Java Coding Standards

This document describes the java code standards applied across our products. Main purpose is to make sure all members produce code with same guaranteed quality, testability and readability. This standards might change time to time. If change happens then all new code is written according the new version and old code is updated with update on touch style.

There are 3 areas everybody need to care about during writing any java code:

Note that each item of the mentioned above is EQUAL in importance for us. **Code** is the what makes products alive. Working and easy to read code is a elementar y must. For this reason it can make the impression that this is the most important point and at the beginning it really is. Over the time the other two pieces become more and more important. Code is already released and everyone forgot about it. Then we do not want to have surprises one year later. And that's where the Unit tests becomes handy. **Unit tests** holds our product consistent. Features need bug fixes, optimizations and extensions. And well defined unit tests prevent us from spending time on reoccurring bugs or introducing new bugs. That's why they are important. **Comments** are important to understand the code. Anyone who is creating method, property, enumeration must be able to explain what he/she is doing by words. Otherwise there is no sense to do this. In addition code might be reviewed by project owners and other third party companies. Not completed javadoc (even the trivial one) might be considered as a partial reason for lost in sales. Therefore it is important to keep everything documented. **In summary experience says that we spend more time doing unit tests and comments than the actual code.** Now let's go into detail trough each of these pieces.

Code

As it was stated before having a working, stable and clean code is the way how products can survive. If this is not the case, then the following errors would happen:

1. Compilation errors – easy to fix
2. Exception errors – can have 2 causes (wrong code / system failures)
3. Errors when something is wrongly calculated, persisted and then possibly used in future – the most terrible ones

There is no problem to deal with errors 1. and 2, everybody can fix them. Problem is error of type 3. and this becomes more and more important over the time when product allocates new features which interacts together. Therefore we define standards which are primary focused on minimizing the possibility of error 3. Performance, memory and other issues are secondary and enough level is achieved simply by creating the right algorithms on the high level and tuning up functions without touching the interface. The rest of this chapter will go into details about our code.

Package naming convention

Package naming format is **prefix.project[.section[.type[...]]]**. Please see the following examples:

- com.fusionsystems.spher.utils
- com.fusionsystems.spher.reporter.dto
- com.fusionsystems.spher.reporter.service
- com.fusionsystems.spher.reporter.dao
- com.fusionsystems.spher.engine.dto
- com.fusionsystems.spher.engine.detection.service
- com.fusionsystems.utils

Before creating new package, please look whether your class fits into the any of the existing packages. If no, then package can be created. Convention is to use same package names for folders with unit tests. Please note that packages are time to time refactored to fit the project needs and to keep the original naming convention.

Data holders

Data holders are java objects which are used to encapsulate primitive data in the suitable format. For purpose of this standards we will define these types of data holders:

1. DTO objects
2. Model objects

DTO objects are building blocks of the business logic. Intention for them is to be used in the wide scope. This means across all layers and possibly applications. Therefore there are strict rules for them. On the other side there are **model objects**. They are used in a narrow scope, typically just to connect frameworks with our application. Therefore they don't have specific rules. Typical examples of model objects are hibernate entities and objects for binding values from the web forms. That's it. Now let's go through details.

(Note there are various definitions of data holders over the internet and in this project we do not care about them)

DTO objects

DTO objects are building blocks for our products. Thy guarantee our products to be consistent and stable. Therefore they have the most amount of rules to be satisfied. In short they must be plain, truly immutable and constructed in the unified way with immediate consistent validation. This implies the following set of rules, which is **MANDATORY** for **ALL** DTO objects:

- Simple class name
  No common prefix or suffix is allowed (examples of names: AccountStatus, SecuritAuditRecord, UserPreference, GraphGroup...)
- No inheritance
  - key words **extends** and **implements** are prohibited in the class definition
  - use composite pattern if needed - composite means that one DTO object can contain another DTO object (see the example below)

