Interacting Objects

Mikael Svahnberg*

2023-08-22

Contents

*Mikael.Svahnberg@bth.se

1 Introduction

- Barnes & Kölling Chapter 3, Object Interaction
- Design Principle: *High Cohesion*
- Design Principle: *Encapsulation*
- Object Oriented Analysis

2 High Cohesion

- Each method should have a single well defined responsibility
- Each object should have a small and well defined area of responsibility
- Different responsibilities → different methods, different objects
- This means we will be creating new objects in our program for different responsibilities
- We may have to write different classes too.
- For Example, `t1:Ticket` represent one trip by one person.
  - More trips → more tickets
  - More persons → more tickets
  - Different parts of the same trip (e.g. train, flight, taxi) → more tickets
  - We may thus also need a `tc1:TicketCollection` to collect and manage all the separate tickets.

3 Encapsulation

- Each object has a *public* interface and a *private* implementation
- Public interface:
  - Methods to access and/or modify the object's state.
  - Constructors to initially set up the object.
- Private implementation:
  - Attributes to represent the object's state
  - "helper" methods used by the public interface method.
  - Associations to other objects that are accessed via *their* public interface.
- Note that in Java, the implementation is mixed with the interface.
- In C++, the interface will be in class declaration, and the implementation can be anywhere else.

```
public class Car {
  static final int FUELCAPACITY;

  private String myPlate;
  private String myColour;
  private int myCurrentFuelLevel;
  private int myCurrentSpeed;
  private int myOdometer;

  public Car() {
    myOdometer = 0;
  }

  public void refuel(int amount) {
    int fuel = myCurrentFuelLevel + amount;
    if (fuel > FUELCAPACITY) {
      myCurrentFuelLevel = FUELCAPACITY;
      overflow(fuel-FUELCAPACITY);
    } else {
      myCurrentFuelLevel = fuel;
    }
  }

  private void overflow(int amount) {
    if (amount > 100) {
      // Explode
    } else if (amount > 50) {
      // Start a fire
    } else if (amount > 20) {
      // create a flammable pool
    } else {
      // create a stain on the car
      myColour += " horribly stained by fuel";
    }
  }
}
```

4 Modularisation

- A well encapsulated object can be seen as a *module*.
- There are also other means for modularisation.
  - `package` in Java
  - `namespace` in C++
  - Structuring your program into directories and sub-directories in the filesystem
  - Separate projects in your Configuration Management tool

- Modularisation helps *development*, but is only loosely connected to the *execution architecture*

5 The right Abstraction Level for a Module

- From Barnes & Kölling, let's build a clock.

```
public class Clock {

  public Clock() {
  }

  public void update() {
  }

  public void display() {
  }

  public static void main(String [] args) {
    Clock theClock = new Clock();
    TimeUnit t = new TimeUnit(3,"tst");

    for(int i = 0; i < 10; i++) {
      theClock.update();
      theClock.display();

      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) { }
    }

  }
}
```

6 Clock Challenges

- Seconds range from 0 to 60, updates every 1000 milliseconds (`Thread.sleep(1000)`), and restarts from 0.
- Minutes range from 0 to 60, updates once the seconds restart, and restarts from 0.
- Hours range from 0 to 24, updates once the minutes restart, and restarts from 0.
- Numbers are drawn.
- We *could* put all of this logic into the `update()` function.
  - We would have a very specialised clock, but no flexibility.

  - One long method with all the logic in one difficult to overview place.

- We *could also* write a separate class for seconds, minutes, and hours
  - The program code would be almost the same.
  - Find a bug in one place, figure out all other places to modify
  - *Be lazy.* Can you generalise the behaviour?

