

SOLID, DRY, SLAP

Design principles

presenter: Sergey Karpushin



Elite Software R&D Services
Since 1990

Agenda

- Justification
- DRY
- SLAP
- **Single Responsibility**
- **Open/Closed**
- **Liskov substitution**
- **Interface segregation**
- **Dependency inversion**

Justification 1/4

Where you spend time?

Fixing tons of bugs

Implementing new functionality

How do you feel about
new change requests?

I'm scared! This will
break something

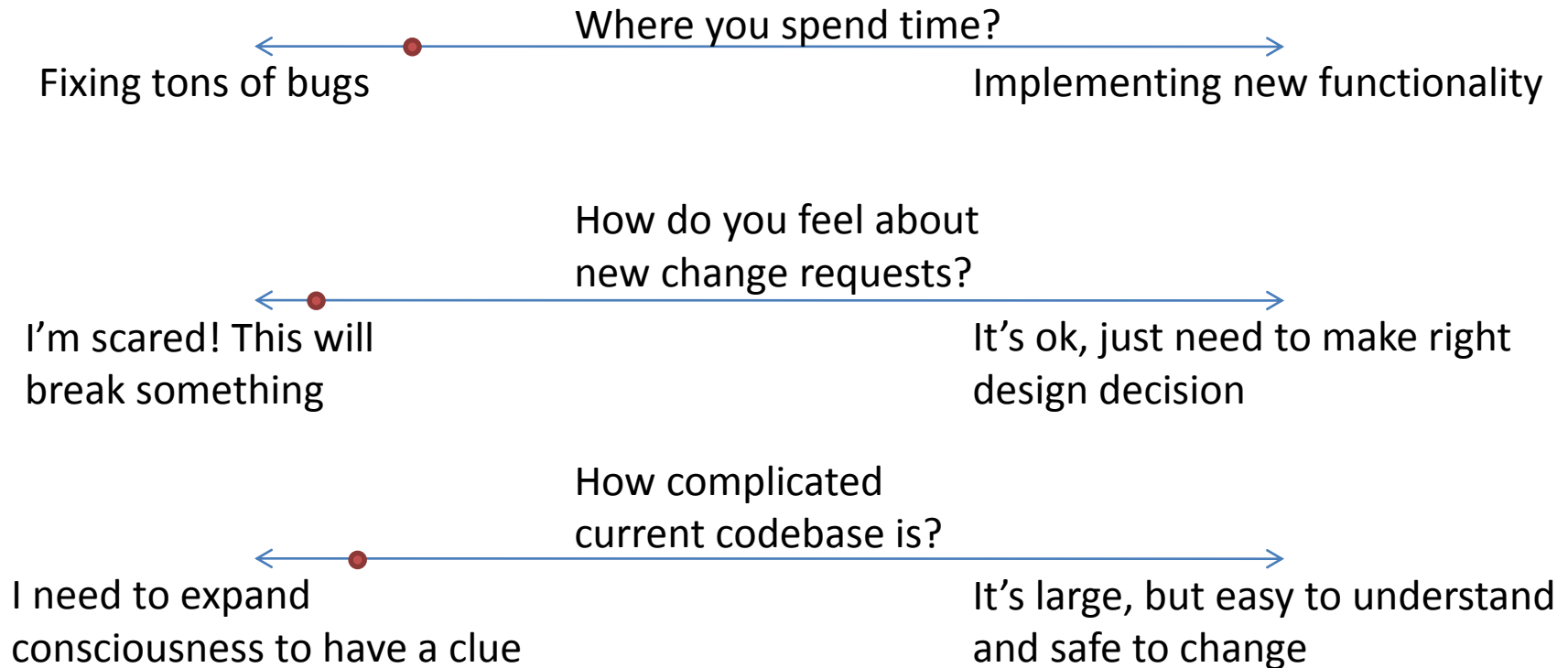
It's ok, just need to make right
design decision

How complicated
current codebase is?

I need to expand
consciousness to have a clue

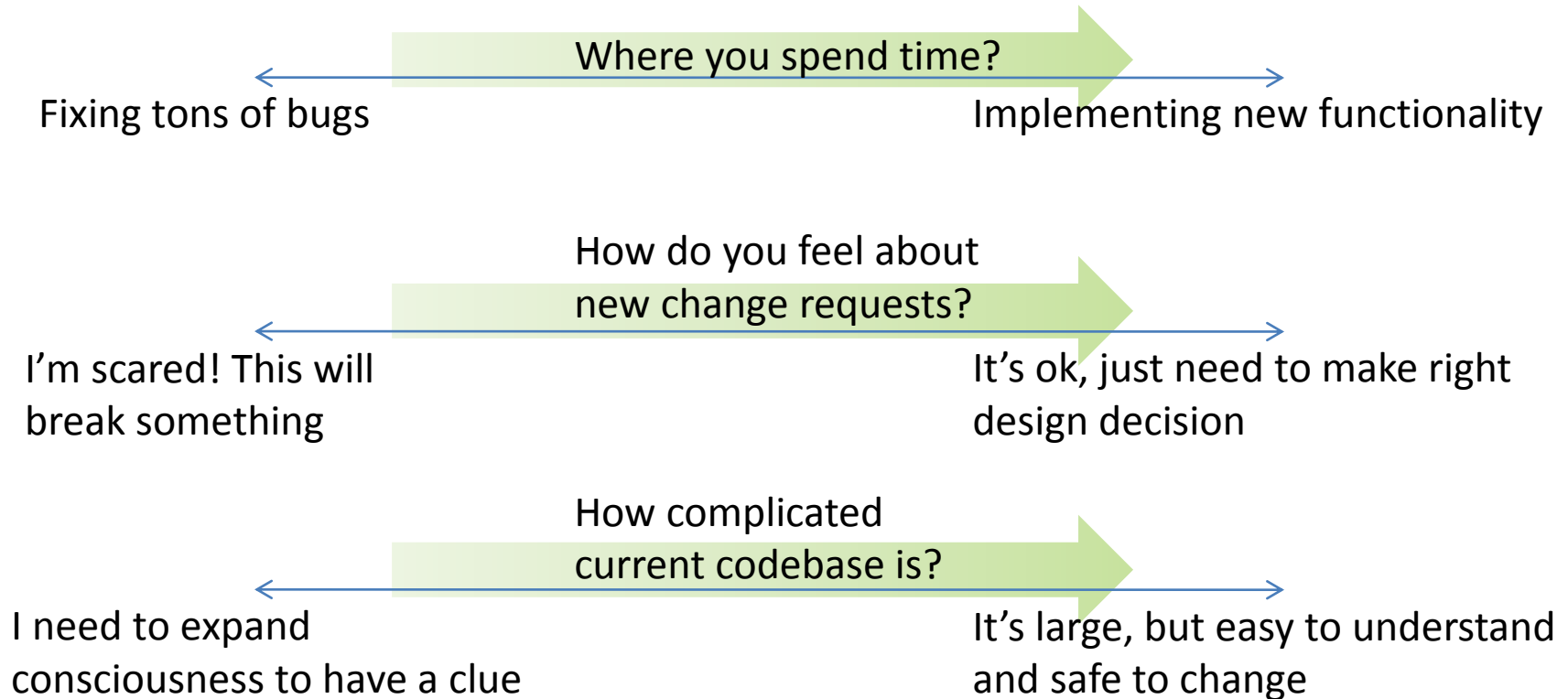
It's large, but easy to understand
and safe to change