- No annotations
- Builder pattern
- Truly immutable
    - Defense copies of dates
    - Defense copies of collections
    - Never use object which can be modified inside the DTO object
    - Only "get" type method are allowed in the main object
    - Implementation of "get" type method MUST make sure the returned object is unmodifiable or it's modification doesn't affect the original object (e.g. unmodifiable collections or defense copy of date)
- HashCode + Equals comparing all properties and are symmetric, reflexive and transitive
- ToString method is overridden

There is the **STRICT** pattern which **must** be followed for all DTO objects (demonstrated with examples below)

1. Builder object
    a. properties
    b. setters - setters starting with set* or add*, always return instance to that builder for chaining
    c. build method
2. Properties
3. Constructor accepting builder
4. guardInvariants method - validates that object is consistent and throws exception if not
5. getters
6. hashCode - must be well defined (same for the 2 equal objects)
7. equals - must be reflexive, symmetric and transitive and compare all fields deeply
8. toString - must be defined, dump all fields except protected information like password or huge lists (in this situation custom implementation can be made

For better illustration, please see the following examples:

Minimal domain object

```
/**
 * This is the minimal DTO object.
 * There are no properties within this object, but contains all other mandatory parts in the required order.
 *
 * @author radek.hecl
 *
 */
public class MinimalObject {
```

```
/**
 * Builder object.
 */
public static class Builder {

    /**
     * Builds the result object.
     *
     * @return created object
     */
    public MinimalObject build() {
        return new MinimalObject(this);
    }

}

/**
 * Creates new instance.
 *
 * @param builder builder object
 */
public MinimalObject(Builder builder) {
    guardInvariants();
}

/**
 * Guards this object to be consistent. Throws exception if this is not the case.
 */
private void guardInvariants() {
}

@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this);
}

@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj);
}

@Override
```

```
    public String toString() {
```

```
        return ToStringBuilder.reflectionToString(this);
    }
}
```

Properties handling example

```
/**
 * Object which defines the account status of the individual.
 * Please see how the properties are handled and validated.
 *
 * @author radek.hecl
 *
 */
public class AccountStatus {

    /**
     * Builder object.
     */
    public static class Builder {

        /**
         * Total amount of money on the account.
         */
        private Double totalMoney;

        /**
         * Account holder's date of birth.
         */
        private Date dateOfBirth;

        /**
         * Account holder's names.
         */
        private List<String> names = new ArrayList<>();

        /**
         * Sets total amount of money on the account.
         *
         * @param totalMoney total amount of money on the account
         * @return this instance
```

```
     */
    public Builder setTotalMoney(double totalMoney) {
        this.totalMoney = totalMoney;
        return this;
    }

    /**
     * Sets account holder's date of birth.
     *
     * @param dateOfBirth date of birth
     * @return this instance
     */
    public Builder setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = DomainUtils.copyDate(dateOfBirth);
        return this;
    }

    /**
     * Sets account holder's names.
     *
     * @param names sorted names where first is the given name, then middle names and last family name(s)
     * @return this instance
     */
    public Builder setNames(List<String> names) {
        this.names = DomainUtils.softCopyList(names);
        return this;
    }

    /**
     * Adds another name to the account holder's names.
     *
     * @param name name to be added
     * @return this instance
     */
    public Builder addName(String name) {
        names.add(name);
        return this;
    }

    /**
     * Builds the result object.
     *
```

```
     * @return created object
     */
    public AccountStatus build() {
        return new AccountStatus(this);
    }
}

/**
 * Total amount of money on the account.
 */
private Double totalMoney;

/**
 * Account holder's date of birth.
 */
private Date dateOfBirth;

/**
 * Account holder's names.
 */
private List<String> names = new ArrayList<>();

/**
 * Creates new instance.
 *
 * @param builder builder object
 */
public AccountStatus(Builder builder) {
    totalMoney = builder.totalMoney;
    dateOfBirth = DomainUtils.copyDate(builder.dateOfBirth);
    names = Collections.unmodifiableList(DomainUtils.softCopyList(builder.names));
    guardInvariants();
}

/**
 * Guards this object to be consistent. Throws exception if this is not the case.
 */
private void guardInvariants() {
    ValidationUtils.guardNotNull(totalMoney, "totalMoney cannot be null");
    ValidationUtils.guardBoundedDouble(totalMoney, "totalMoney must be bounded");
    ValidationUtils.guardNotNull(dateOfBirth, "dateOfBirth cannot be null");
    ValidationUtils.guardNotEmptyStringInCollection(names, "names cannot have empty string");
```

```
        ValidationUtils.guardPositiveInt(names.size(), "at least 1 name must be defined");
    }

    /**
     * Returns the total mount of money on the account.
     *
     * @return total amount of money on the account
     */
    public double getTotalMoney() {
        return totalMoney;
    }

    /**
     * Returns account holder's date of birth.
     *
     * @return account holder's date of birth
     */
    public Date getDateOfBirth() {
        return DomainUtils.copyDate(dateOfBirth);
    }

    /**
     * Returns the account holder's names. First item is first names, then middle names and then family name.
     *
     * @return account holder's names
     */
    public List<String> getNames() {
        return names;
    }

    @Override
    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this);
    }

    @Override
    public boolean equals(Object obj) {
        return EqualsBuilder.reflectionEquals(this, obj);
    }

    @Override
    public String toString() {
```

```
        return ToStringBuilder.reflectionToString(this);
    }
}
```

