… Programming as a profession

Clean code

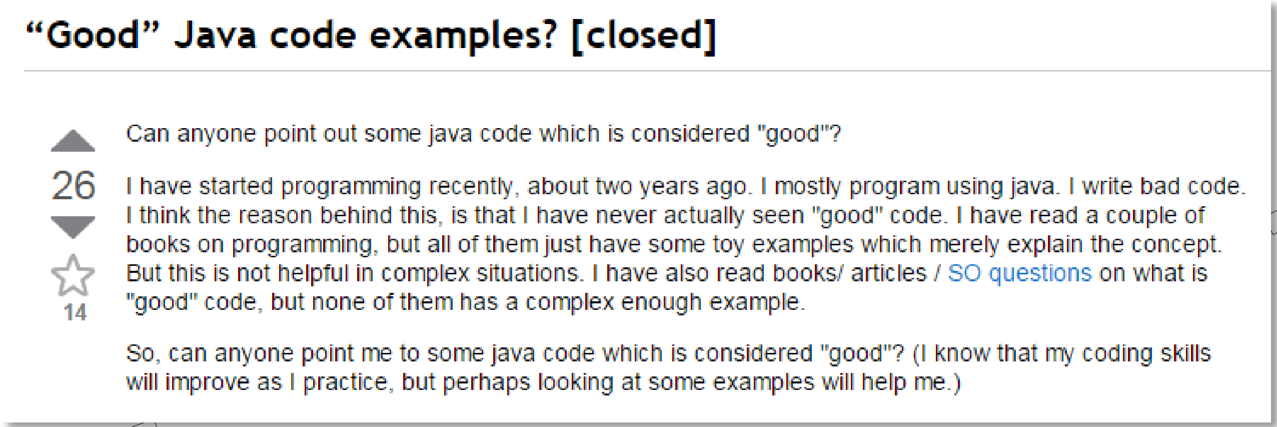
Learning „writing” (in natural language)

- First, we learn to **read** texts
  - Short **reference** texts
  - **Good** literature in original language
    - Moby-Dick, Robinson Crusoe, Pride and Prejudice, War and Peace, The War of the Worlds, The Lord of the Rings
    - Articles (scientific, journals)
- Next, we learn to write texts yourself.
  - Short template texts (letters, resumes, reports, applications)
  - …

Learning "writing" called programming

- First, we learn to **write** (sic!)
- Has anyone heard about a course (or a book): „Reading software code in Java”?
- Yet, to teach through „reading” good (and bad) examples of code are needed
  - How does a good and a bad code looks?

Examples of a good code, where to look?

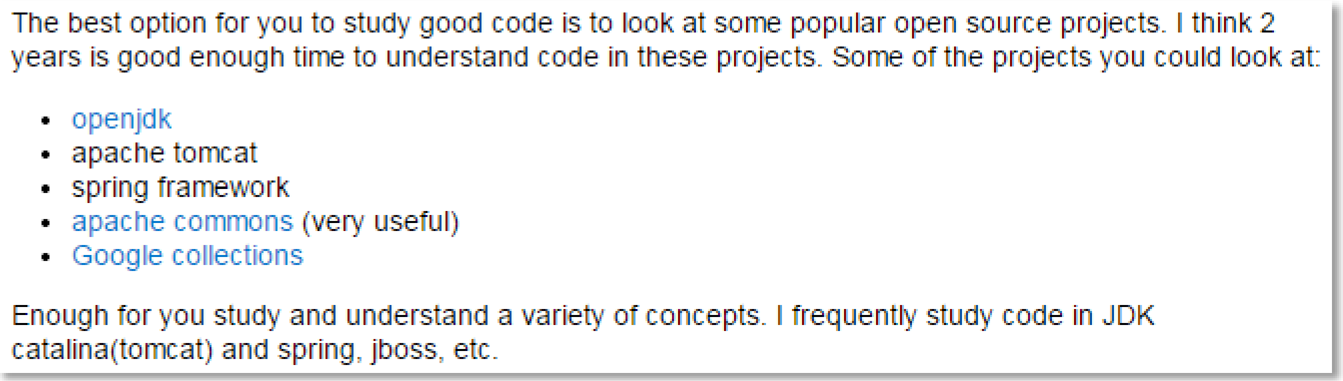


“Good” Java code examples? [closed]

Can anyone point out some java code which is considered "good"?

I have started programming recently, about two years ago. I mostly program using java. I write bad code. I think the reason behind this, is that I have never actually seen "good" code. I have read a couple of books on programming, but all of them just have some toy examples which merely explain the concept. But this is not helpful in complex situations. I have also read books/ articles / SO questions on what is "good" code, but none of them has a complex enough example.

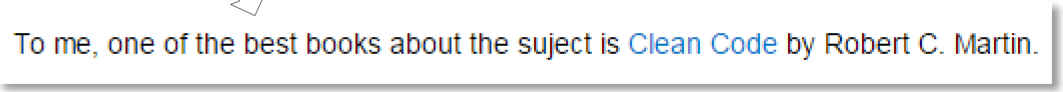
So, can anyone point me to some java code which is considered "good"? (I know that my coding skills will improve as I practice, but perhaps looking at some examples will help me.)



The best option for you to study good code is to look at some popular open source projects. I think 2 years is good enough time to understand code in these projects. Some of the projects you could look at:

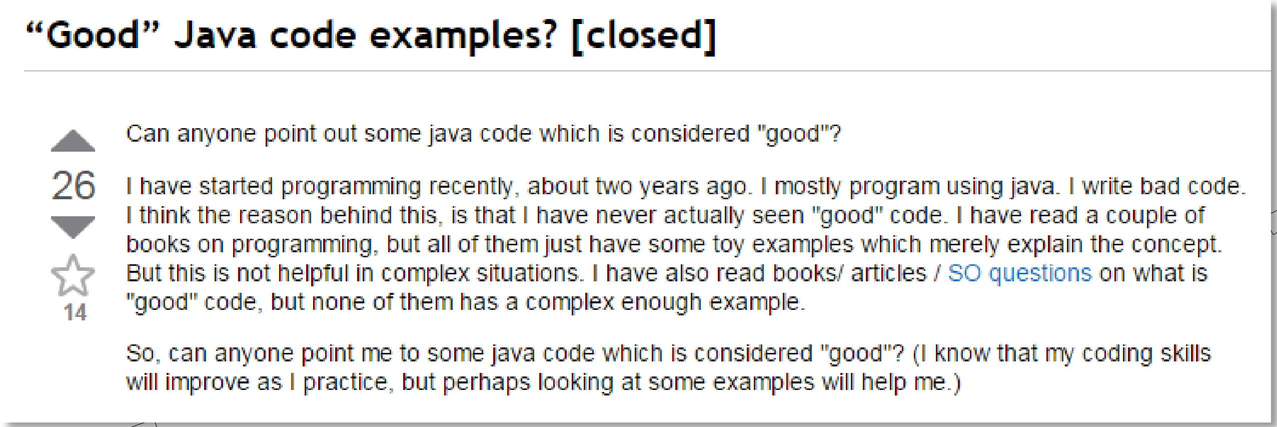
- openjdk
- apache tomcat
- spring framework
- apache commons (very useful)
- Google collections

Enough for you study and understand a variety of concepts. I frequently study code in JDK catalina(tomcat) and spring, jboss, etc.



To me, one of the best books about the suject is Clean Code by Robert C. Martin.

Examples of a good code, where to look?

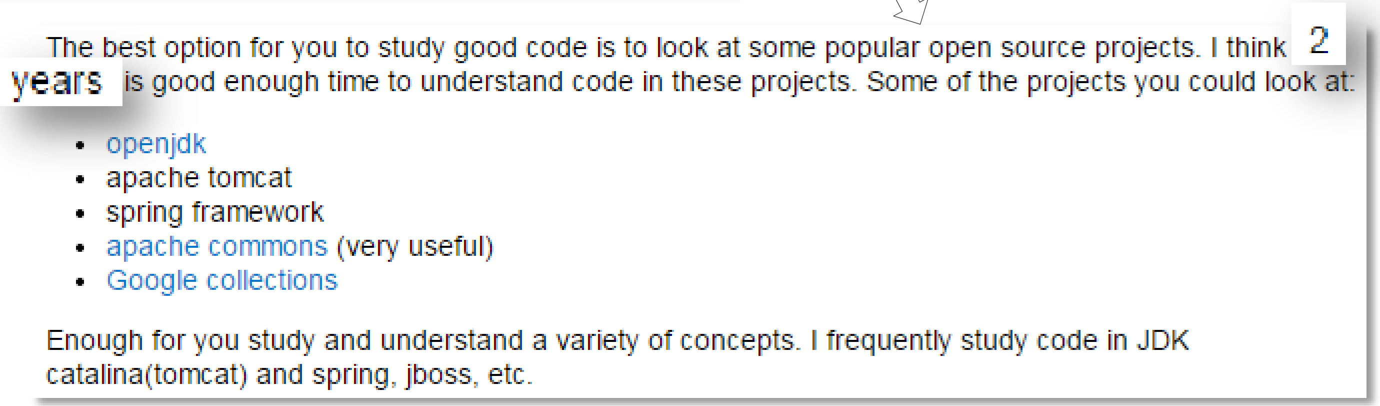


“Good” Java code examples? [closed]

Can anyone point out some java code which is considered "good"?

I have started programming recently, about two years ago. I mostly program using java. I write bad code. I think the reason behind this, is that I have never actually seen "good" code. I have read a couple of books on programming, but all of them just have some toy examples which merely explain the concept. But this is not helpful in complex situations. I have also read books/ articles / SO questions on what is "good" code, but none of them has a complex enough example.

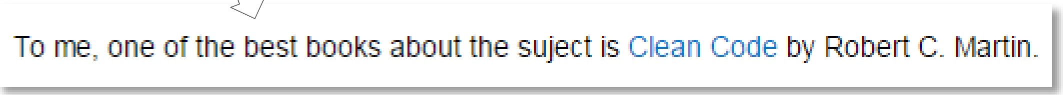
So, can anyone point me to some java code which is considered "good"? (I know that my coding skills will improve as I practice, but perhaps looking at some examples will help me.)