Justification 2/4



Signal words: urgent, stress, overtime, chaos, more than 100 open bugs, "who wrote this crap?"

Justification 3/4



Signal words: ahead of schedule, only 3 open bugs, thank you

Justification 4/4

OOD & other related principles

- Takes time to understand and feel for the first time
- Relatively easy to apply later

Disclaimer:

NAMING CONVENTION

Naming convention

- Design principles are common for all Object-Oriented languages
- Java naming conventions used within this presentation
- Name example for interfaces, DTOs and “simple” classes:
UserService
- Name example for classes which implements interfaces
UserServiceImpl
- All variable and parameter names example
camelCaseName

DRY + SLAP + SOLID

DON'T REPEAT YOURSELF

DRY 1/15

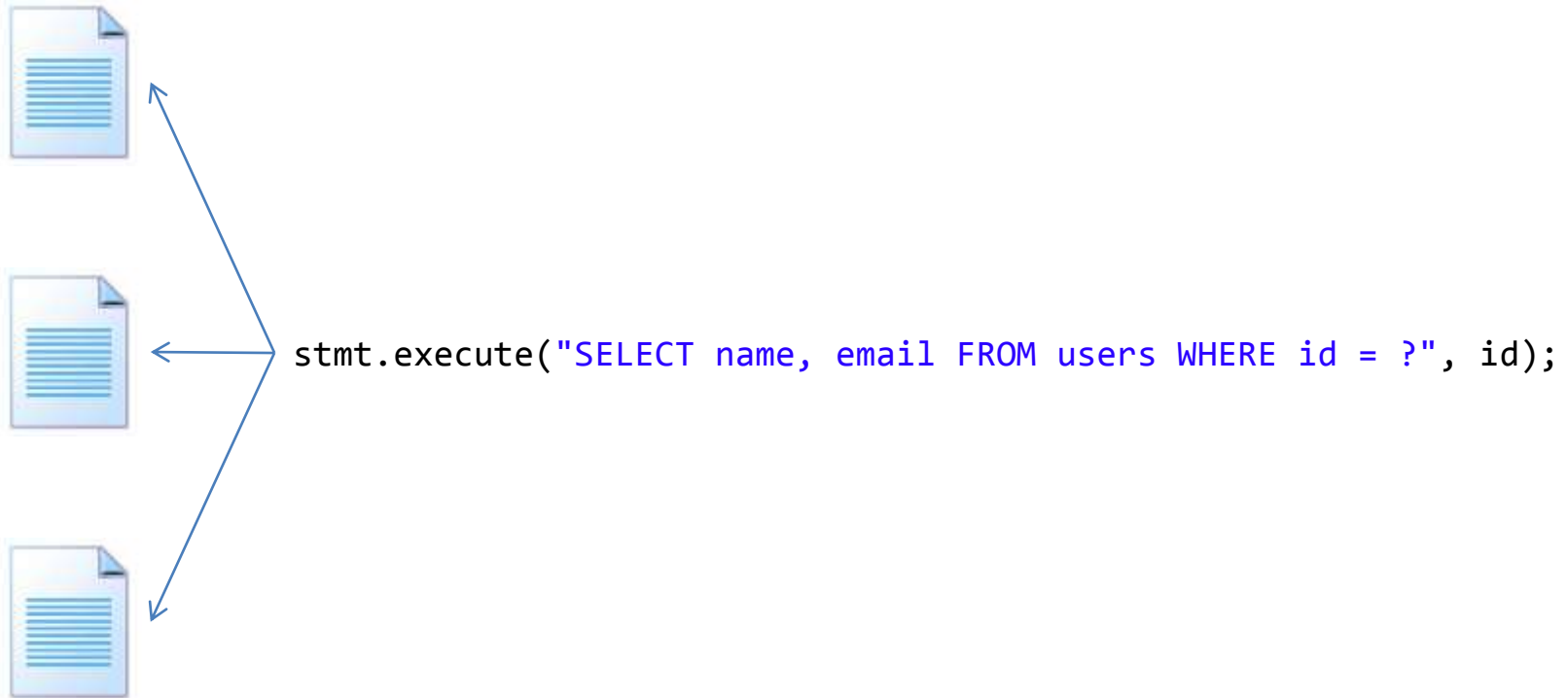
- **Do not Repeat Yourself**
 - Reuse your code, do not duplicate it
 - Assign clear names
 - Choose right location
- Clear names and right location allows to identify duplicate candidates

DRY 2/15 – reuse

- Maintain “single source of truth”
 - Change once
 - Test once
- Do not confuse with abstractions, it's different