Builder usage

```
return new AccountStatus.Builder().
        setTotalMoney(totalMoney).
        setDateOfBirth(dateOfBirth).
        addName(firstName).
        addName(lastName).
        build();
```

Composite object

```
/**
 * This is the user profile object.
 * User profile is composition of user preference information and account status.
 *
 * @author radek.hecl
 *
 */
public class UserProfile {

    /**
     * Builder object.
     */
    public static class Builder {
        ...
    }

    /**
     * Status of the account.
     */
    private AccountStatus status;

    /**
```

```
 * User preference.
 */
private UserPreference preference;

/**
 * Creates new instance.
 *
 * @param builder builder object
 */
public MinimalObject(Builder builder) {
    status = builder.status;          // can do this, because AccountStatus is also DTO object
    preference = builder.preference;
    guardInvariants();
}

/**
 * Guards this object to be consistent. Throws exception if this is not the case.
 */
private void guardInvariants() {
    ValidationUtils.guardNotNull(status, "status cannot be null");
    ValidationUtils.guardNotNull(preference, "preference cannot be null");
}

/**
 * Returns the status of the account.
 *
 * @return status of the account
 */
public AccountStatus getStatus() {
    return status;
}

/**
 * Returns the total mount of money on the account.
 *
 * @return total amount of money on the account
 */
public double getTotalMoney() {
    // proxy to the composite object
    return status.getTotalMoney();
}
```

```
    ...
}
```

This was about the strict part for DTO objects. Now let's see what are the allowed options:

- Static factory methods

Static factory methods are here to create a convenient method for construction. They are optional. If they exists then the position is right after the toString method and all of them are named with pattern create*. In the simplest implementation they call the builder object. If this is necessary for the speed, then private constructor with no arguments can be created for the purpose of factory method. In this case factory method is responsible to call the guardInvariants method before returning the object. For details please look to the examples below.

Static factory calling builder

```
...

@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}

/**
 * Creates account status for people with first and last name.
 *
 * @param totalMoney total amount of money on the account
 * @param dateOfBirth date of birth
 * @param firstName first name
 * @param lastName last name
 */
public static AccountStatus create(double totalMoney, Date dateOfBirth, String firstName, String lastName) {
    return new AccountStatus.Builder().
            setTotalMoney(totalMoney).
            setDateOfBirth(dateOfBirth).
            addName(firstName).
            addName(lastName).
            build();
}
```

Static factory calling private constructor

```
... private constructor is right after the property definitions

/**
 * Creates new instance.
 */
private AccountStatus() {
}

.... private constructor is right before constructor accepting builder

... factory method is right after toString method

/**
 * Creates account status for people with first and last name.
 *
 * @param totalMoney total amount of money on the account
 * @param dateOfBirth date of birth
 * @param firstName first name
 * @param lastName last name
 */
public static AccountStatus create(double totalMoney, Date dateOfBirth, String firstName, String lastName) {
    AccountStatus res = new AccountStatus();
    res.totalMoney = totalMoney;
    res.dateOfBirth = DomainUtils.copyDate(dateOfBirth);
    res.names = Collections.unmodifiableList(Arrays.asList(firstName, lastName));
    res.guardInvariants();
    return res;
}
```