- Let us instead introduce an *abstraction* for one `TimeUnit`

```
public class TimeUnit {
  private int myLimit;
  private int myValue;
  private String myUnit;

  public TimeUnit() { this(60); }

  public TimeUnit(int theLimit) { this(theLimit, ""); }

  public TimeUnit(int theLimit, String theUnit) {
    myLimit = theLimit;
    myUnit = theUnit;
    myValue = 0;
  }

  public boolean update() {
    myValue = ++myValue % myLimit;
    return (0 == myValue);
  }

  public String getDisplayValue() {
    return String.format("%02d", myValue);
/*    if (10 > myvalue) {
      return "0" + myValue;
    } else {
      return "" + myValue;
    }*/
  }
}
```

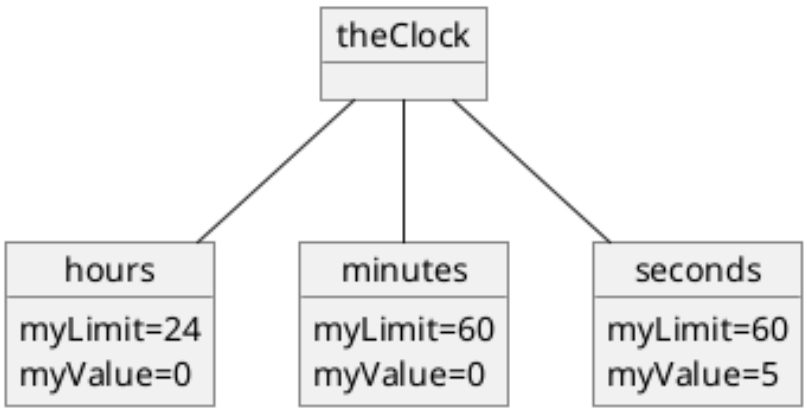
6.1 Updated Clock class

```
public class Clock {
  private TimeUnit hours = new TimeUnit(24, "h");
  private TimeUnit minutes = new TimeUnit(60, "m");
  private TimeUnit seconds = new TimeUnit(60, "s");

  public Clock() {
  }

  public void update() {
```

```
    if (seconds.update()) {
      if (minutes.update()) {
        hours.update();
      }
    }
  }

  public void display() {
    StringBuilder sb = new StringBuilder();
    sb.append(hours.getDisplayValue());
    sb.append(":");
    sb.append(minutes.getDisplayValue());
    sb.append(".");
    sb.append(seconds.getDisplayValue());
    System.out.println(sb.toString());
  }

  public static void main(String [] args) {
    Clock theClock = new Clock();

    for(int i = 0; i < 10; i++) {
      theClock.update();
      theClock.display();

      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) { }
    }

  }
}
```

7 Summary of the Clock

- Four Objects:
  - `theClock:Clock`
  - `hours:TimeUnit`
  - `minutes:TimeUnit`
  - `seconds:TimeUnit`
- Each `TimeUnit` object is separate and knows nothing about the others
- The `Clock` object knows about the three `TimeUnit` objects, but not how they work.
- After the constructor, it does not even know how many hours there are in a day, or minutes in an hour.

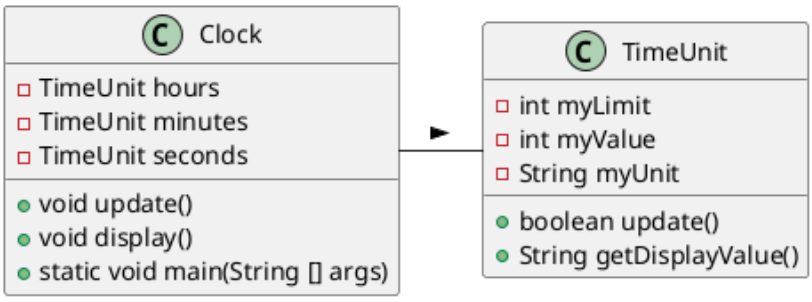


- We can optimise further.
- Why, for example, do we need to have three variable names `hours`, `minutes`, `seconds`?
- Why is the `Clock` responsible for the `Thread.sleep()` and not the smallest `TimeUnit`?
- Why does the `Clock` know which character to preceed each TimeUnit with (i.e., a ':' or a '.')?
- Why is the `main()` function be responsible for calling `theClock.display()`?