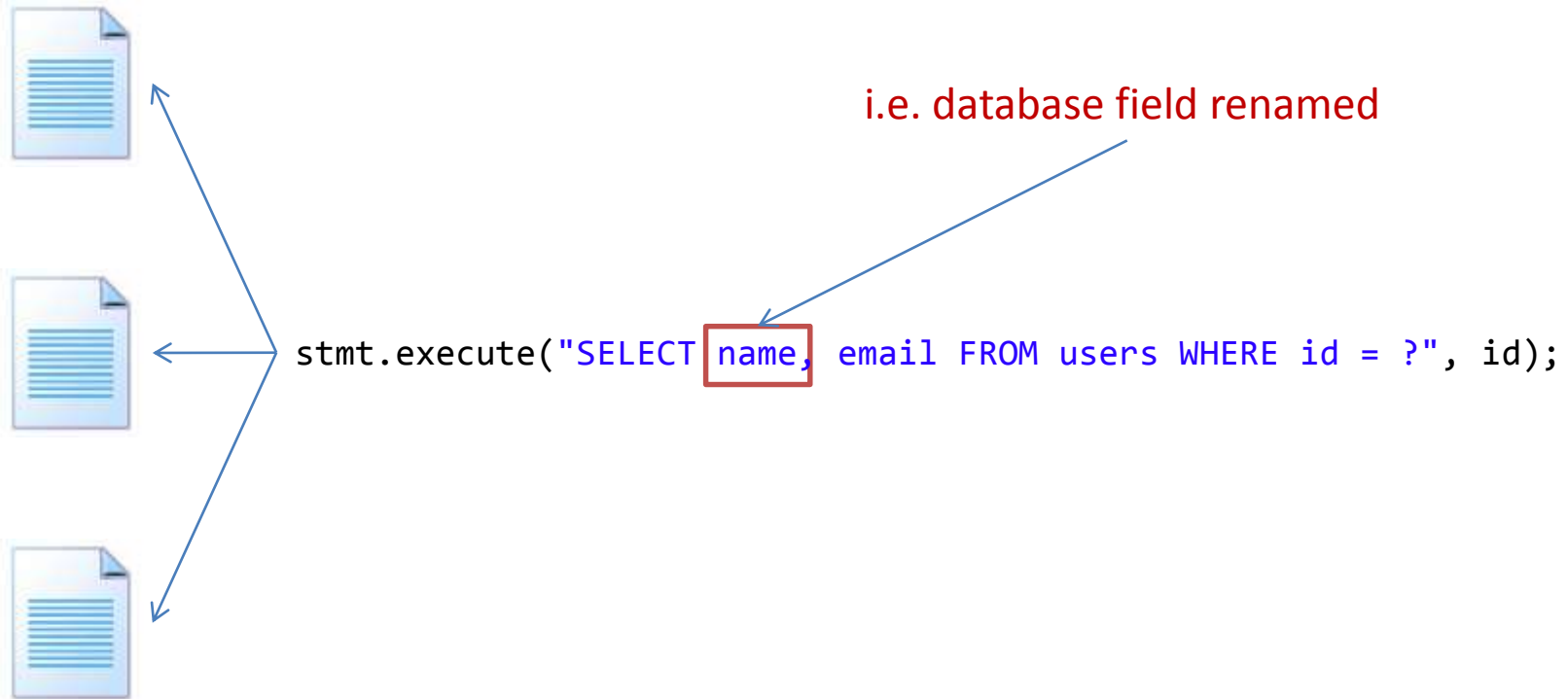
DRY 3/15 – reuse

- Same code in multiple places



DRY 4/15 – reuse

- A change required?



DRY 5/15 – reuse

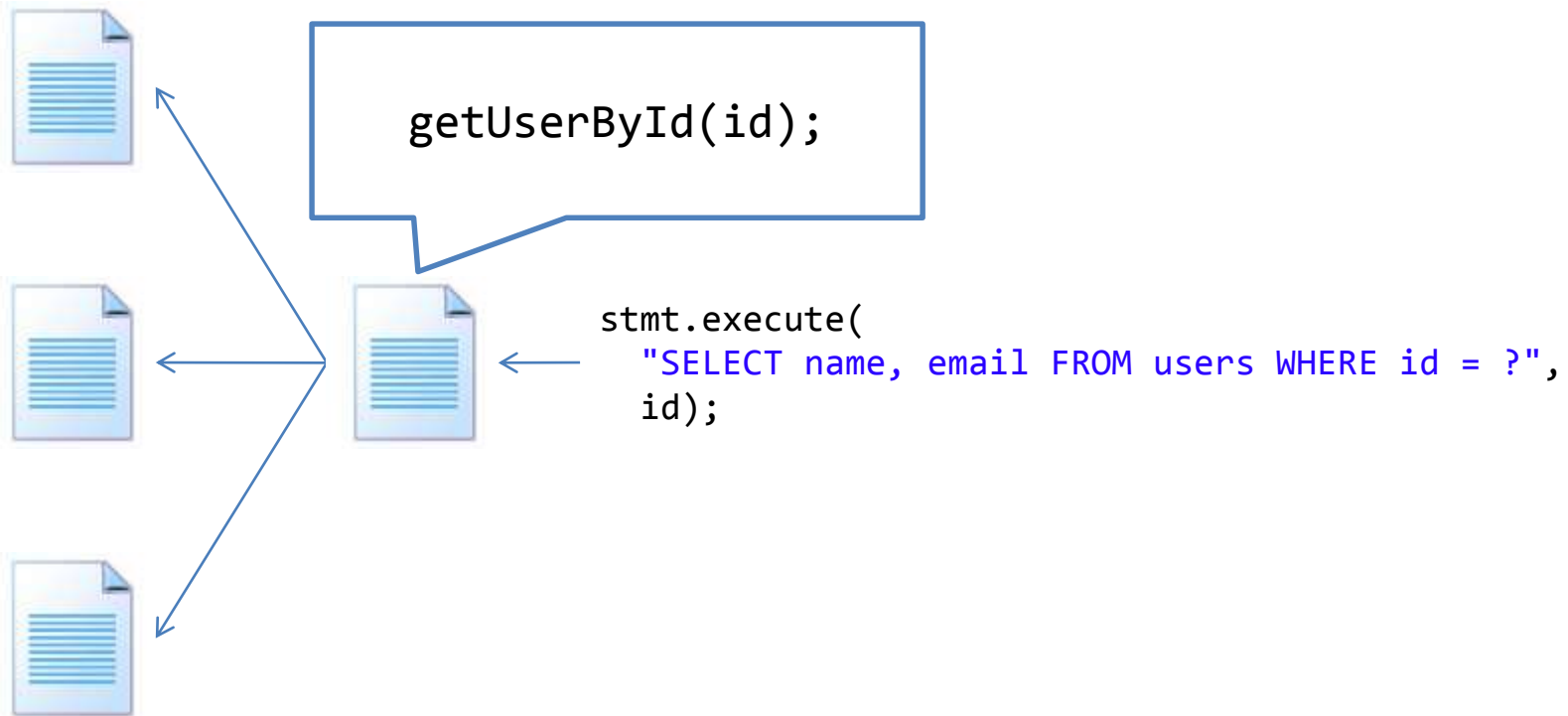
- A multiple changes required!