Model objects

Model objects are here to be able connect various frameworks together. They cannot participate in the business logic and they are not intended to be reused. All these implies 3 rules:

1. No structure rules
2. Naming

3. Narrow scope

No structure rules means there is not defined way how to write model objects. Everything is driven by the framework or usage. Example is hibernate entity. This is typically simple class with getters/setters and can extend the other classes. This implies that hibernate entity is not thread safe, collection safe and in case of inheritance is impossible to write well defined hashCode and equals methods which compares all fields. Really the opposite of dto object. Another examples are JAXB classes and web form model for spring framework. Each of them has different needs and therefore different structure. Everything is allowed as far as **narrow scope** rule is followed.

Naming rule defines what names used for the model objects. For the name use pattern *Suffix, where suffix says something about the model object. For example suffix *Entity is used for hibernate entities, suffix *RequestModel is used for incomming http requests and suffix *FilterModel is used for objects which are creating search filters. If there is unclear what suffix to use, then use just *Model.

Narrow scope rule means that model objects can be used only in the smallest possible scope. This is typically within the implementation of the particular interface or a unit which is called by framework. Model object **MUST** be translated to the DTO object to shift data between layers. **Services which accepts or returns model objects are not allowed** (this is discussed later).

Enumerations

Enumerations are here to enumerate options, statues and others. Name rule is to match the pattern *Type, *Style, *Status... NEVER use *Enum pattern. Rule for enumeration definition is that it has to be **simple**, nothing else. This means there is nothing else except enumerated values as it is in the example below. That's all.

Enumeration example

```
/**
 * Enumerated the states of the graph group.
 *
 * @author radek.hecl
 *
 */
public enum GraphGroupState {

    /**
     * State when graph group is creating.
     * In this state new graphs are added into group.
     */
    CREATING,

    /**
     * State when graph group is completed.
     */
    COMPLETED
}
```

Interfaces

Interfaces are serving us 3 purposes:

1. Functionality definition
2. Interceptors
3. Unit testing

Functionality definition is the primary purpose of interface in Java. Client / user then operates only with interface and doesn't care about the actual implementation. Typical example is interface for sending emails. Implementation for development environment stores them as files in the predefined directory. Implementation in production sends emails via SMTP service. Higher level layers (= clients which are sending emails) just calls the method for message send and do not care about the actual implementation. **Interceptors** allows frameworks to invoke extra code before / after the method invocation. Typical example is database transactions handled by Spring framework. **Unit testing** allows us to test the whole business logic just by unit tests. Interface is mocked by testing framework (jMock) and logic is verified only for single element. This is extremely important, because unit tests provides fastest way how to verify bugs. Therefore all service classes must be tested in this way.

Now about the names used for the interfaces. The whole rule is based on the fact that interface provides the functionality for specific data structures. Therefore names are as pattern [data-type][functionality], where data-type can be for example Dates, Incident, Log and functionality can be for example *Service, *Tracker, *Dao, *Provider, *Converter. Examples of the interface names are below.

- CriteriumStringMapConverter
- DatesProvider
- IncidentDao
- LogService

Once interface name is defined, then the method comes in. Method has also set of rules for **declaration, naming** and **parameters**, so let's go through of them.

Declaration rule says that every method in the interface must be explicitly marked as public and this is required across all interfaces. **Naming** rule standardized names which are used for the frequently method types. Frequently method types are the ones which works with persisted objects. Please see the standard name patterns and usage:

- select*
  Select name pattern is used for retrieving the selected entities. 2 subsequent calls with the same parameters will return the same values if the persisted objects haven't been changed.
- count*
  Count name pattern is used for counting the number of persisted objects which satisfies the criteria.
- insert*
  Insert name pattern is used for persisting new object.
- delete*
  Delete name pattern is used for deleting the existing object.
- update*
  Update name pattern is used for updating the existing object.
- generate*
  Generate name pattern is used to generate new object. Typically unique codes. 2 subsequent calls are expected to return different result.
- create*
  Create pattern is used for creating new objects which are not supposed to be persisted. For example creating new stream.