The best option for you to study good code is to look at some popular open source projects. I think 2 years is good enough time to understand code in these projects. Some of the projects you could look at:

- openjdk
- apache tomcat
- spring framework
- apache commons (very useful)
- Google collections

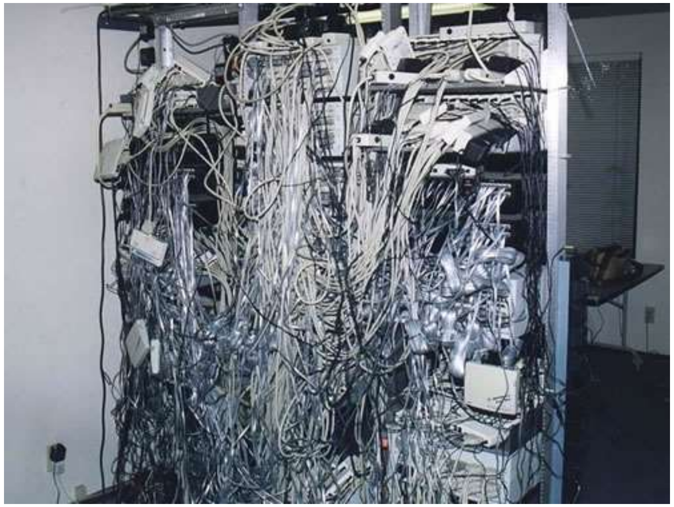
Enough for you study and understand a variety of concepts. I frequently study code in JDK catalina(tomcat) and spring, jboss, etc.

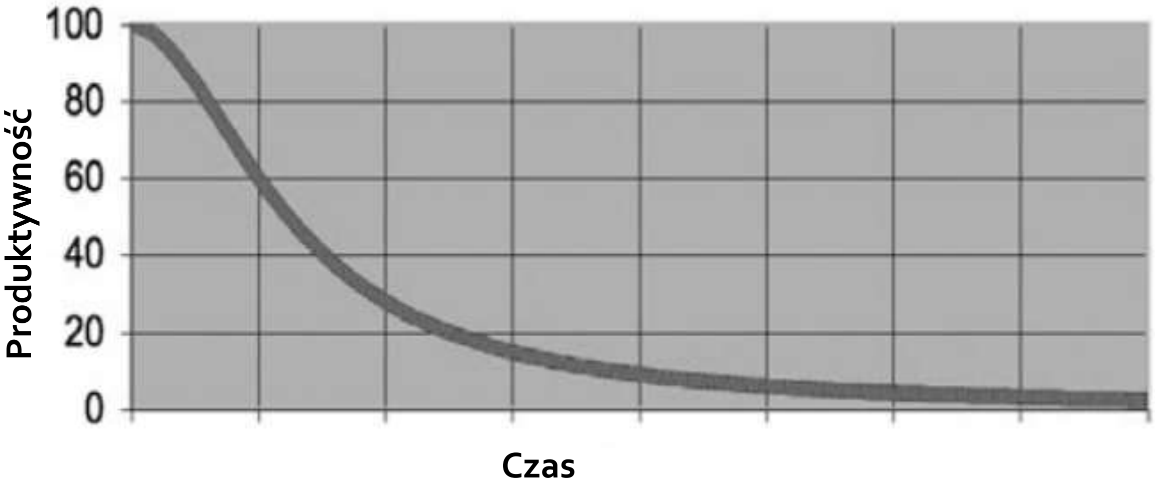


To me, one of the best books about the suject is Clean Code by Robert C. Martin.

Technical debt

- Technical debt is an expression for the increasing level of difficulty a developer will have to introduce a desired change in the codebase. If the debt grows, increase:
  - values of estimates, unpredictability, number of side-effects, levels of developer fear





Technical debt in practice

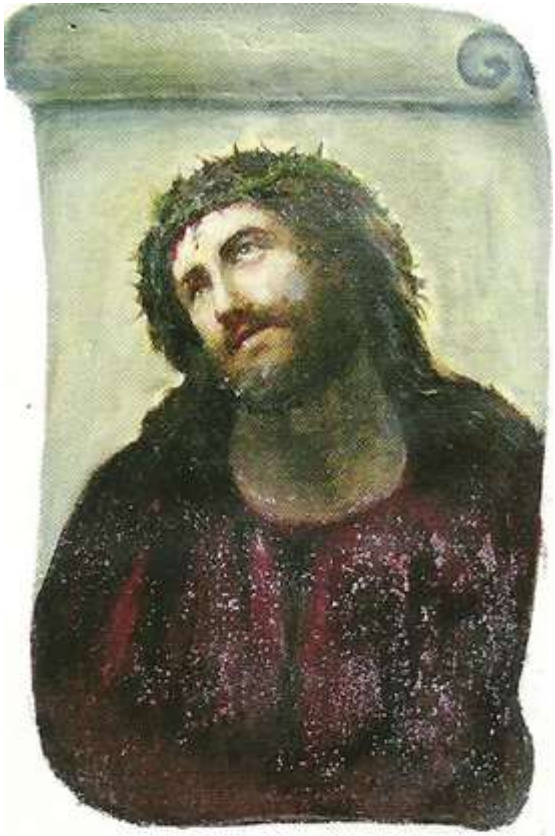
- Code quality
  - Complexity
  - Duplications
  - Violation of Principles
  - Test coverage
  - Documentation

When we incur a technical debt?

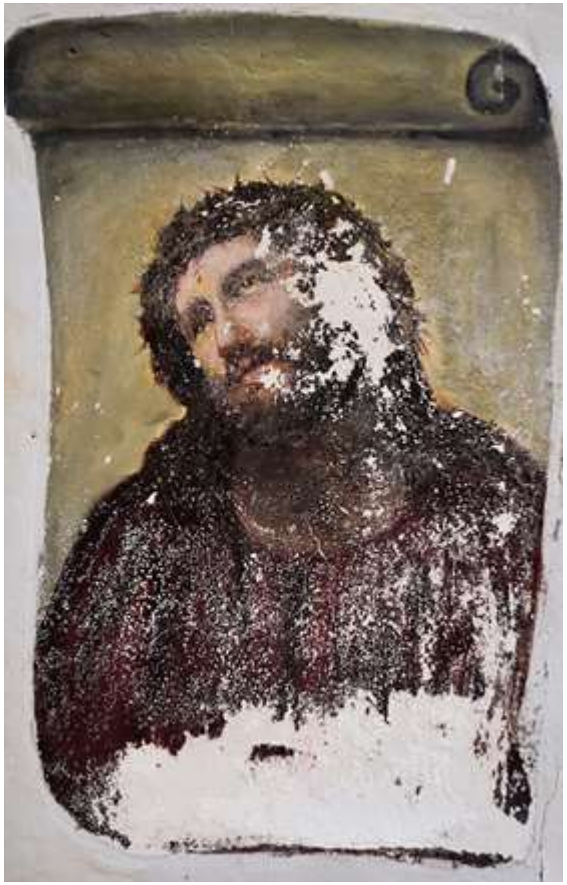
- When we lower code quality! Reasons:
  - Lack of knowledge
  - Lack of professionalism
  - Chasing deadlines
  - Entrophy

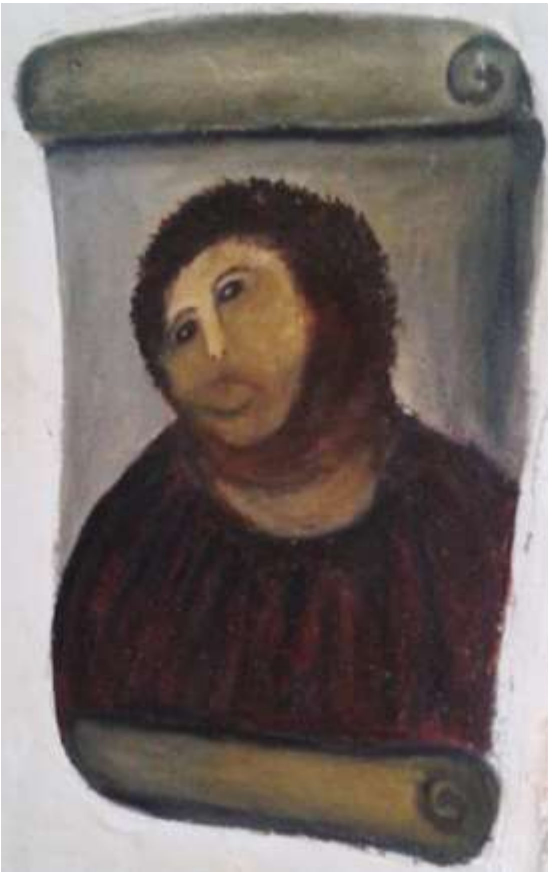
Preception of quality

- We can recognize quality



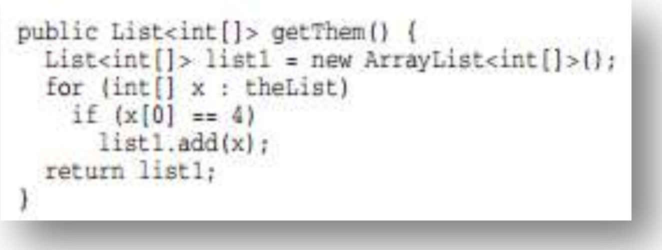
Ecce Homo de Elías García Martínez.