Fix these issues to get *higher cohesion* !

- `Clock` should know it has a set of `TimeUnit` objects. When the first rolls over, the next one updates.
- `Clock` should know when to stop updating, but nothing more.
- The rest is known by each separate `TimeUnit` object.

8 Classes and Objects



- The four objects are defined in two classes
- The *Class Diagram* defines the attributes and methods that every object will have

- It does not describe the *value* of any attribute
- Indeed, it does not even describe how many objects of a particular type there will be

9 Logical Operators

```
System.out.println(1==1);
System.out.println(1==1 && 1==2); // AND
System.out.println(1==1 || 1==2); // OR
System.out.println(!(1==1)); // NOT
```

- With `||` java keeps evaluating from left to right until it finds a `true` value
- With `&&` java keeps evaluating from left to right until it finds a `false` value

```
public class LogicTest {
  public static boolean test(int i) {
    System.out.println("" + i + " : " + (0==i%5));
    return 0==i%5;
  }

  public static void main(String[] args) {
    if( test(1) || test(2) || test(3) || test(4) || test(5) || test(6) || test(7)) {
      System.out.println(true);
    }

    if( test(1) && test(2) && test(3) && test(4) && test(5) && test(6) && test(7)) {
      System.out.println(true);
    }
  }
}
```

- this is often useful to try different approaches.
- More common in functional programming.

```
public void update() {
  boolean result = seconds.update() && minutes.update() && hours.update();
}
```

10 Some more on Java Strings

- A String is an *immutable object* in Java.
- We can glue together (concatenate) strings: `"First" + "Second"`
- The result, however, is a *new* string `"FirstSecond"`

```
String f = "First";
String s = "Second";
f = f+s; // Create a new string "FirstSecond" and store a reference to it in f. Garbage co

// Be wary of the == operator:
String ss = "Second";
String fs = "FirstSecond";

System.out.println(s==ss);
System.out.println(f==fs);

// Instead, use equals():
System.out.println(f.equals(fs));

// Many concatenations drive the garbage collector into overtime.
// Better to use a StringBuilder:
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
  sb.append(f);
}

System.out.println(sb.toString());
```

11 Type Conversion

```
int x = 42 + 12; // Two integers, the result is as expected.

//  int y = 42 + 12.2; // an int and a double will not work since 12.2 will need to be "do
double y = 12.2 + 42;  // the int can be "upgraded" to a double, so this is ok.

// String s = 12; // String has no such constructor ~public String(int)~
String s = "Hello" + 12; // Now we solicit the help of the compiler to upgrade the int to

// This is a common pattern in Java:
String output = "" + myInteger;

// SUM: Leftmost value determines the desired type. Everything after has to be convertible
```

12 The `this` pointer

- All methods on an object have an implicit parameter `this` which points to the object.
- We do not need to write this in the method signature, Java knows to add it.
- Not always necessary to use if the method or attribute name is unique anyway.

- Can be used to disambiguate, e.g. if a parameter has the same name as an attribute.
- Must be used when "passing on" to another constructor.

```
public class Car {
  private String licensePlate;

  public Car() {
    this("Default");
  }

  public Car(String licensePlate) {
    this.licensePlate = licensePlate;
  }

  public void setPlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }
}
```

13 Becoming Friends with a Debugger

- When you really need understand what is going on in a program
- The debugger is a very important tool, sorely under-utilised by young developers.
- Often built-in to your Development Environment
  - Or else, there is `jdb` on the command line.
- Each debugger differ in exactly how to use them
- Basic Operation
  - Set a *Breakpoint*
  - *Inspect* a class, method, or attribute
  - *Watch* an attribute
  - *Step* through the code and *Step into* method calls