`stmt.execute("SELECT last_name, email FROM users WHERE id = ?", id);`

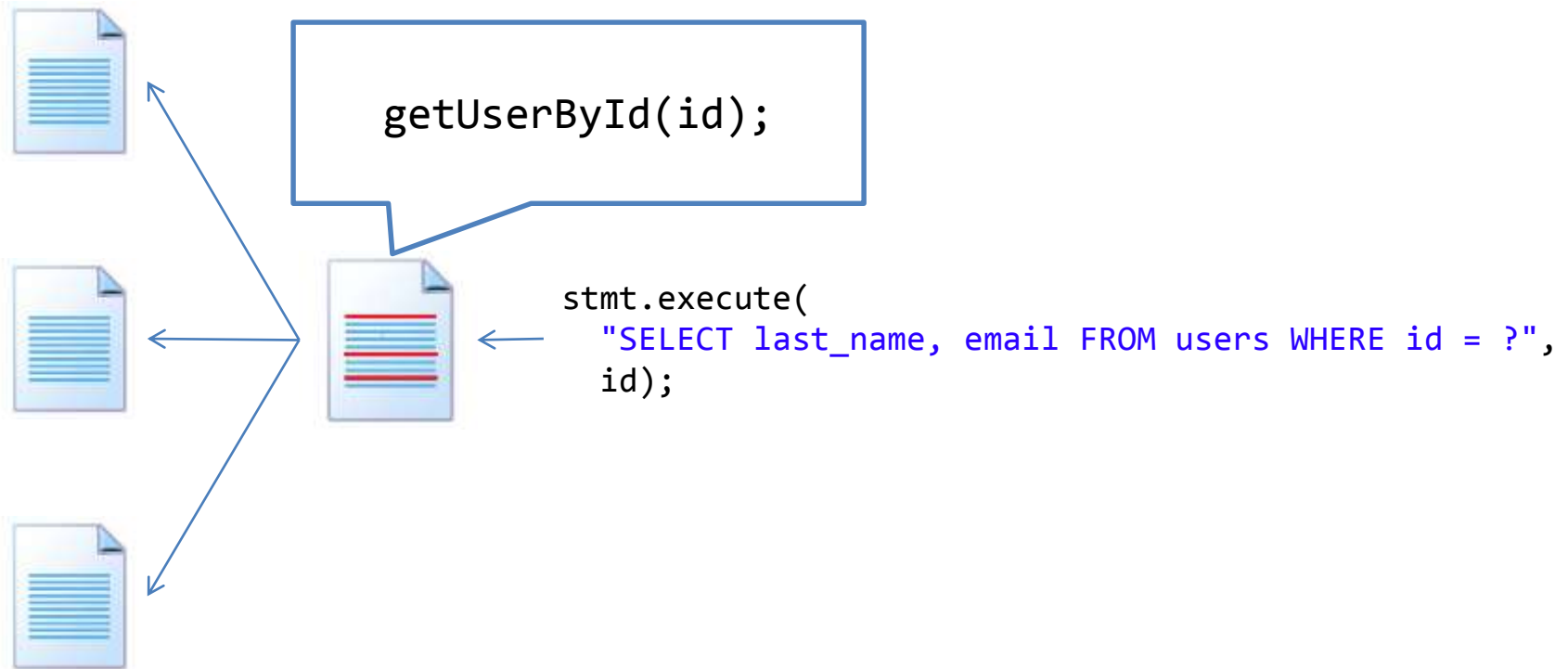
DRY 6/15 – reuse

- What if code is in one place?



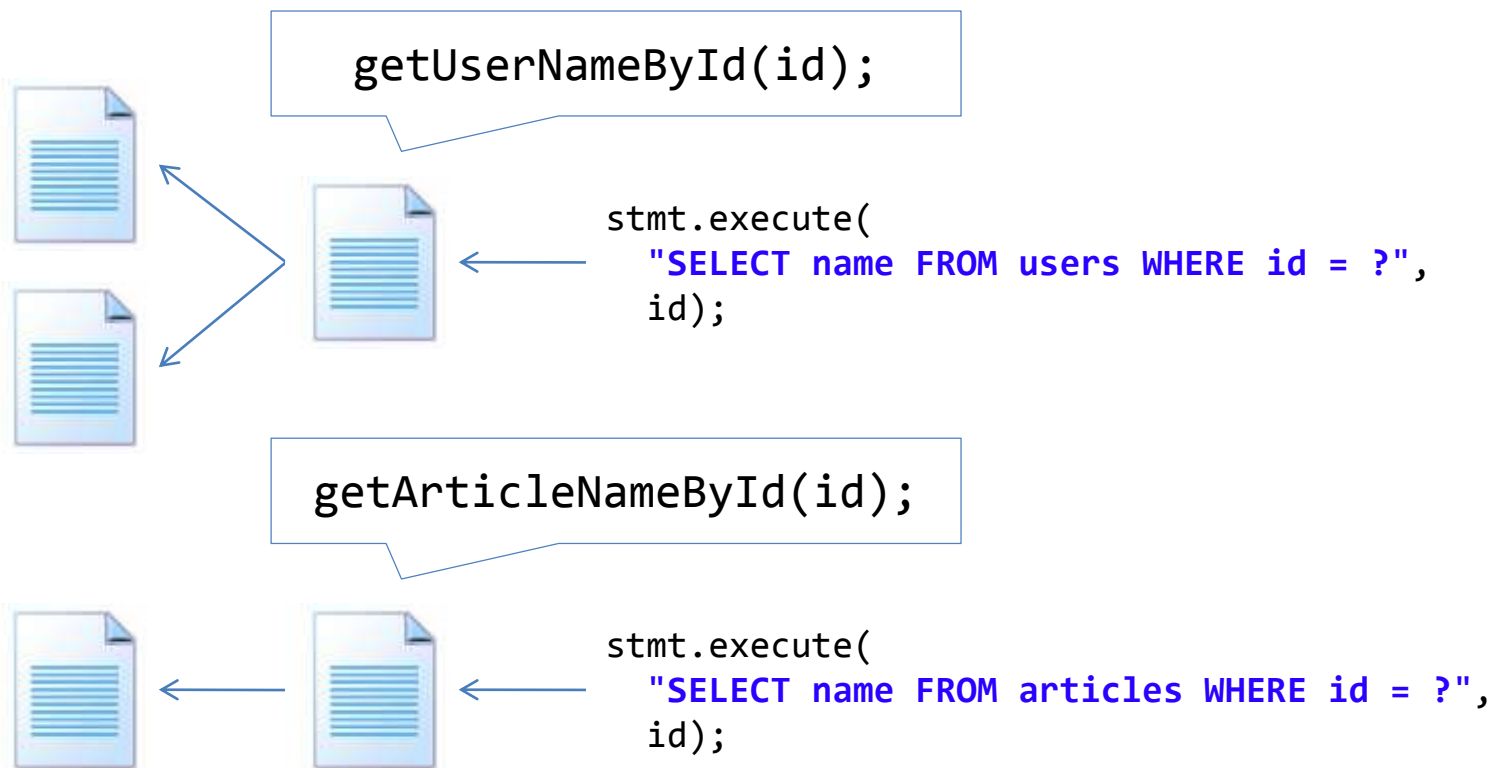
DRY 7/15 – reuse

- Exactly! 1 change required.