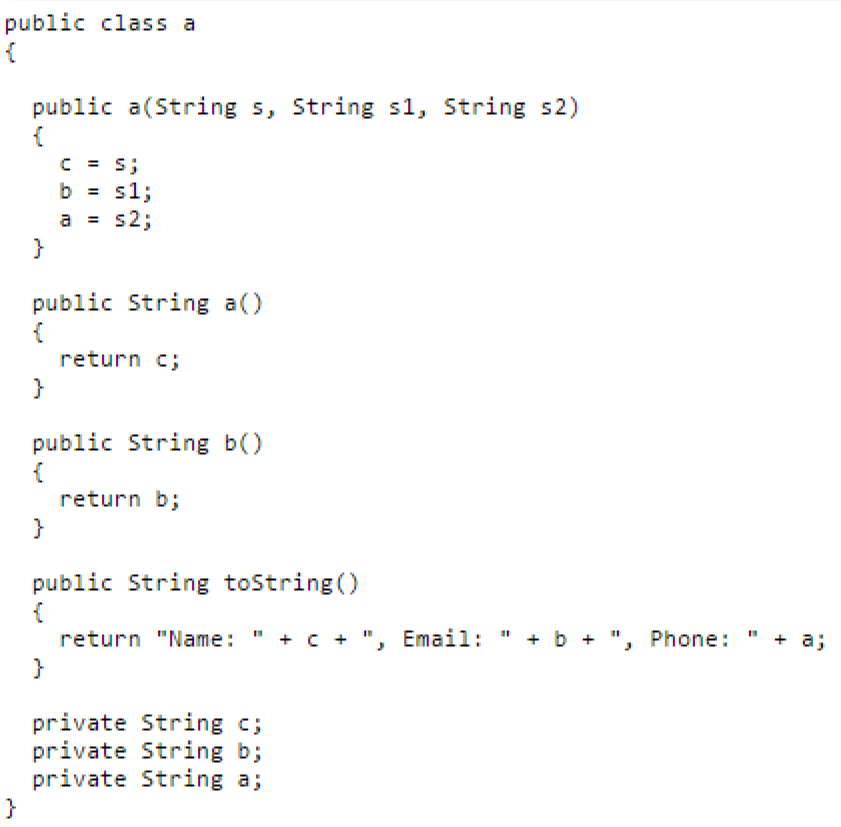
Measures of code quality

- Readibility



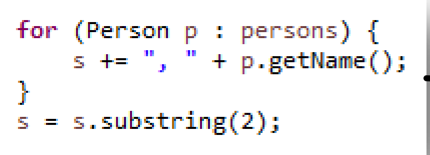
```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

- Manageability (maintainability)

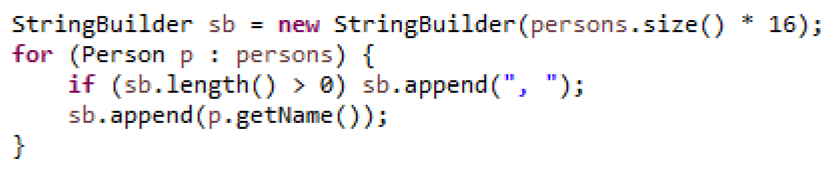


```
public class a
{

  public a(String s, String s1, String s2)
  {
    c = s;
    b = s1;
    a = s2;
  }

  public String a()
  {
    return c;
  }

  public String b()
  {
    return b;
  }

  public String toString()
  {
    return "Name: " + c + ", Email: " + b + ", Phone: " + a;
  }

  private String c;
  private String b;
  private String a;
}
```

- Efficiency



```
for (Person p : persons) {
    s += ", " + p.getName();
}
s = s.substring(2);
```



```
StringBuilder sb = new StringBuilder(persons.size() * 16);
for (Person p : persons) {
    if (sb.length() > 0) sb.append(", ");
    sb.append(p.getName());
}
```

Characteristics of a clean code?

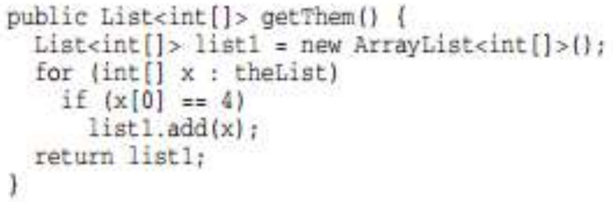
1. Elegant and efficient
2. It reads like a good prose (!)
3. Easy to improve by others
4. It pays attention to details
5. It does not contain repetitions
6. It is consistent, simple, captivating

Clean code- tips

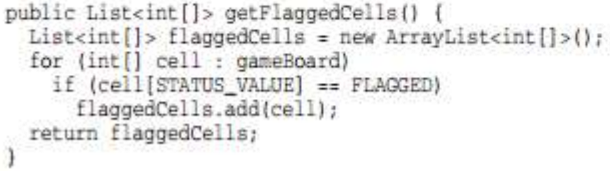
1. Naming
2. Functions design
3. Comments
4. Classes and Objects
   - SOLID principles

Naming

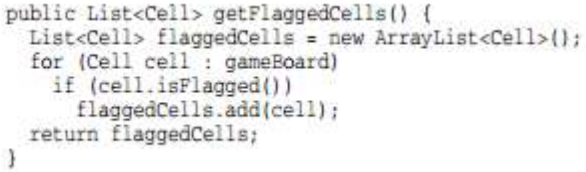
- Names clearify intentions



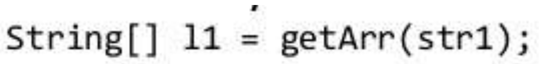
```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```



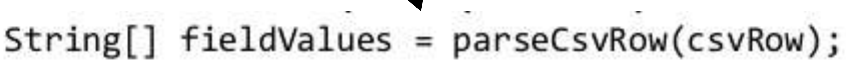
```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```



```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```



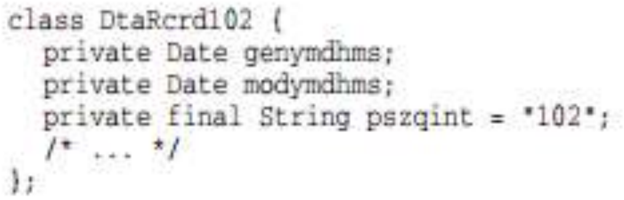
```
String[] l1 = getArr(str1);
```



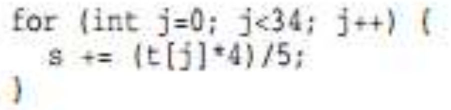
```
String[] fieldValues = parseCsvRow(csvRow);
```

Naming

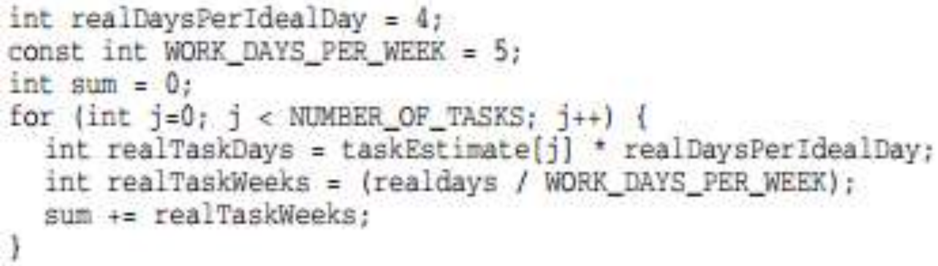
- Names are unambiguous and meaningful



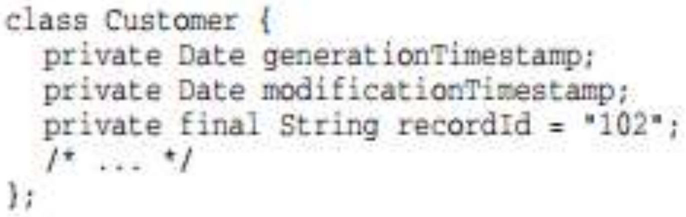
```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```



```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```



```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

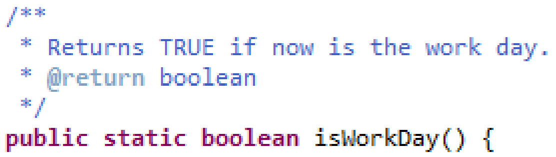


```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private final String recordId = "102";
  /* ... */
};
```

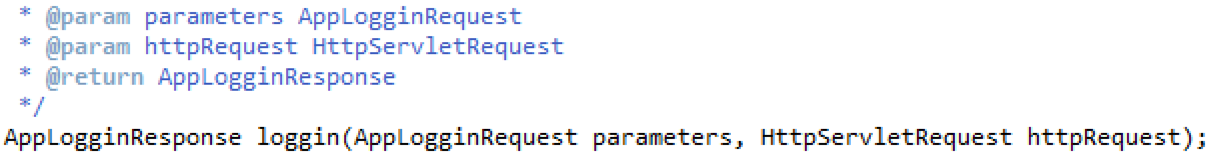
Comments

- We insert comments because:
  - We want to describe the mess in code
  - We want to clarify intentions
  - We want to clarify or emphasize something
  - After all, you need to comment

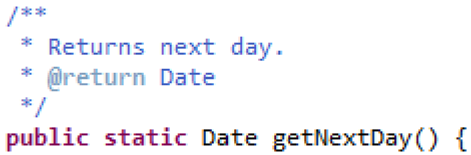
Comments – The good, the bad and the ugly



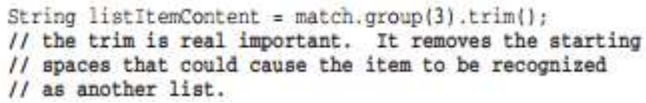
```
/**
 * Returns TRUE if now is the work day.
 * @return boolean
 */
public static boolean isWorkDay() {
```



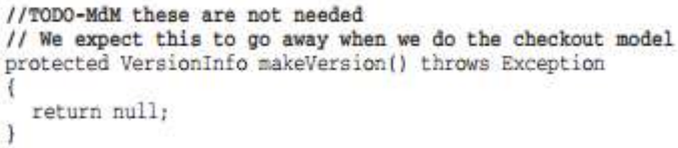
```
 * @param parameters AppLogginRequest
 * @param httpRequest HttpServletRequest
 * @return AppLogginResponse
 */
AppLogginResponse loggin(AppLogginRequest parameters, HttpServletRequest httpRequest);
```



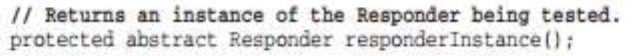
```
/**
 * Returns next day.
 * @return Date
 */
public static Date getNextDay() {
```



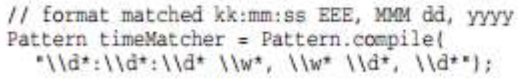
```
String listItemContent = match.group(3).trim();
// the trim is real important.  It removes the starting
// spaces that could cause the item to be recognized
// as another list.
```



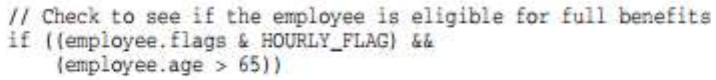
```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
  return null;
}
```



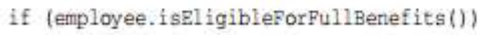
```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```



```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```



```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
   (employee.age > 65))
```



