Objects and Data Structures

Data Abstractions

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

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

- The first one reveals its implementation details.
  - Having getters and setters would not change anything.
- The second one hides its implementation details.
  - It's not possible to know how the data is stored.
  - Exposes abstract interfaces that allow its users to manipulate the *essence* of the data, without having to know its implementation.

Copyright © 2009 Esko Luontola

Clean Code ch6 p94

Data Abstractions

```
public interface Vehicle {
    double getFuelTankCapacityInLiters();
    double getLitersOfGasoline();
}

public interface Vehicle {
    double getPercentFuelRemaining();
}
```

- "We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. **Serious thought** needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters."

Copyright © 2009 Esko Luontola

Clean Code ch6 p95

Data/Object Anti-Symmetry

- **Objects** *hide their data* behind abstractions and expose functions that operate on that data.
- **Data structures** *expose their data* and have no meaningful functions.
- Objects and data structures are virtual opposites.

Copyright © 2009 Esko Luontola

Clean Code ch6 p95

```
public class Square {
    public Point topLeft;
    public double side;
}
```

```
public class Circle {
    public Point center;
    public double radius;
}
```

```
public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}
```

```
public class Geometry {
    public static final double PI = 3.141592653589793;

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

Copyright © 2009 Esko Luontola

Clean Code ch6 p95

```java
public class Rectangle implements Shape {
    public Point topLeft;
    public double height;
    public double width;

    public double area() {
        return height * width;
    }
}
```

```java
public interface Shape {
    double area();
}
```

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

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

```java
public class Circle implements Shape {
    public static final double PI = 3.141592653589793;
    public Point center;
    public double radius;

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

Copyright © 2009 Esko Luontola

Clean Code ch6 p96

Data/Object Anti-Symmetry

- **Procedural code** (code using data structures) makes it *easy to add new functions* without changing the existing data structures.
- **OO code** makes it *easy to add new classes* without changing existing functions.

- **Procedural code** makes it *hard to add new data structures* because all the functions must change.
- **OO code** makes it *hard to add new functions* because all the classes must change.

- Know when to use which approach.

Copyright © 2009 Esko Luontola

Clean Code ch6 p97

The Law of Demeter

- A module should not know the innards of the *objects* it manipulates.
- Definition:
  - a method *f* of class *C* should only call the methods of
    - *C*
    - an object created by *f*
    - an object passed as an argument to *f*
    - an object held in an instance variable of *C*

Copyright © 2009 Esko Luontola

Clean Code ch6 p97

The Law of Demeter

- Does this "train wreck" violate it?

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

- If *ctx*, *Options* and *ScractchDir* are data structures, then the Law of Demeter does not apply.

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

- How is *outputDir* used?

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

- It can be made an object!
  - OutputStream out = ctx.createScratchFileStream(classFileName);

Copyright © 2009 Esko Luontola

Clean Code ch6 p98

Error Handling

Error Handling

- "Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of the scattered error handling. Error handling is important, *but if it obscures logic, it's wrong.*"
- Use exceptions rather than return codes.
- Write your `try-catch-finally` statement first.
  - Exceptions *define a scope* within your program. When writing code that could throw an exception, starting with the `try-catch` block helps you to define what the user of that code should expect, no matter what goes wrong in the `try` block.

Copyright © 2009 Esko Luontola

Clean Code ch6 p103

Exceptions

- Prefer unchecked exceptions.
  - If the catcher is many levels higher then where the exception is thrown, with checked exceptions that exception leaks (because of the `throws` clause) to all levels in between.
- Know the difference between a contingency and a fault. [1]
  - **Contingency**: An expected condition demanding an alternative response from a method that can be expressed in terms of the method's intended purpose. The caller of the method expects these kinds of conditions and has a strategy for coping with them.
  - **Fault**: An unplanned condition that prevents a method from achieving its intended purpose that cannot be described without reference to the method's internal implementation.

Clean Code ch6 p106

[1] http://www.oracle.com/technology/pub/articles/dev2arch/2006/11/effective-exceptions.html

Copyright © 2009 Esko Luontola

Exceptions

- Provide context with exceptions.
  - For example include the reason and method parameters in the exception message thrown by the method. Enough information for logging the error in the `catch`.
- Define exception classes in terms of the caller's needs.
  - When we define exceptions in an application, our most important concern should be *how they are caught.*
  - When using a third-party API, wrap it in an adapter that behaves and throws exceptions in a way that suits your code.

Copyright © 2009 Esko Luontola

Clean Code ch6 p107

```
AcmePort port = new AcmePort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} finally {
   ...
}
```

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

```
public class LocalPort {
    private AcmePort innerPort;

    public LocalPort(int portNumber) {
        innerPort = new AcmePort(portNumber);
    }

    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
    ...
}
```

Clean Code ch6 p108

Copyright © 2009 Esko Luontola

Define the Normal Flow

- Use the Special Case pattern instead of exceptions.

```
try {
    MealExpenses expenses = expenseReportDao.getMeals(employee.getId());
    total += expenses.getTotal();
} catch (MealExpensesNotFound e) {
    total += getMealPerDiem();
}
```

```
MealExpenses expenses = expenseReportDao.getMeals(employee.getId());
total += expenses.getTotal();

public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
        // return the per diem default
    }
}
```

Clean Code ch6 p109
http://martinfowler.com/eaaCatalog/specialCase.html

Copyright © 2009 Esko Luontola

Null

- When these rules are followed throughout the system, barely any null checks are needed:
- **Don't return null.**
  - Then the caller does not need to do null checks.
  - Instead use the Special Case/Null Object pattern.