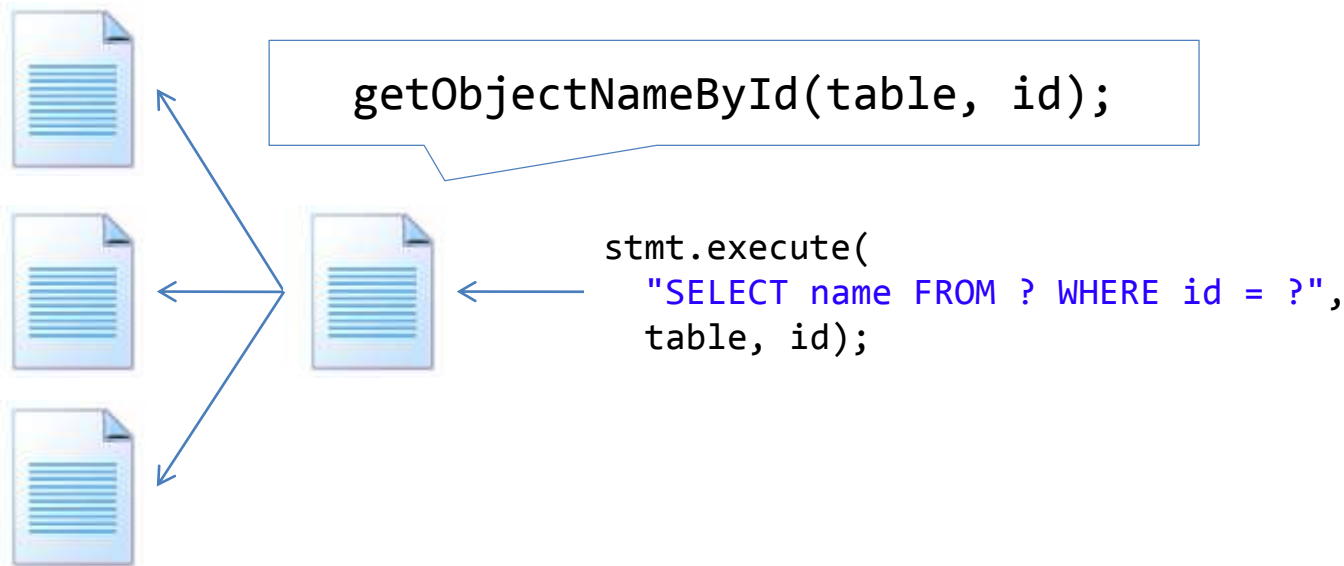
DRY 8/15 – confusing w/ abstractions

- Suppose we have Users and Articles



DRY 9/15 – confusing w/ abstractions

- Hey! It was a duplicate! We got rid of it!

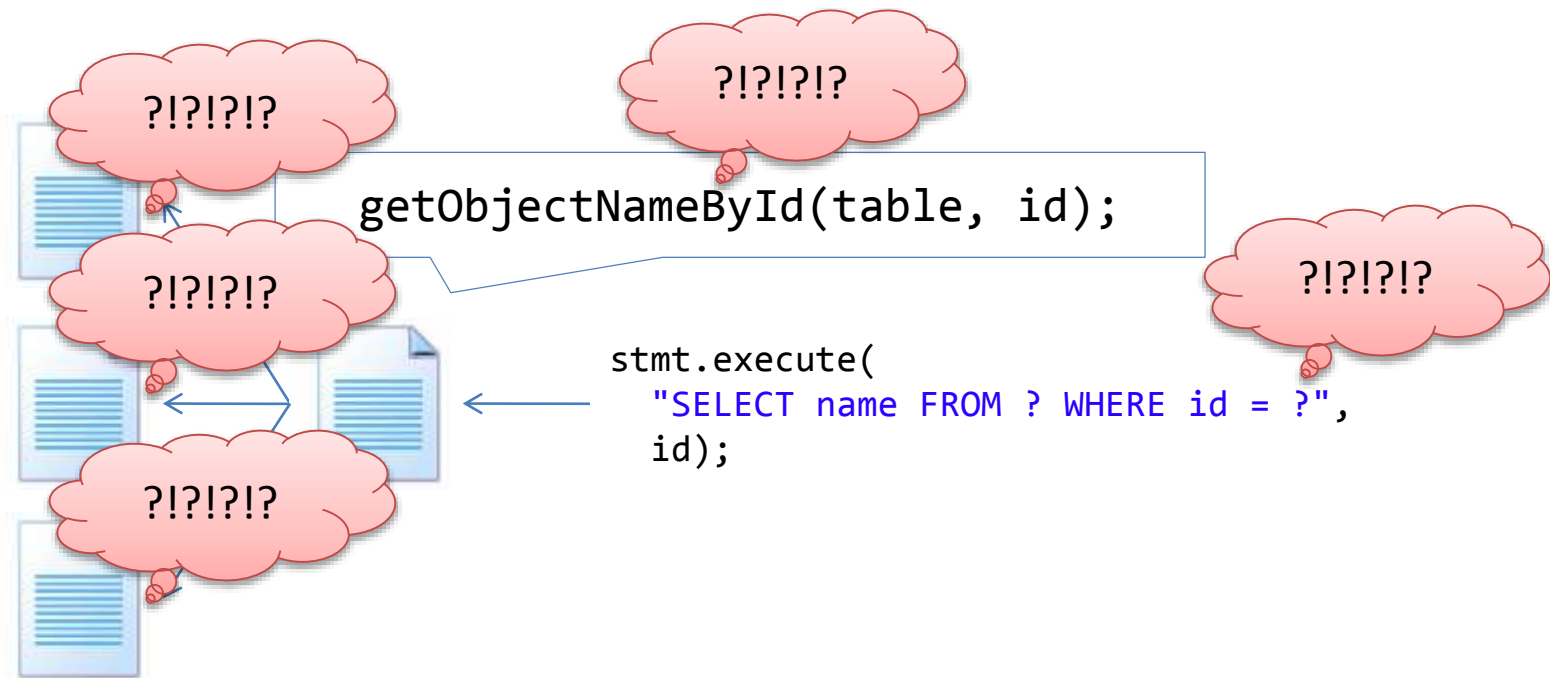


DRY 10/15 – confusing w/ abstractions

- Incoming change request: “Article name must be displayed along with author and date”