```
if (employee.isEligibleForFullBenefits())
```

Don’t Repeat Yourself!

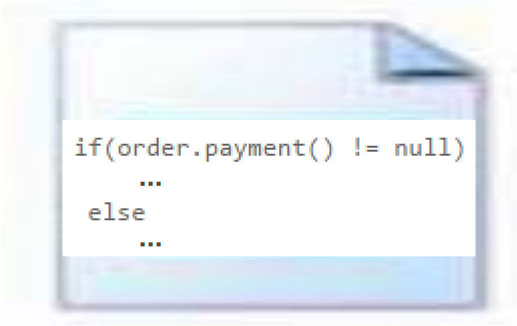
- DRY rule
  - Reuse code instead of duplicate
  - Choose a suitable location for the code
  - "Single source of truth" / "Every piece of knowledge must have a single, unambiguous and authoritative representation within a system" - a single change, a single test
  - Use the correct naming rules

DRY violation - examples

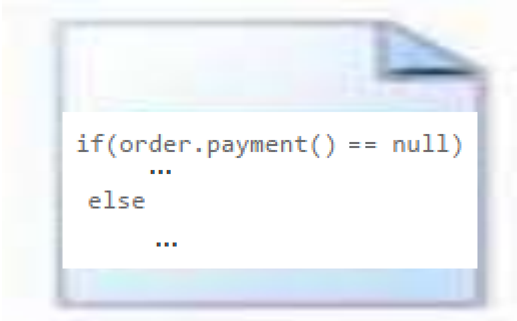
```
class Order {
   Date payment = null;
   Date payment() { return this.payment; }
}
```

→

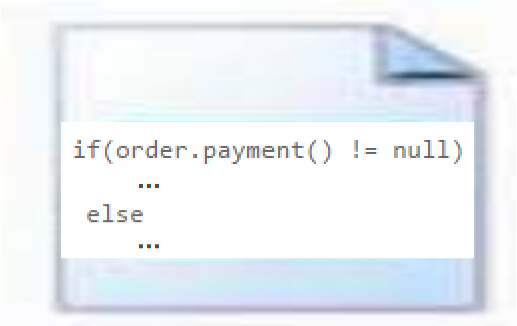
```
class Order {
   Date payment = null;
   Date payment() { return this.payment; }
   boolean isPaid { return order.payment() != null; }
}
```



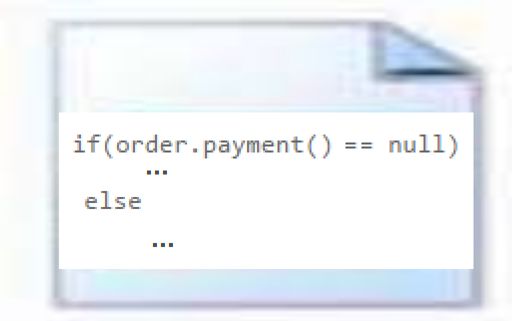
```
if(order.payment() != null)
    …
 else
    …
```



```
if(order.payment() == null)
    …
 else
     …
```

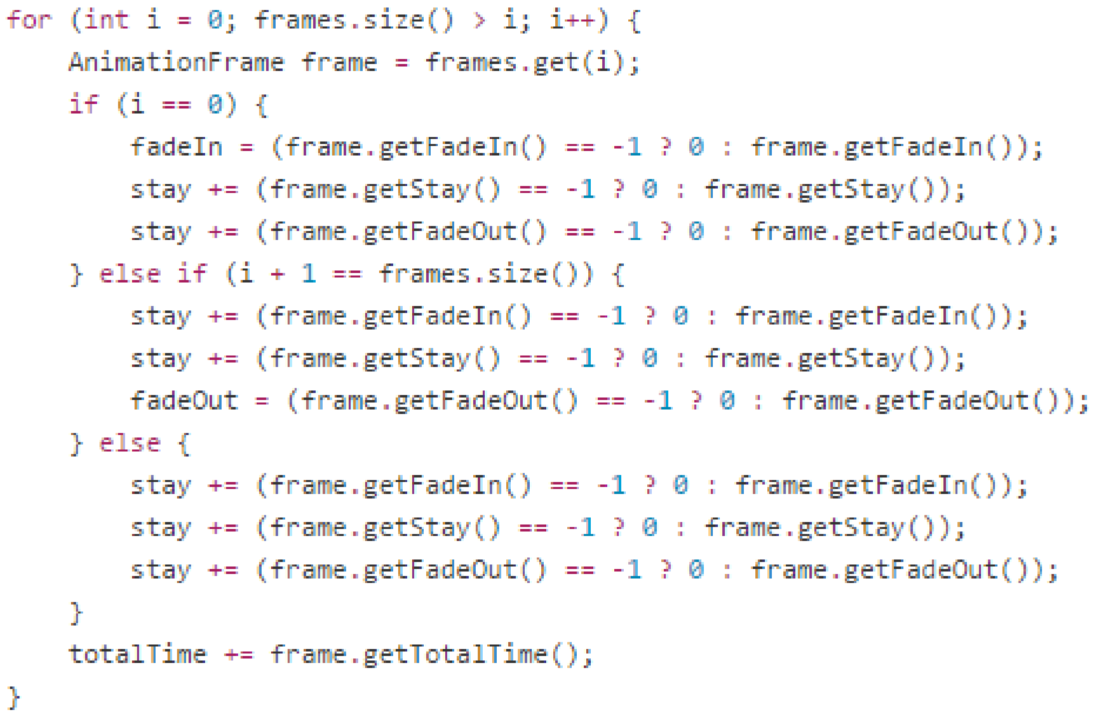


```
if(order.payment() != null)
    …
 else
    …
```



```
if(order.payment() == null)
    …
 else
     …
```

DRY violation - examples

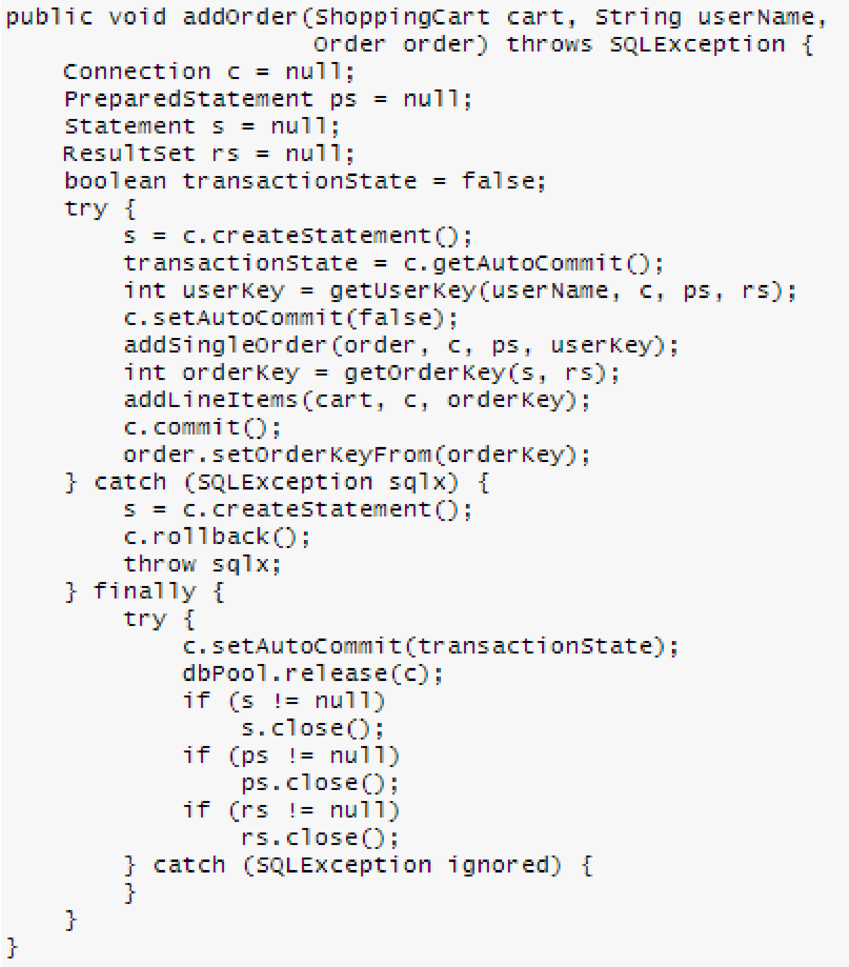


```java
for (int i = 0; frames.size() > i; i++) {
    AnimationFrame frame = frames.get(i);
    if (i == 0) {
        fadeIn = (frame.getFadeIn() == -1 ? 0 : frame.getFadeIn());
        stay += (frame.getStay() == -1 ? 0 : frame.getStay());
        stay += (frame.getFadeOut() == -1 ? 0 : frame.getFadeOut());
    } else if (i + 1 == frames.size()) {
        stay += (frame.getFadeIn() == -1 ? 0 : frame.getFadeIn());
        stay += (frame.getStay() == -1 ? 0 : frame.getStay());
        fadeOut = (frame.getFadeOut() == -1 ? 0 : frame.getFadeOut());
    } else {
        stay += (frame.getFadeIn() == -1 ? 0 : frame.getFadeIn());
        stay += (frame.getStay() == -1 ? 0 : frame.getStay());
        stay += (frame.getFadeOut() == -1 ? 0 : frame.getFadeOut());
    }
    totalTime += frame.getTotalTime();
}
```

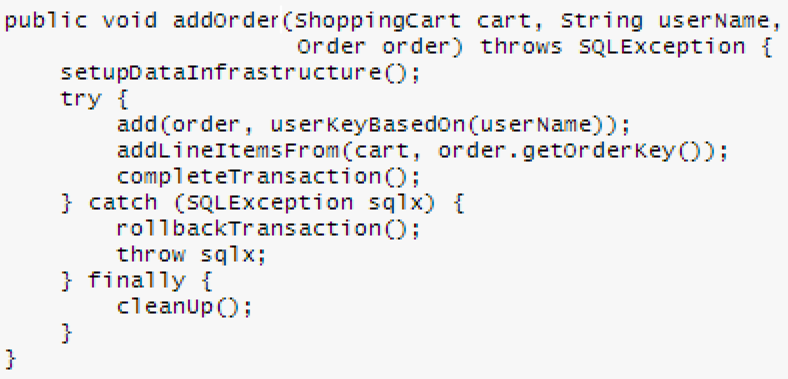
Functions and abstraction levels

- SLA(P) (Single Layer of Abstraction Principle)
  - All instructions within the function / method should operate at the same level of abstraction and be associated with a single task (single responsibility)

SLAP - example



```java
public void addOrder(ShoppingCart cart, String userName,
                     Order order) throws SQLException {
    Connection c = null;
    PreparedStatement ps = null;
    Statement s = null;
    ResultSet rs = null;
    boolean transactionState = false;
    try {
        s = c.createStatement();
        transactionState = c.getAutoCommit();
        int userKey = getUserKey(userName, c, ps, rs);
        c.setAutoCommit(false);
        addSingleOrder(order, c, ps, userKey);
        int orderKey = getOrderKey(s, rs);
        addLineItems(cart, c, orderKey);
        c.commit();
        order.setOrderKeyFrom(orderKey);
    } catch (SQLException sqlx) {
        s = c.createStatement();
        c.rollback();
        throw sqlx;
    } finally {
        try {
            c.setAutoCommit(transactionState);
            dbPool.release(c);
            if (s != null)
                s.close();
            if (ps != null)
                ps.close();
            if (rs != null)
                rs.close();
        } catch (SQLException ignored) {
        }
    }
}
```