Parameters rule defines what can be used as an input or output. The first **core** rule is **NEVER accept or return model object!!!** Example from practice is that there is no single interface which would accept / return the JPA entity object. For input primitive types and DTO objects are allowed. Collections of primitive types or DTO objects are also allowed if interface is designed in the way that these are non-modifiable (implementation can't modify them - this is discussed more in detail later). Allowed outputs are primitives, DTO objects and collection of primitive types and DTO objects. If return is collection, then expected is collection without read limit. Another possible pseudo output is exception. If all implementations are expected to throw an run time exception on particular situation, then it must be explicitly specified in the documentation. Please look to the examples below.

Interface example

```
/**
 * This interface defines practice management functionality.
 *
 * @author radek.hecl
 *
 */
public interface PracticeService {

    /**
     * Inserts new practice.
     *
     * @param practice practice to insert
     */
    public void insertPractice(PracticeInfo practice);

    /**
     * Selects the specified practice.
     *
     * @param npi npi of the practice to select
     * @return selected practice
     * @throws IllegalArgumentException if npi doesn't match any existing practice
     */
    public PracticeInfo selectPractice(String npi);

    /**
     * Selects list with practices.
     *
     * @param filter filter object
     * @return selected records
     */
    public List<PracticeInfo> selectPractices(Filter<PracticeFilterColumns> filter);

    /**
     * Counts how many practices satisfies the criteria.
     *
     * @param criteria criteria which need to be satisfied
     * @return number of practices codes which satisfies the criteria
     */
    public long countPractices(Set<Criterium<PracticeFilterColumns>> criteria);
```

```
/**
 * Selects the practices.
 *
 * @param filter filter object
 * @return selected records
 */
public Records<PracticeInfo> selectPracticeRecords(Filter<PracticeFilterColumns> filter);

/**
 * Updates practice information.
 *
 * @param npi npi of the original practice
 * @param newNpi new NPI number of the practice
 * @param newName new name of the practice
 * @param newIncidentPartnerCode new code of the partner who is responsible for incident resolution
 */
public void updatePractice(String npi, String newNpi, String newName, String newIncidentPartnerCode);

/**
 * Deletes the specified practice.
 * @param npi npi number of the practice
 * @throws IllegalArgumentException if npi doesn't match any existing practice
 */
public void deletePractice(String npi);

/**
 * Generates new actor code. This is unique code which applies for performers and targets.
 *
 * @return generated code
 */
```

```
    public String generateActorCode();

}
```

Functional objects

Functional objects are classes which holds the code for performing actions and their instances are created during run time. In majority of cases this is connected with implementation of interface and in few cases these objects created without interface. This chapter covers both situations. The difference for functional objects is also whether they are considered as **reusable** (e.g. as a library) or as **project** (e.g. controller for user details) code. **Reusable functional objects** have more restrictions and are intended to be used across projects. **Project functional objects** has less restrictions, are faster to be written and are tided to the particular project. Now let's look to the common requirements and then for each of them in detail.

Common requirements:

- No setters
- Override toString - in most of cases ToStringBuilder.reflectionToString(this)
- Thread safe
- Use override annotation for overridden / interface methods
- Method hashCode and equals are default

Reusable functional objects

Reusable functional objects are intended to be used across the projects. Therefore we take an effort to standardize them, even if the cost is to write a little bit more code. Please note that there must be a reason to make an functional object reusable. Typically this happens when exactly same class is used in at least 3 projects. As they are part of the library, they need to be framework independent. In fact they are functional equivalent of **DTO objects** with almost the same structure (builder, properties, constructor(s), guardInvariants, functional methods, toString, optional static methods). The differences in structure are following:

- No hashCode and equals methods
- Functional methods are used instead of getters
- Override annotations on methods where applicable
- Properties with interfaces to other functional objects are allowed

It is worth to say that in reusable functional objects there are **not allowed framework annotations** (e.g. Autowired or PostConstruct). If these objects needs to be created from framework context, then it must be done in the project which is using the library (e.g. by xml configuration). Please see examples below.