DRY 11/15 – confusing w/ abstractions

- Complete mess, this is what you end up with



DRY 12/15 – confusing w/ abstractions

- For abstractions recommended approach is
 - Write
 - Copy & Paste
 - Only then – extract abstraction

DRY 13/15 – clear names

- Q: How can I determine if name is “clear”?
- A: If javadoc will look redundant
- Q: How to pick clear name?
- A: Write java doc, get essence of it, use as name
- Q: It sometimes so complicated to pick short but still clear name!
- A: Be honest with yourself, it happens rarely. In such case write descriptive javadoc

DRY 14/15 – clear names

An example

- You can say almost nothing about it:

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


- But this one is pretty descriptive:

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

DRY 15/15 – right location

- No one will guess that it might be here!

```
package ru.it.projects.umms.dom.users;  
  
public class ArrayUtils {  
  
}
```



DRY + SLAP + SOLID

**SINGLE LAYER OF ABSTRACTION
PRINCIPLE**

SLAP 1/3

- **Single Layer of Abstraction Principle**
...for methods
- Have common things with DRY, SRP, ISP
- Basic idea is:
 - Do not write endless methods which logic is hard to follow.
 - All instructions for each method should operate on same level of abstraction and related to “one task”

SLAP 2/3 (violated)

```
private UserInfo retrieveUserInfo(String userToken) {
    String userTokenParts[] = userToken.split(":");
    long userId;
    if (userTokenParts.length == 1) {
        userId = Long.parseLong(userToken);
    } else {
        if ("legacy".equals(userTokenParts[0])) {
            userId = Long.parseLong(userTokenParts[1]);
        } else {
            userId = Long.parseLong(userTokenParts[0] + userTokenParts[1]);
        }
    }
    String url = userServiceBasePath;
    if ("legacy".equals(userTokenParts[0])) {
        url += "/v1/getUserData?userId=" + userId;
    } else {
        url += "/v2/users/" + userId;
    }
    try {
        HttpResponse getResponse = client.execute(url);
        final int statusCode = getResponse.getStatusLine().getStatusCode();
        if (statusCode != HttpStatus.SC_OK) {
            Log.w(getClass().getSimpleName(), "Error " + statusCode + " for URL " + url);
            return null;
        }
    }
```

SLAP 3/3 (following)

```
private UserInfo retrieveUserInfo(String userToken) {  
    long userId = extractUserIdFromToken(userToken);  
    if (isLegacyUser(userToken)) {  
        return retrieveLegacyUserInfo(userId);  
    } else {  
        return retrieveUserInfo(userId, extractUserDomainIdFromToken(userToken));  
    }  
}
```

- The only concern: you'll have more methods. Yes, but each of them is clear and damn simple to follow (thus maintain)

SOLID

OPEN-CLOSED PRINCIPLE

OCP 1/11 – general idea

Open-Closed Principle

- Class should be
 - Open for extension
 - Closed for modification
- Effects
 - Increased stability – existing code (almost) never changes
 - Increased modularity, but many small classes

OCP 2/11 – i.e.

- Customer brings new user story:
As the user I want to see results of past games. Let's keep up with NHL & NBA games

OCP 3/11 – i.e.

```
public class SportInfoParser {  
    public SportInfo parseSportInfo(String[] sportInfoArray) {  
        if ("nba".equals(sportInfoArray[0])) {  
            NbaSportInfo nbaSportInfo = new NbaSportInfo();  
            Map<Long, Integer> scores = new HashMap<Long, Integer>();  
            scores.put(Long.parseLong(sportInfoArray[12]), Integer.parseInt(sportInfoArray[13]));  
            NbaSportInfo.setScores(scores);  
            return nbaSportInfo;  
        } else if ("nhl".equals(sportInfoArray[0])) {  
            NhlSportInfo nhlSportInfo = new NhlSportInfo();  
            nhlSportInfo.setSlapShotCount(1);  
            return nhlSportInfo;  
        }  
    }  
}
```


OCP 4/11 – i.e.

- Customer feedback:
Thank you, it works well!
- And brings another user story:
As the user I want to see results of MLB games

OCP 5/11 – i.e.

```
public class SportInfoParser {  
    public SportInfo parseSportInfo(String[] sportInfoArray) {  
        if ("nba".equals(sportInfoArray[0])) {  
            NbaSportInfo nbaSportInfo = new NbaSportInfo();  
            Map<Long, Integer> scores = new HashMap<Long, Integer>();  
            scores.put(Long.parseLong(sportInfoArray[12]), Integer.parseInt(sportInfoArray[13]));  
            NbaSportInfo.setScores(scores);  
            return nbaSportInfo;  
        } else if ("nhl".equals(sportInfoArray[0])) {  
            NhlSportInfo nhlSportInfo = new NhlSportInfo();  
            nhlSportInfo.setSlapShotCount(1);  
            return nhlSportInfo;  
        } else if (sportInfoArray[0].equalsIgnoreCase("mlb")) {  
            MlbSportInfo mlbSportInfo = new MlbSportInfo();  
            mlbSportInfo.setHits(Integer.parseInt(sportInfoArray[1]));  
            mlbSportInfo.setRuns(Integer.parseInt(sportInfoArray[2]));  
            return mlbSportInfo;  
        }  
    }  
}
```

OCP 6/11 – i.e.

- Customer feedback:
Why users see errors? I pay you for working app, not for errors! Make it work right till evening!
- After debugging you found silly NPE mistake

OCP 7/11 – i.e.

```
public class SportInfoParser {
    public SportInfo parseSportInfo(String[] sportInfoArray) {
        if ("nba".equals(sportInfoArray[0])) {
            NbaSportInfo nbaSportInfo = new NbaSportInfo();
            Map<Long, Integer> scores = new HashMap<Long, Integer>();
            scores.put(Long.parseLong(sportInfoArray[12]), Integer.parseInt(sportInfoArray[13]));
            NbaSportInfo.setScores(scores);
            return nbaSportInfo;
        } else if ("nhl".equals(sportInfoArray[0])) {
            NhlSportInfo nhlSportInfo = new NhlSportInfo();
            nhlSportInfo.setSlapShotCount(1);
            return nhlSportInfo;
        } else if (sportInfoArray[0].equalsIgnoreCase("mlb")) { // NPE occurred
            MlbSportInfo mlbSportInfo = new MlbSportInfo();
            mlbSportInfo.setHits(Integer.parseInt(sportInfoArray[1]));
            mlbSportInfo.setRuns(Integer.parseInt(sportInfoArray[2]));
            return mlbSportInfo;
        }
    }
}
```

OCP 8/11 – i.e.