```java
public void addOrder(ShoppingCart cart, String userName,
                     Order order) throws SQLException {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (SQLException sqlx) {
        rollbackTransaction();
        throw sqlx;
    } finally {
        cleanUp();
    }
}
```

http://www.ibm.com/developerworks/library/j-eaed4/

SOLID principles in object-oriented approach

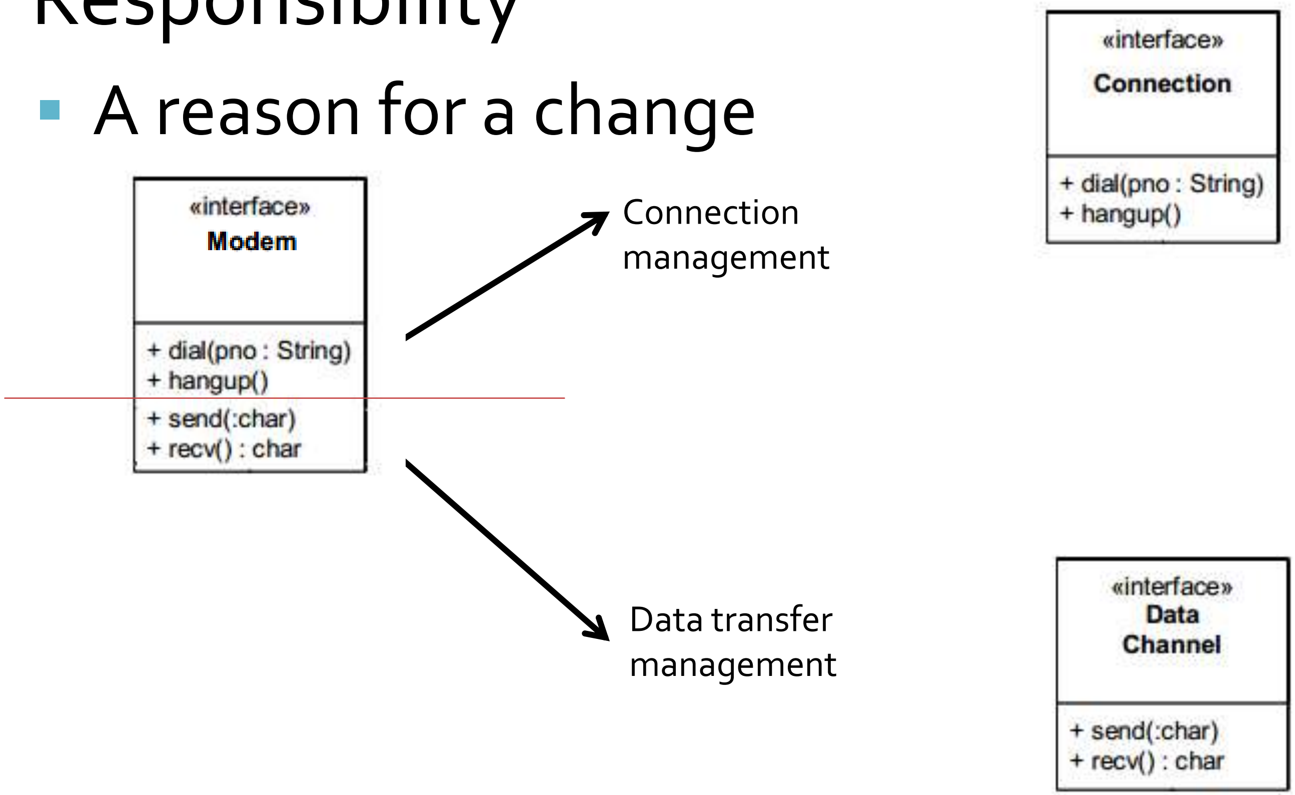
- **S**RP - Single Responsibility
- **O**CP - Open/Closed
- **L**SP - Liskov Substitution
- **I**SP - Interface Segregation
- **D**IP - Dependency Inversion

Principle

- Robert C. Martin
  http://www.objectmentor.com/resources/publishedArticles.html
- Strategy in Agile approach

SRP - Single Responsibility Principle

- Responsibility
  - A reason for a change



«interface»
Modem

+ dial(pno : String)
+ hangup()
+ send(:char)
+ recv() : char

Connection management

Data transfer management

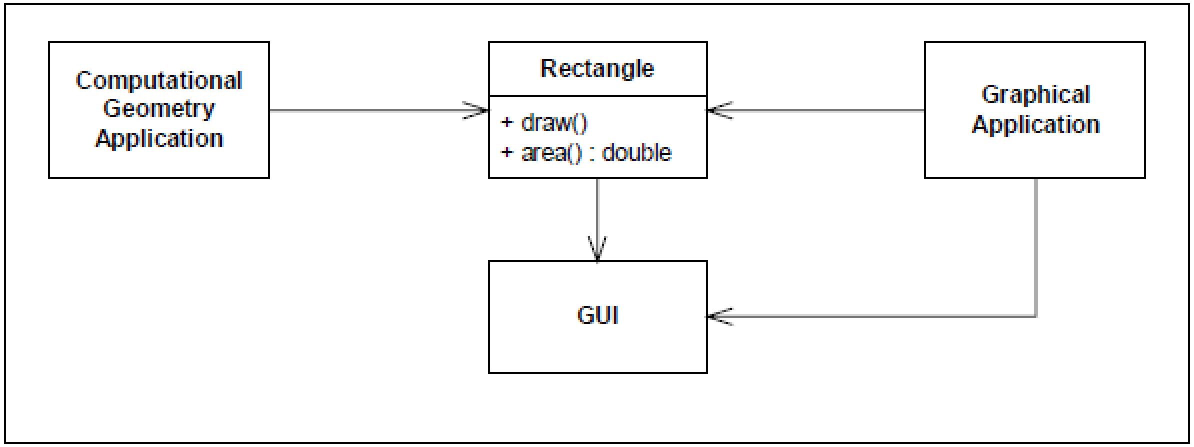
«interface»
Connection

+ dial(pno : String)
+ hangup()

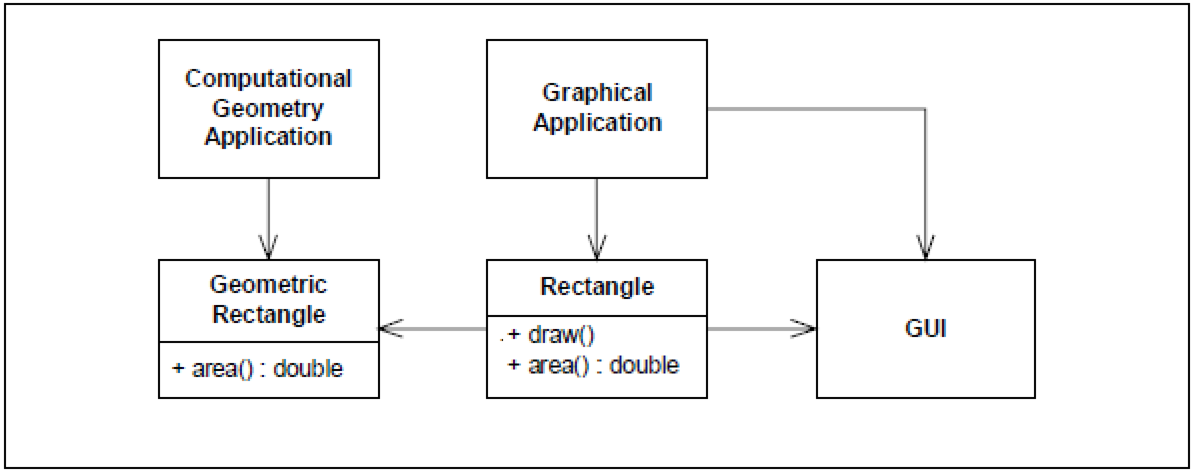
«interface»
Data Channel

+ send(:char)
+ recv() : char

SRP – Rectangle example



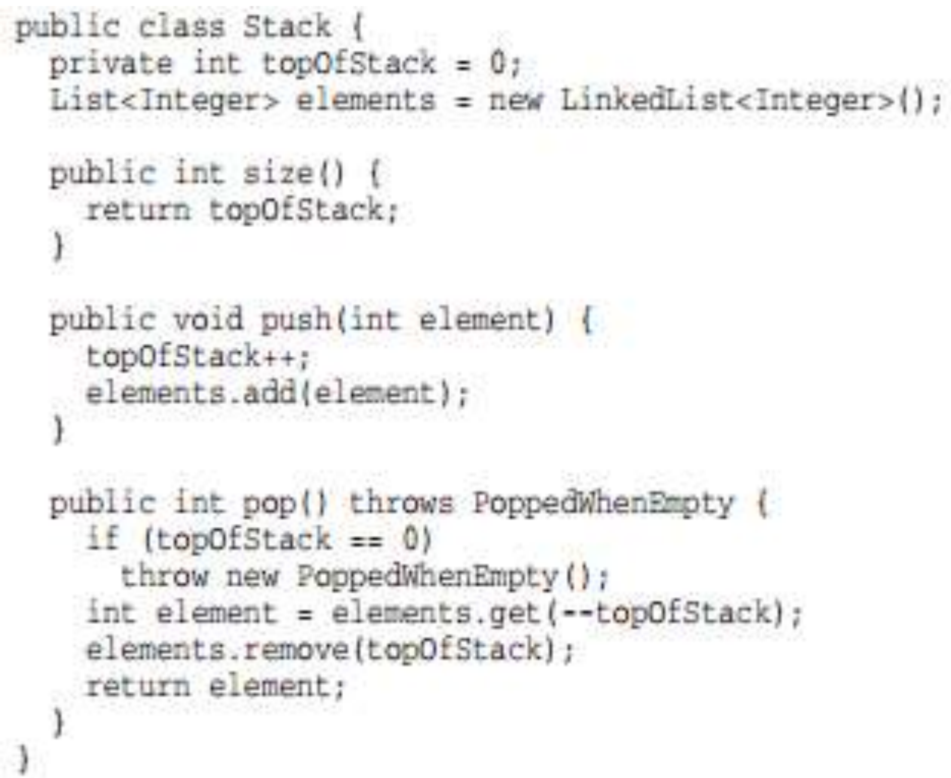
SRP



Cohesion and coupling

- Cohesion
  - Change in A allows a change in B, so both gain new value
  - Internally entity (e.g. class) should be as cohesive as possible (high cohesion) – supports SRP
    - Class instances should have a small number of properties.
    - Each object method should use one or more of this properties.
- Coupling
  - Change in B is enforced be a change in A
  - Entities (e.g. classes) should be as loosely coupled (loose coupling)
    - Interfacas, minimalization of dependencies

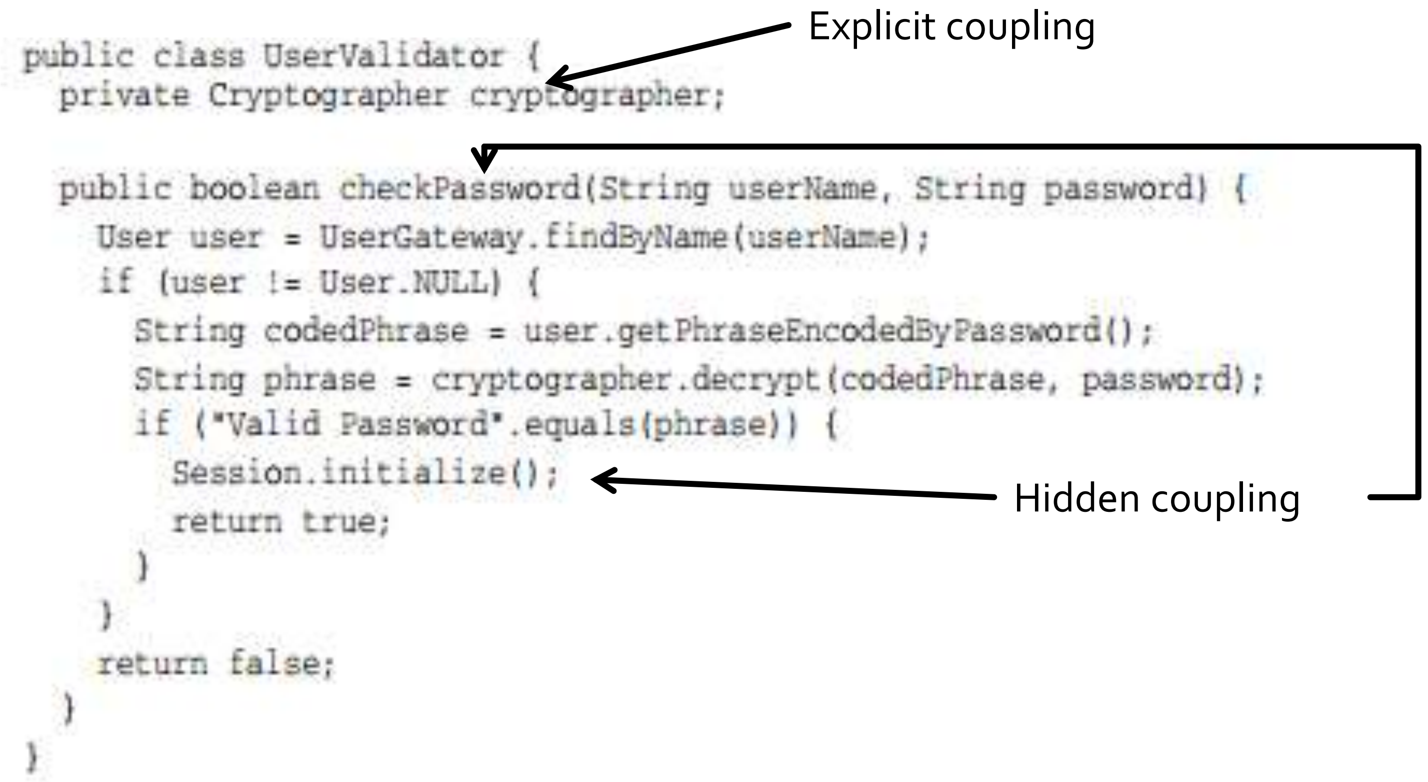
Cohision