Basic reusable functional object

```
/**
 * Graph service which uses data access layer for accessing the data.
 *
 * @author radek.hecl
 *
 */
public class DaoGraphService implements GraphService {

    /**
     * Builder object.
     */
    public static class Builder {

        /**
         * Dao layer object.
         */
        private GraphDao dao;

        /**
         * Sets the dao layer object.
         *
         * @param dao dao layer object
         * @return this instance
         */
        public Builder setDao(GraphDao dao) {
            this.dao = dao;
            return this;
        }

        /**
         * Builds the result object.
         *
         * @return created object
         */
        public GraphServiceBean build() {
            return new GraphServiceBean(this);
        }
    }

    /**
```

```
 * Dao layer object.
 */
private GraphDao dao;

/**
 * Creates new instance.
 * @param builder builder object
 */
public GraphServiceBean(Builder builder) {
    dao = builder.dao;
    guardInvariants();
}

/**
 * Guards this object to be consistent. Throws exception if not.
 */
private void guardInvariants() {
    ValidationUtils.guardNotNull(dao, "dao cannot be null");
}

@Override
public void insertGraphGroup(GraphGroup group) {
    ...
}

... helper methods are after the main ones...

/**
 * Helper method.
 *
 * @param arg argument
 * @return whether argument satisfies special criteria or not
 */
private boolean superHelperMethod(int arg) {
    ...
}

...

@Override
public String toString() {
```

```
        return ToStringBuilder.reflectionToString(this);
    }
}
```

Project functional objects

Project functional object are intended to be used only in project. Therefore there is no need to keep them framework independent. And it is worth to get rid of builder as this is overhead for the code which is changing often. Therefore builder object is replaced with private constructor without parameters, static factory methods are used for unit tests and framework annotations are allowed. This for example means that properties can be annotated and autowired and guardInvariants method is invoked by framework right after the construction. One more relaxation is for the naming convention. If there is only one implementation and there is no name to be put in, then name is constructed as interface name + Bean. More in the example below.

```
/**
 * Bean implementation of the user service.
 *
 * @author radek.hecl
 *
 */
@Service
public class UserServiceBean implements UserService {

    /**
     * Generator of unique codes.
     */
    @Autowired
    private UniqueCodeGenerator codeGenerator;

    /**
     * Data access object for user manipulation.
     */
    @Autowired
    private UserDao userDao;

    /**
     * Constructor to allow frameworks create new instances.
     */
```

```
private UserServiceBean() {
}

/**
 * Guards this object to be consistent. Throws exception if this is not the case.
 */
@PostConstruct
private void guardInvariants() {
    ValidationUtils.guardNotNull(codeGenerator, "codeGenerator cannot be null");
    ValidationUtils.guardNotNull(userDao, "userDao cannot be null");
}

@Override
public String generateUserCode() {
    ....
}

@Override
@Transactional(readOnly = true)
public long countUsers(Set<Criterium<UserFilterColumns>> criteria) {
    ....
}

....

@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}

/**
 * Creates new instance.
 *
 * @param codeGenerator generator for unique codes
 * @param userDao data access object for users
 * @return created instance
public static UserServiceBean create(UniqueCodeGenerator codeGenerator, UserDao userDao) {
    UserServiceBean res = new UserServiceBean();
    res.codeGenerator = codeGenerator;
    res.userDao = userDao;
    res.guardInvatriants();
```

```
            return res;
        }
    }
```

Utility classes

Utility classes are class which contains only the static methods. Rule for them it to make private constructor which prevents construction and all methods static. This will make easy to invoke them from anywhere. Utility classes also cannot have any dependencies. They can just call another utility classes. Please see the example below.