- You did small change, but whole class became unstable
- OCP is violated - we modified existing code
- How to make code open for extension?
 - Use inheritance
 - Use delegation/composition

OCP 9/11 – i.e.

```
public class SportInfoParserDelegatingImpl implements SportInfoParser {
    private Map<String, SportInfoParser> parsers;

    public SportInfoParserDelegatingImpl (Map<String, SportInfoParser> parsers) {
        this.parsers = parsers;
    }

    public SportInfo parseSportInfo(String[] sportInfoArray) {
        SportInfoParser parser = parsers.get(sportInfoArray[0]);
        if (parser == null) {
            return null;
        }
        return parser.parseSportInfo(sportInfoArray);
    }
}

public class SportInfoParserMbaImpl implements SportInfoParser { ... }
public class SportInfoParserNhlImpl implements SportInfoParser { ... }
public class SportInfoParserMlbImpl implements SportInfoParser { ... }
```

OCP 10/11 – i.e.

- You fixed bug and did refactoring to follow OCP
- Customer feedback:
Great, now it works well! Lets add support for WNBA!

OCP 11/11 – i.e.

```
public class SportInfoParserDelegatingImpl implements SportInfoParser {// No changes required!
    private Map<String, SportInfoParser> parsers;

    public SportInfoParserDelegatingImpl (Map<String, SportInfoParser> parsers) {
        this.parsers = parsers;
    }

    public SportInfo parseSportInfo(String[] sportInfoArray) {
        SportInfoParser parser = parsers.get(sportInfoArray[0]);
        if (parser == null) {
            return null;
        }
        return parser.parseSportInfo(sportInfoArray);
    }
}

public class SportInfoParserMbaImpl implements SportInfoParser { ... }
public class SportInfoParserNhlImpl implements SportInfoParser { ... }
public class SportInfoParserMlbImpl implements SportInfoParser { ... }
public class SportInfoParserWnbaImpl implements SportInfoParser { ... }
```


SOLID

SINGLE RESPONSIBILITY PRINCIPLE

SRP 1/8 - idea

- Single class should have a single responsibility
- Class should have only one reason to change
- Friendly principles:
 - DRY – less responsibility – more descriptive class and method names, right location
 - OCP – less responsibility – less reason for modification

SRP 2/8 – ex. 1

```
public class CurrencyConverter {  
    public BigDecimal convert(Currency from, Currency to, BigDecimal amount) {  
        // 1. asks some online service to convert currency  
        // 2. parses the answer and returns results  
    }  
  
    public BigDecimal getInflationIndex(Currency currency, Date from, Date to) { // 3  
        // 4. ask some online service to get data about currency inflation  
        // 5. parses the answer and returns results  
    }  
}
```

- 1, 2, 4, 5: What if online service changes?
- 3: Why getInflationIndex located in CurrencyConverter?!
- Principles violated: SRP, DRY

SRP 3/8 – ex. 1

```
public class CurrencyConverter {  
    public BigDecimal convert(Currency from, Currency to, BigDecimal amount) {  
        // asks some online service to convert currency  
        // parses the answer and returns results  
    }  
}  
  
public class InflationIndexCounter {  
    public BigDecimal getInflationIndex(Currency currency, Date from, Date to) {  
        // ask some online service to get data about currency inflation  
        // parses the answer and returns results  
    }  
}
```

- Now each class have only single responsibility
- Will change only CurrencyConverter if currency conversion logic changes
- Methods placed at right location
- Names are clear

SRP 4/8 – ex. 2

```
public class UserAuthenticator {  
    public boolean authenticate(String username, String password) {  
        User user = getUser(username);  
        return user.getPassword().equals(password);  
    }  
  
    private User getUser(String username) { // 1  
        // ...  
        st.executeQuery("select user.name, user.password from user where id=?"); // 2  
        // ...  
        return user;  
    }  
}
```

- 1: Why getUser located in UserAuthenticator?
- 2: What if jdbc will be changed to ORM or user might come from different sources?
- Principles violation: SRP, DRY, OCP

SRP 5/8 – ex. 2

```
public class UserAuthenticator {  
    private UserDetailsService userDetailsService;  
  
    public UserAuthenticator(UserDetailsService service) {  
        userDetailsService = service;  
    }  
  
    public boolean authenticate(String username, String password) {  
        User user = userDetailsService.getUser(username);  
        return user.getPassword().equals(password);  
    }  
}
```

- Now UserAuthenticator has single responsibility
- UserAuthenticator doesn't depend on specific UserDetailsService implementation

SRP 6/8 – ex. 3

```
public class DocumentServiceImpl {  
    Document getCachedDocumentUuid(String documentUuid) { ... };  
  
    SignedDocument signDocument(Document document, Signature signature) { ... };  
  
    boolean isDocumentSignatureValid(SignedDocument signedDocument) { ... };  
}
```

- Task 1: need new implementation of DocumentService which uses ehcache as caching mechanism

SRP 7/8 – ex. 3

```
public class DocumentServiceEhCacheImpl extends DocumentServiceImpl {  
    @Override  
    public Document getCachedDocumentUuid(String documentUuid) {  
        // some impl  
    }  
}
```

- Task 2: now we need new impl which the same as base but uses different way of verifying document signature

SRP 8/8 – ex. 3

```
public class DocumentServiceGost1024SignImpl extends DocumentServiceImpl {  
    @Override  
    public SignedDocument signDocument(Document document, Signature signature) {  
        // some impl  
    }  
  
    @Override  
    public boolean isDocumentSignatureValid(SignedDocument signedDocument) {  
        // some impl  
    }  
}
```

- Task 3: ok, now we need impl which uses ehcache as caching mechanism and new signature verification mechanism

SOLID

INTERFACE SEGREGATION PRINCIPLE

ISP 1/4 - idea

- Client code shouldn't be obligated to depend on interfaces it doesn't use
- It's better if class implements many small interfaces rather than one big/fat/polluted interface
- Initially hard to distinguish with SRP, but there is a difference
- Friendly principles
 - DRY – if ISP is violated, it might lead to code/logic duplicate

ISP 2/4 – i.e.

```
public interface SpamDetector {  
    boolean isLooksLikeSpam(QuickMessage quickMessage); // 2  
}
```

```
public interface QuickMessage {  
    Author getAuthor();  
    String getMessageText();  
}
```

- 2: SpamDetector depends on QuickMessage interface, but there is no need to access author, why we need it here?!
- Q: What will happen if you need to detect spam in file or, lets say, email?