```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship

Hidden couplings



```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Explicit coupling

Hidden coupling

Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship

Open/close principle

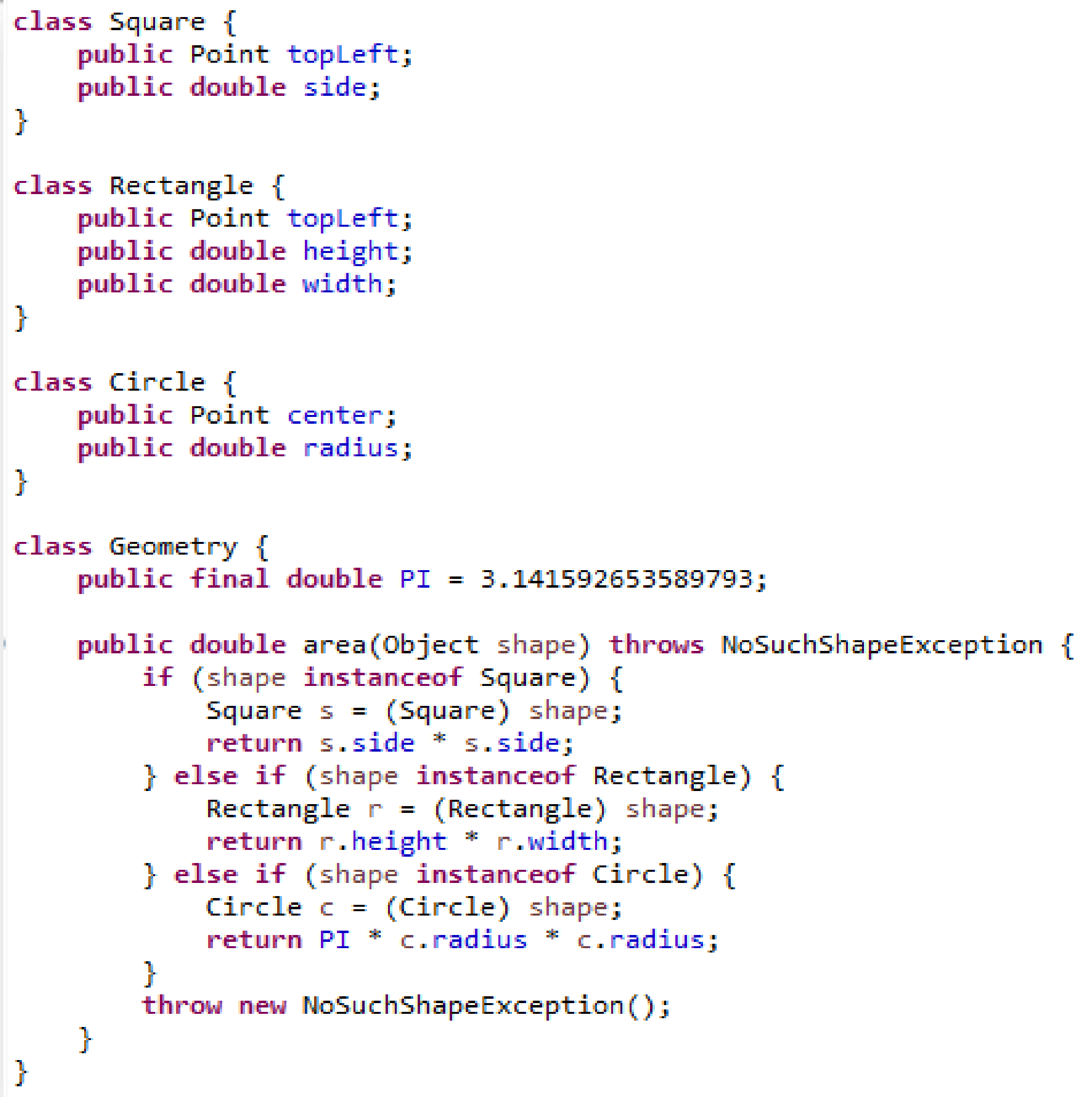
- Open-closed principle [Bertrand Meyer, 1988]:
- *„Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*
  - Ideally, the introduction of new features should not require any modifications to existing code.

OCP and Agile development

- "Opening" to all kinds of changes is not possible.
- The programmer's decision in this area may be inappropriate.

- Agile methodology can help to reveal important fields for changes at an early stage.
- You can then "open" for these changes (refactoring).

Structural/Procedural Shapes

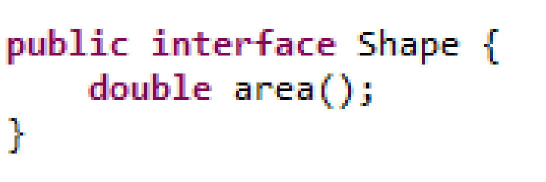


```
class Square {
    public Point topLeft;
    public double side;
}

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

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

class Geometry {
    public 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;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.height * r.width;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

