stack
  // PRE: !empty()
  // POST: !empty()
  T top();

  // check if stack is empty
  bool empty();
}
```