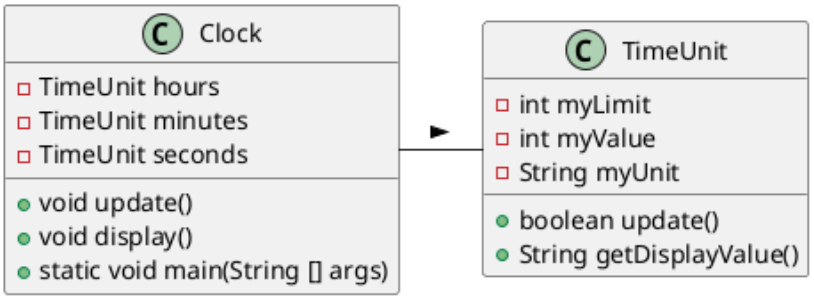
```
$ jdb Clock
Initializing jdb ...
> stop at Clock.update()
Deferring breakpoint Clock.update().
It will be set after the class is loaded.
> run
run Clock
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
```

```
VM Started: Set deferred breakpoint Clock.update()

Breakpoint hit: "thread=main", Clock.update(), line=10 bci=0
10        if (seconds.update()) {

main[1] watch TimeUnit.myValue
Set watch modification of TimeUnit.myValue
main[1] cont
>
Field (TimeUnit.myValue) is 0, will be 1: "thread=main", TimeUnit.update(), line=17 bci=9
17          myValue = ++myValue % myLimit;

main[1] step
>
Field (TimeUnit.myValue) is 1, will be 1: "thread=main", TimeUnit.update(), line=17 bci=17
17          myValue = ++myValue % myLimit;

main[1] step
>
Step completed: "thread=main", TimeUnit.update(), line=18 bci=20
18          return (0 == myValue);

main[1] step
>
Step completed: "thread=main", Clock.update(), line=10 bci=7
10        if (seconds.update()) {

main[1] step
>
Step completed: "thread=main", Clock.update(), line=15 bci=28
15      }

main[1] step
>
Step completed: "thread=main", Clock.main(), line=32 bci=20
32          theClock.display();

main[1] clear Clock.update()
Removed: breakpoint Clock.update()
main[1] cont
> 00:00.01

Field (TimeUnit.myValue) is 1, will be 2: "thread=main", TimeUnit.update(), line=17 bci=9
17          myValue = ++myValue % myLimit;

main[1] print myValue
 myValue = 1
main[1] list
13          myValue = 0;
14        }
```

```
15
16          public boolean update() {
17 =>         myValue = ++myValue % myLimit;
18            return (0 == myValue);
19          }
20
21          public String getDisplayValue() {
22            return String.format("%02d", myValue);
main[1]
```

14 Again in C++

14.1 Headers




```
#ifndef CLOCK_HH
#define CLOCK_HH
#include "timeunit.hh"

class Clock {
public:
  Clock(void);
  void update(void);
  void display(void);
private:
  TimeUnit* hours;    // The objects will be created in the constructor.
  TimeUnit* minutes;
  TimeUnit* seconds;
};
#endif

#ifndef TIMEUNIT_HH
#define TIMEUNIT_HH
#include <string>

class TimeUnit {
public:
  TimeUnit() : TimeUnit(60) {};                          // It is allowed but mostly inadvi
  TimeUnit(int theLimit) : TimeUnit(theLimit, "") {};    // to inline code in the class dec
  TimeUnit(int theLimit, const char* theUnit);
```

```
  bool update(void);
  std::string getDisplayValue(void);
private:
  int myLimit;
  int myValue;
  const char* myUnit; // Old-school form for a string.
};

#endif
```