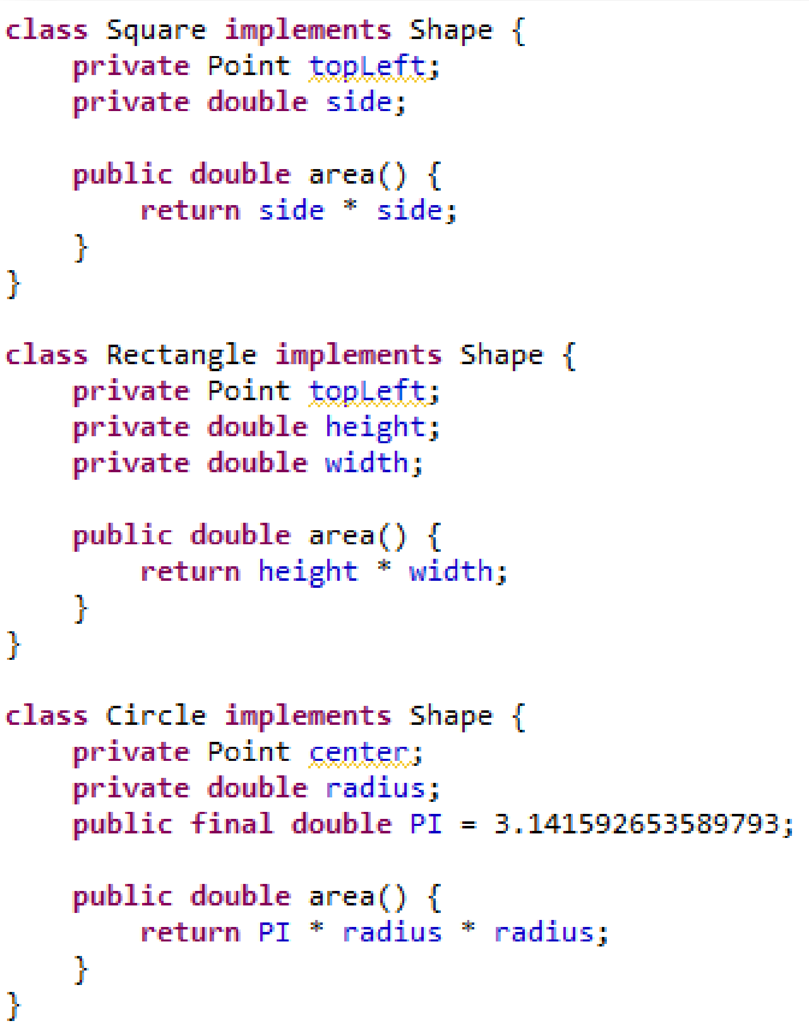
Data structures

Procedure

Object-Oriented Shapes



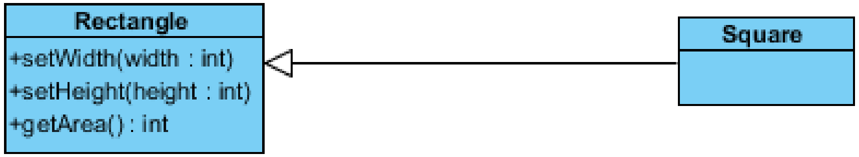
```java
public interface Shape {
    double area();
}
```

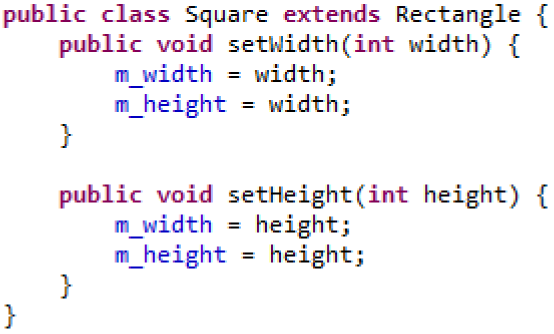


```java
class Square implements Shape {
    private Point topLeft;
    private double side;

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

class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

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

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

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

Liskov Substitution Principle

- The instructions that control an object using a reference to the base class should be able to use a derived class object without any knowledge of existence of derived class.
- In object oriented approch inheritance relation (*is a type of*) refers to behaviour.

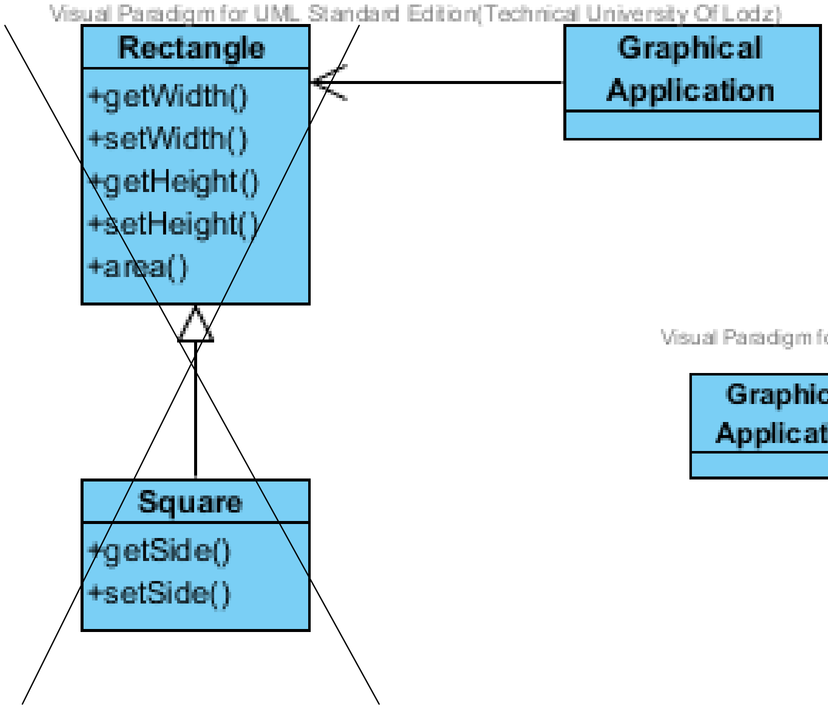


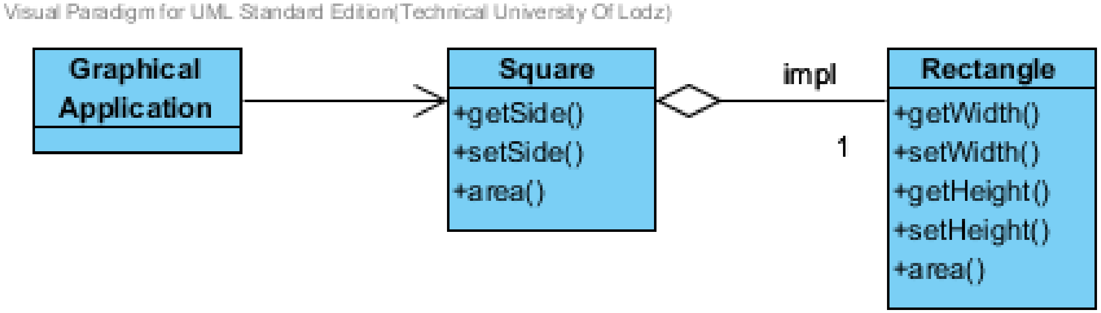