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for (Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

```
List<Employee> employees = getEmployees();
for (Employee e : employees) {
    totalPay += e.getPay();
}
```

- **Don't pass null.**
  - Then the method does not need to do null checks.
    - The compiler will anyways generate null checks. If they are not enough to fail early (and the code is part of public API), add them yourself.

Copyright © 2009 Esko Luontola

Clean Code ch6 p110

Boundaries

Using Third-Party Code

- Providers of third-party APIs want a wide audience, so they make their APIs as generic as possible.
- Users of the APIs want an interface that is focused on their particular needs.
- Solution: Hide library code as an implementation detail behind abstractions.
  - Prevents the users of the API from calling unwanted third-party methods.
  - Allows the API to evolve with very little impact on the rest of the application.
  - Makes testing easier (using Test Doubles).
- Don't pass Maps (etc.) in public APIs. Keep them inside a class or a close family of classes.

Copyright © 2009 Esko Luontola

Clean Code ch6 p114

```
java.util.Map
clear()
containsKey(Object key)
containsValue(Object value)
entrySet()
equals(Object o)
get(Object key)
hashCode()
isEmpty()
keySet()
put(Object key, Object value)
putAll(Map t)
remove(Object key)
size()
values()
```

```
Map sensors = new HashMap();
...
Sensor s = (Sensor) sensors.get(sensorId);

Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId);
```

- Protect users from API changes.
- Restrict which operations are allowed.

```
public class Sensors {
    private Map sensors = new HashMap();

    public Sensor getById(String id) {
        return (Sensor) sensors.get(id);
    }
    ...
}
```

Copyright © 2009 Esko Luontola

Clean Code ch6 p114

Learning Tests

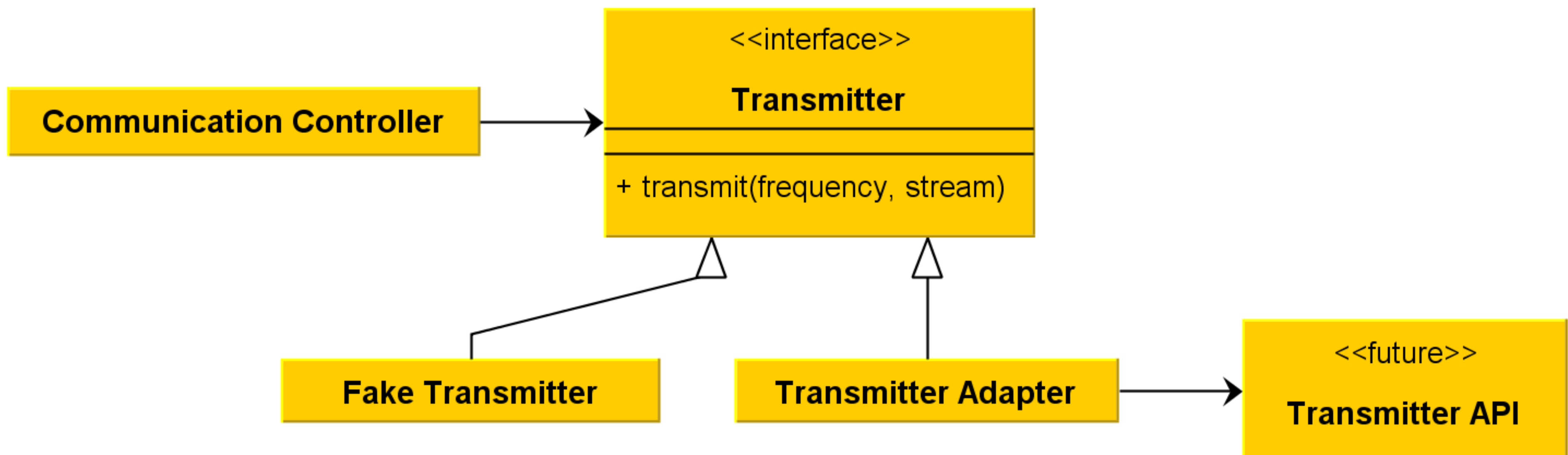
- It's not our responsibility to test third-party libraries, but it's advantageous to us to write tests for them.
- Learning and integrating third-party code is hard. But instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code.
- Write some tests that call the third-party API. Try how it works. Isolate what you learn as test cases.
- Additional benefits:
  - When the third-party code is changed, our learning tests can detect it. Migration to new library versions will be easier.

Copyright © 2009 Esko Luontola

Clean Code ch6 p116

Using Code That Does Not Yet Exist

- Even if some external system does not yet exist and we don't know how it's API will be like, we can still make our own boundary and create an API that suits our needs.
- Use a Test Double during development. When the external system becomes ready, write an adapter for it.



Copyright © 2009 Esko Luontola

Clean Code ch6 p118

Course Material

- *Clean Code*, chapter 6: Objects and Data Structures
- *Clean Code*, chapter 7: Error Handling
- *Clean Code*, chapter 8: Boundaries
- http://www.oracle.com/technology/pub/articles/dev2arch/2006/11/effective-exceptions.html

Copyright © 2009 Esko Luontola