14.2 Implementation

```
#include <iostream>
#include <string>
#include <chrono>
#include <thread>
using namespace std;


#include "clock.hh"
#include "timeunit.hh"

Clock::Clock(void) {
  hours = new TimeUnit(24, "h");    // If we remove the Clock object, these will be
  minutes = new TimeUnit(60, "m");  // floating around as allocated but unaddressable memo
  seconds = new TimeUnit(60, "s");  // In C++ we need a ~destructor() for our class. Will
}

void Clock::update(void) {
  if (seconds->update()) {   // Pointers, so arrow notation.
    if (minutes->update()) {
      hours->update();
    }
  }
}

void Clock::display(void) {
  cout << hours->getDisplayValue() << ":"  // To be fully analogous with the Java solution
    << minutes->getDisplayValue() << "."   // We should use std::string.append() here inst
    << seconds->getDisplayValue() << endl; // of jumping straight to printouts.
}

int main(void) {
  Clock* theClock = new Clock();

  for (int i = 0; i < 10; i++) {
    theClock->update();
    theClock->display();

    this_thread::sleep_for(chrono::milliseconds(1000));
```

```
  }
}

#include <string>
using namespace std;

#include "timeunit.hh"

TimeUnit::TimeUnit(int theLimit, const char* theUnit) : myLimit(theLimit),myUnit(theUnit)
  this->myValue = 0;  // Note the use of C++ *this pointer.
}

bool TimeUnit::update(void) {
  myValue = ++myValue % myLimit;  // Using the *this pointer is not necessary if the attri
  return (0 == myValue);          // names are unambiguous.
}

string TimeUnit::getDisplayValue(void) { // There is the c function sprintf() which we cou
  if (myValue<10) {                      // we did for java, but it is more of a bother in
    return string("0")+std::to_string(myValue);
  } else {
    return std::to_string(myValue);
  }
}
```

14.3 Type Conversion

```
#include <iostream>
#include <string>

int main(void) {
  int x = 42 + 42;
  double y = 42 + 12.2;  // Ok in C++ since int can be upgraded.
  double yy = 12.2 + 42; // Also ok.

  //std::string s = 12; // Will not work: std::string has no constructor for this.
  std::string s = std::to_string(42);
  std::string ss = std::to_string(12.2); // Same method can do other conversions as well.
  //std::string sss = "Hello"+12; // Will not fail but also not work.
  std::string sss = "hello" + std::to_string(42); // But + is available to concatenate str
                                                  // and things that can be upgraded to a
                                                  // with the help of a constructor in std

  int z = std::stoi("12"); // see also stof(), stod(), ...
  int zz = std::stoi("99.2 bottles of beer on the wall 12");
  std::cout << zz << std::endl;
}
```

14.4 Logical Operators

```
#include <iostream>
using namespace std;

bool test(int i) {
  cout << i << flush;
  return (0==i%5);
}

int main(void) {
  cout << (1==1)  // evaluates to 1 (anything other than zero is true);
       << ! (1==1) // NOT evaluates to 0
       << true    // evaluates to 1
       << false   // evaluates t0 0
       << (1==1 && 1==2) // AND
       << (1==1 || 1==2) // OR
       << endl;

  // Keep evaluating until a true value is found.
  cout << (test(1) || test(2) || test(3)|| test(4)|| test(5)|| test(6)|| test(7)) << endl;

  // Keep evaluating until a false value is found.
  cout << (!test(1) && !test(2) && !test(3)&& !test(4)&& !test(5)&& !test(6)&& !test(7)) <

}
```

15 Summary

- See the World as Objects: *Object Oriented Analysis*
- Finding the right abstraction level – What do objects have in common?
- Initial analysis may use one abstraction level, implementation may use another.
- *Object Oriented Analysis and Design* try to minimise this gap.
- Applications are built as a set of *collaborating objects*
- Design Principle: *High Cohesion*
  - Each object/method has a single responsibility
  - Each object has full control over this responsibility
- Design Principle: *Encapsulation*
  - Only expose what is necessary to others; the public interface.

16 Next Lecture: Designing Applications

- Barnes & Kölling Chapter 15, Designing Applications
- Object Oriented Analysis and Design
- UML (Unified Modelling Language) / RUP
- Discovering Classes
- Designing Interfaces