```
public class Square extends Rectangle {
    public void setWidth(int width) {
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height) {
        m_width = height;
        m_height = height;
    }
}
```

LSP – subclasses and „contract"

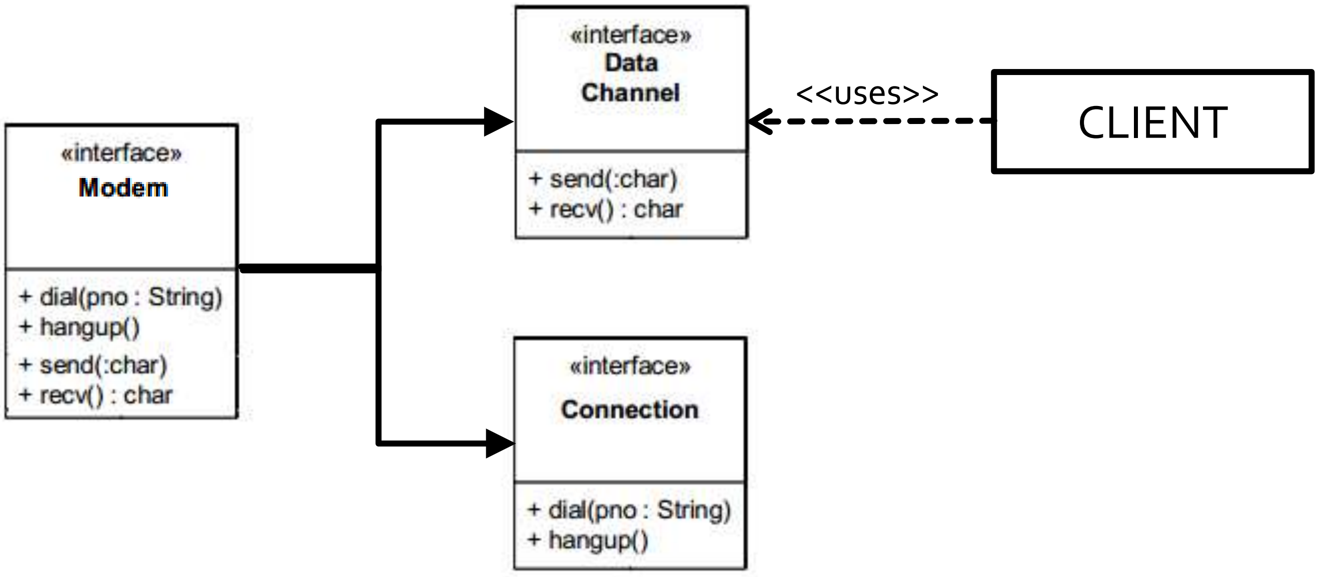
The basis for a good object design is a clearly defined and coherent *contract* between cooperating objects.





LSP Violation

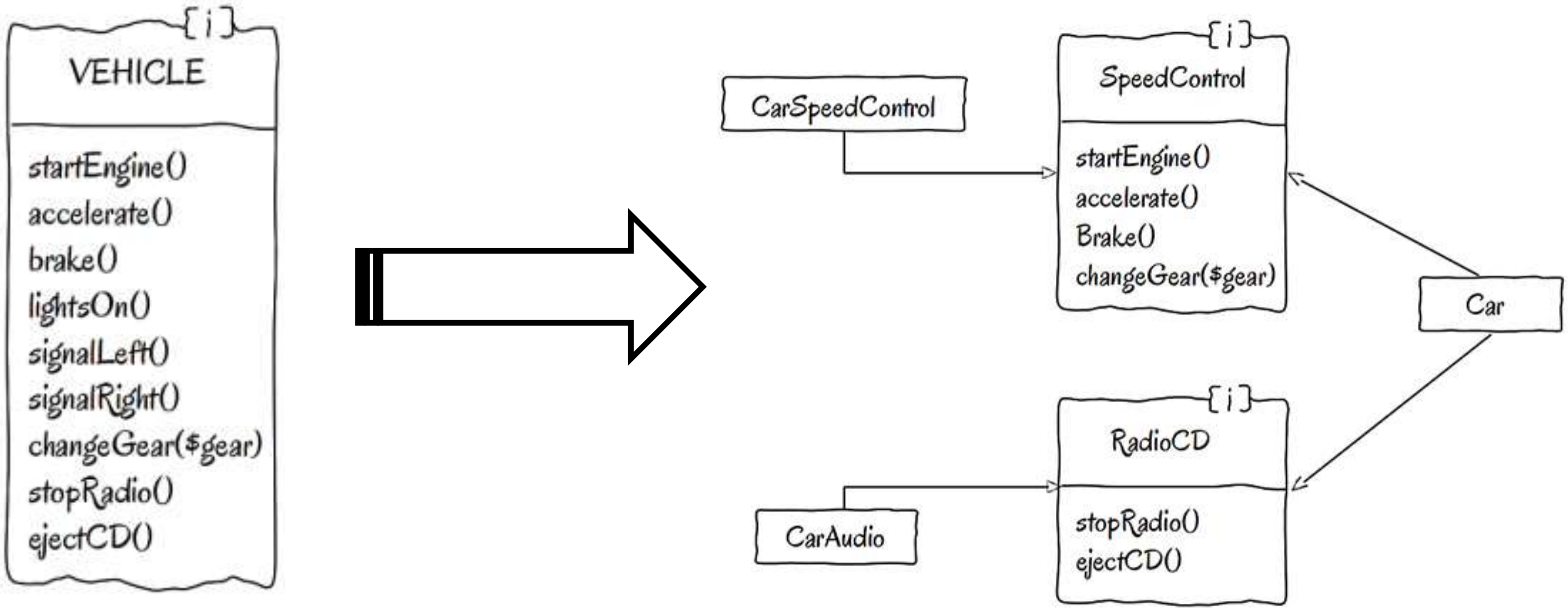
- If inheriting, we make sure that the existing client code will have to check with what type it has to deal with.



Interface Segregation Principle

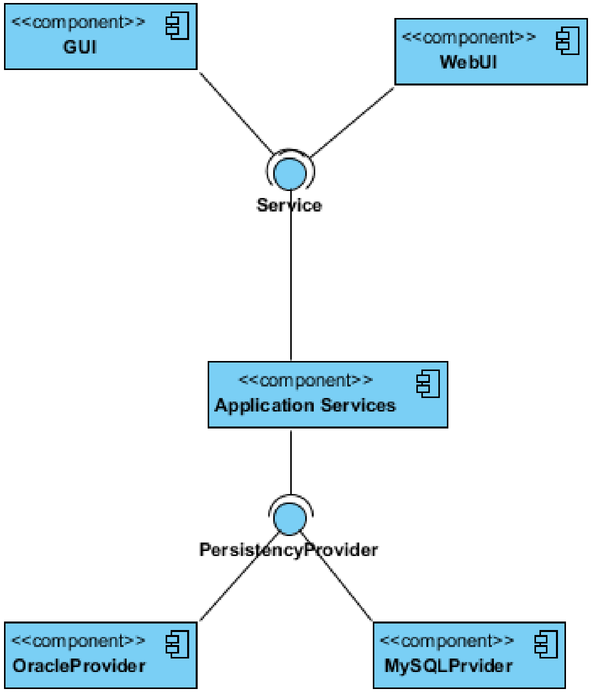
- Client should not be forced to depend on interfaces it does not use.
- SRP for interfaces

https://code.tutsplus.com/tutorials/solid-part-3-liskov-substitution-interface-segregation-principles--net-36710

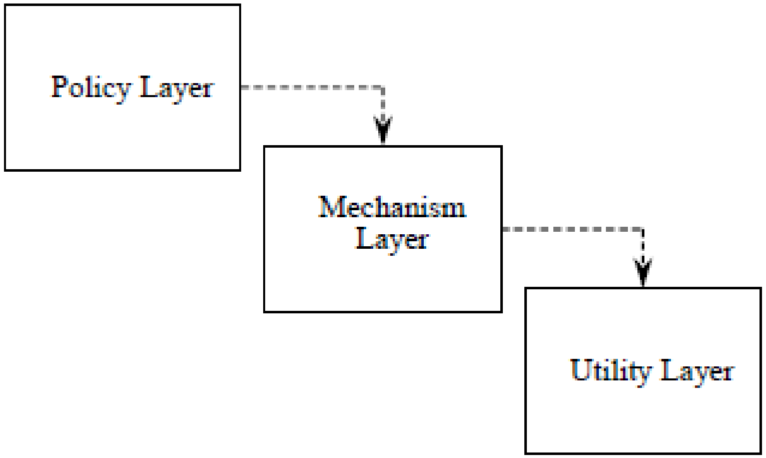


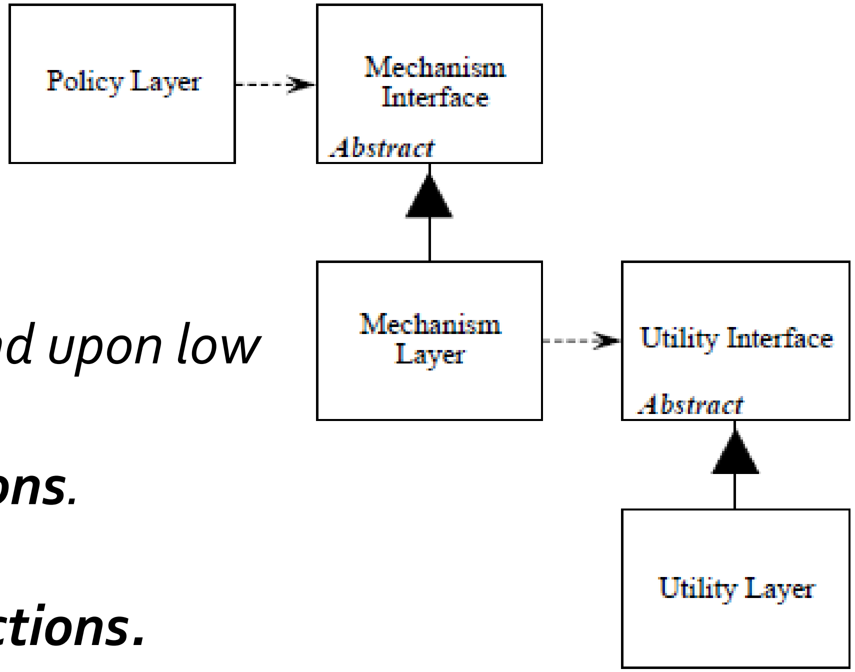
Dependency Inversion Principle

- Higher level modules should not be depend of lower lever modules. Both should depend of abstractions (an interface)



DIP – layered architecture





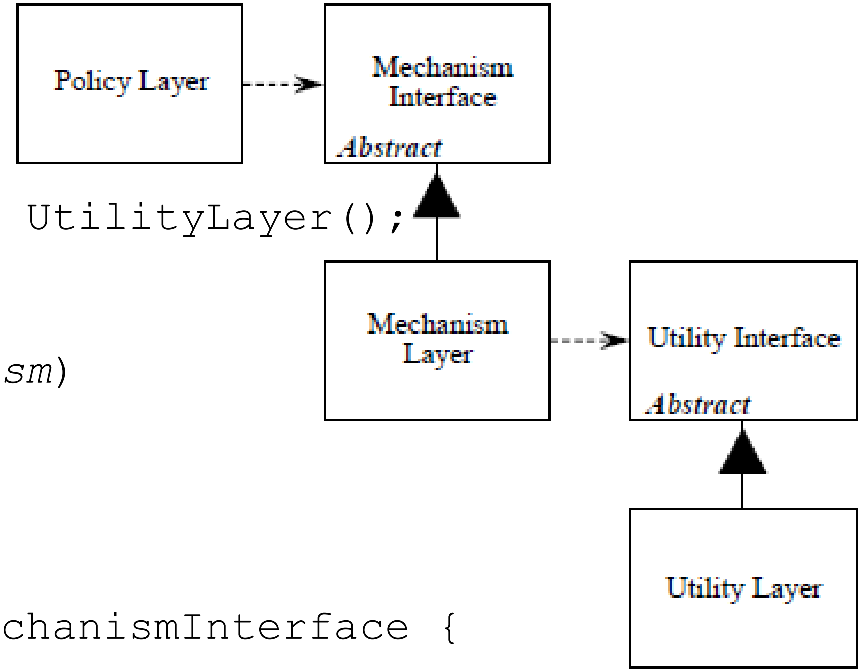
High level modules should not depend upon low level modules.
*Both should depend upon **abstractions**.*

Details** should depend upon **abstractions.
Abstractions** should **not** depend upon **details.

Robert C. Martin

DIP – example of naïve injection using class constructors

```
class DependencyManager {
   …
   static Policy getPolicy() {
      UtilityInterface utility = new UtilityLayer();
      MechanismInterface mechanism =
         new MechanismLayer(utility);
      return new PolicyLayer(mechanism)
   }
   …
}

class MechanismLayer implements MechanismInterface {

   UtilityInterface utility;

   MechanismLayer(UtilityInterface utility) {
      this.utility = utility;
   }
   …
}
```



Dependency Injection methods

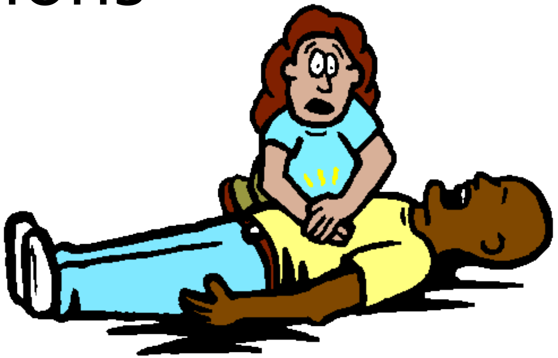
- The basic methods of Dependency Injection use:
  - setters and constructors.
- Their call can be automatic based on a specific configuration, e.g.:
  - IoC containers Spring framework,
  - Pico containers.
- There is also a method for injecting the interface injection (semi-automatic).

Other rules

1. Program to interfaces, not implementations.
2. Encapsulate what varies.
3. Favor composition over inheritance.
4. Use design pattern wherever it is possible.
5. Strive for loosely couple designs between objects that interact.
6. KISS – simplicity, simplicity, …

But to write a good code

- Practice, practice, look for patterns.
- But I practice, I code in work…
- Exercise does not necessarily have to be revealing - the pursuit of perfection requires repeated repeating the same actions during exercises.



References

- hasschapman.blogspot.com/2011/11/visualising-your-technical-debt.html
- dearjunior.blogspot.com/2012/03/dry-and-duplicated-code.html
- c2.com/cgi/wiki?CouplingAndCohesion
- www.slideshare.net/skarpushin/solid-ood-dry
- www.oodesign.com/liskov-s-substitution-principle.html
- www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles
- www.odi.ch/prog/design/newbies.php
- *The Pragmatic Programmer,* Andrew Hunt and David Thomas, 1999
- Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship, 2008

A must read!

- Andy Hunt, **Pragmatic Thinking and Learning: Refactor Your Wetware (The Pragmatic Programmer series)**