ISP 3/4 – i.e.

```
public interface SpamDetector {  
    boolean isLooksLikeSpam(QuickMessage quickMessage);  
}
```

```
public interface QuickMessage {  
    Author getAuthor();  
    String getMessageText();  
}
```

```
// ???  
public interface TextFileContent {  
    String getTextContent();  
}
```

```
// ???  
public interface EmailMessage { // i.e. Legacy DTO you can't change  
    Author getAuthor();  
    String getSubject();  
    String getBody();  
}
```

ISP 4/4 – i.e.

```
public interface SpamDetector { // It's very generic now!
    boolean isLooksLikeSpam(HasText hasText);
}

public interface QuickMessage extends HasText { ... }

public interface TextFileContent extends HasText { ... }

public class LegacyEmailMessageHasTextAdapter implements HasText {
    public LegacyEmailMessageHasTextAdapter(EmailMessage emailMessage) {
        this.emailMessage = emailMessage;
    }

    @Override
    public String getText() {
        return emailMessage.getSubject() + emailMessage.getBody();
    }
}
```

SOLID

**DEPENDENCY INVERSION
PRINCIPLE**

DI 1/4 - idea

- **Dependency Inversion**
 - Code to abstraction, not to implementation
 - Objects that use other objects, shouldn't create latter ones
- **Profits**
 - Unit testing is possible
 - Easy change of concrete implementation
 - Each “module” depends on minimum and well-defined outer abstractions

DI 2/4 – i.e.

```
public interface HtmlParser {
    HtmlDocument parseUrl(String url);
}

public class DomBasedHtmlParser implements HtmlParser {
    public HtmlDocument parseUrl(String url) {
        // do the best
    }
}

public class Crawler {
    public void saveHtmlDocument() {
        DomBasedHtmlParser parser = new DomBasedHtmlParser();
        HtmlDocument document = parser.parseUrl("http://.....");
        // do crawl
    }
}
```

- Unit testing of Crawler is not possible, you can't mock parser
- Crawler is not flexible, you can't just use other implementation

DI 3/4 – i.e.

```
public class Crawler {  
    private DomBasedHtmlParser domBasedHtmlParser;  
  
    public void Crawler(DomBasedHtmlParser domBasedHtmlParser) {  
        this.domBasedHtmlParser = domBasedHtmlParser;  
    }  
  
    public void saveHtmlDocument() {  
        HtmlDocument document = domBasedHtmlParser.parseUri("http://.....");  
        // do crawl  
    }  
}
```

- Ok, now we can inject it and do unit testing
- But still locked on specific implementation

DI 4/4 – i.e.

```
public class Crawler {  
    private HtmlParser htmlParser;  
  
    public void Crawler(HtmlParser htmlParser) {  
        this.htmlParser = htmlParser;  
    }  
  
    public void saveHtmlDocument() {  
        HtmlDocument document = htmlParser.parseUrl("http://.....");  
        // do crawl  
    }  
}
```

- Great! Now Crawler don't depend on outer world specific and it's not really interested in implementation details

SOLID

LISKOV SUBSTITUTION PRINCIPLE

LSP 1/8 - idea

- **Liskov Substitution Principle**
 - Says: derived types must be completely substitutable for their base types
 - Helps to use inheritance correctly
 - Helps to abstract from specific implementation
- Friendly principles
 - DI
 - SRP, ISP

LSP 2/8 - idea

- Major examples of LSP violation
 - Sub-class implements only some methods, other look redundant and ... weird
 - Some methods behavior violates contract
 - equals() method symmetry requirement is violated
 - Subclass throws exception which are not declared by parent class/interface (java prevents from introducing checked exceptions)

LSP 3/8 – redundant methods

```
class Bird extends Animal {  
    @Override  
    public void walk() { ... }  
  
    @Override  
    public void makeOffspring() { ... };  
  
    public void fly() {...} // will look weird for Emu  
}  
  
class Emu extends Bird {  
    public void makeOffspring() {...}  
}
```

LSP 4/8 – redundant methods

```
class Bird extends Animal {  
    @Override  
    public void walk() { ... }  
  
    @Override  
    public void makeOffspring() { ... };  
}  
  
class FlyingBird extends Bird {  
    public void fly() {...}  
}  
  
class Emu extends Bird {  
    public void makeOffspring() {...}  
}
```


LSP 5/8 – contract violation

```
interface ArraySorter {  
    Object[] sort(Object[] args);  
}  
  
class DefaultArraySorter implements ArraySorter {  
    public Object[] sort(Object[] array) {  
        Object[] result = array.clone();  
        // ...  
    }  
}  
  
class QuickArraySorter implements ArraySorter {  
    public Object[] sort(Object[] array){  
        Object[] result = array;  
        // original array changed! Error! Negative side-effect!  
    }  
}
```

LSP 6/8 – buggy equals()

```
public class Point {  
    private int x;  
    private int y;  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Point)) return false;  
        Point point = (Point) o;  
        if (x != point.x) return false;  
        if (y != point.y) return false;  
        return true;  
    }  
}
```

LSP 7/8 – buggy equals()

```
public class ColoredPoint extends Point {  
    private int color;  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof ColoredPoint)) return false;  
        if (!super.equals(o)) return false;  
        ColoredPoint that = (ColoredPoint) o;  
        if (color != that.color) return false;  
        return true;  
    }  
}
```

```
Point point = new Point(1, 1);  
ColoredPoint coloredPoint = new ColoredPoint(1, 1, 1);
```

```
point.equals(coloredPoint) == true  
coloredPoint.equals(point) == false
```

LSP 8/8 – buggy equals()

```
public class ColoredPoint {  
    private Point point; // Use delegation instead of inheritance!  
    private int color;  
}
```

```
Point point = new Point(1, 1);  
ColoredPoint coloredPoint = new ColoredPoint(1, 1, 1);
```

```
point.equals(coloredPoint) == false  
coloredPoint.equals(point) == false
```

THAT'S IT, THANK YOU!

**p.s. Please, don't
write BS which makes
my eyes bleeding**