Utility class

```
/**
 * Contains static method which simplifies the manipulation with domain objects.
 *
 * @author radek.hecl
 *
 */
public class DomainUtils {
    /**
     * To prevent construction.
     */
    private DomainUtils() {
    }

    /**
     * Null safe copy date function.
     *
     * @param source source date, can be null
     * @return copy of the source date or null if source is null
     */
    public static Date copyDate(Date source) {
        if (source == null) {
            return null;
        }
        return new Date(source.getTime());
    }

    ... other static methods
}
```

Exception handling

Exception handling is here to make sure all the errors are properly handled. First of all let's look more close on what **exceptional situation** means. This is context dependent. Let's imagine service which is designed to accept only valid xml files. For this service any file which is not valid xml is an exceptional situation, because this

can happen only by mistake. In this case exception should be thrown. On the other let's imagine service which accepts any file, process if this file is a valid xml and returns the processing result. For this service there is no input which would be exceptional as any file can be passed. Exceptional case would be only if disk is broken and file is not readable. Therefore exceptional situation is always context dependent. Typically exceptions are result of system failure (cannot be controlled from code) and bugs in code. This was about **exceptional situation**. Now let's look how to handle them. The simplest way is let run time exception to go up. In most of cases this works well, because frameworks are already prepared to handle them. The other way is to catch and handle. And the last way is to catch, translate and re throw. That's pretty much it. Now let's write what does it mean for the code in more detail. There are the following rules.

- Throw judiciously
- Use java exceptions
- Catch block handles

Throw judiciously is the first rule. In fact except input validation and enumeration protections there is not many cases when exception would be thrown from our code. One of these cases is when interface defines particular exception for specified input parameters. **Use java exceptions** means that we should not create our own exceptions. Standard java already has build in ones. So let's use them and define our own only if there is a specific way to handle them. **Catch block handles** is the simple rule. Every time when there is a catch block, handling must happen. In fact there are 3 ways of handling exceptions - recover, put into result and translate.

Resources

This part shows how to handle resources without mess. Goal is to know about every error (even during close) and give a chance for close to everything what needs it. This http://illegalargumentexception.blogspot.jp/2008/10/java-how-not-to-make-mess-of-stream.html provides pretty good explanation. The sufficient strategy is to know about the first error and give chance to close for everything. Sample pseudo code in Java 1.6 looks like example below. DTO objects and interfaces can expose methods which creates streams. And then usually client is responsible for closing them.

Resource closing

```
OutputStream resource1 = null;
OutputStream resource2 = null;
try {
    resource1 = new FileOutputStream(f1);
    resource2 = new FileOutputStream(f2);

    ...

    resource1.close();
    resource2.close();
} finally {
    IoUtils.closeQuietly(resource1);
    IoUtils.closeQuietly(resource2);
}
```

Note: There was introduced new resource handling in java 1.7. If this provides the same or even better ability to catch any error and automatically closes everything, then we can use it for projects (not libraries) written in java 1.7. Check it and compare.

Unit tests

Unit tests are key tool for the product stability. They provides fastest way to know whether every single class is working as expected. And more they prevent us from doing changes which might break part of code which we would never expect. This is the whole point of having them. This java coding standards are designed to target 100% business logic test coverage. In addition we can also validate 100% of object properties (this is the result of DTO object definition). Now let's go through the details about it.

Test class

For writing test we are using the notation from JUnit 3. Test class must extend TestCase class. Structure within test class goes in the way - properties, default constructor, optional set up method, optional tear down method, test methods, optional private utility methods and finally toString method dumping all the properties. There are 2 more details. First the naming rule for the test methods is to start with test word. Second it is better to not rely on the tear down method. If there is any option, then please make sure set up method makes all the preparation and do not define tear down method. For more details let's look to the example.

```
/**
```

```
 * Test which proves that service for message management is working.
 *
 * @author radek.hecl
 *
 */
public class MessageServiceBeanTest extends TestCase {

    /**
     * Data access layer for messages.
     */
    private MessageDao messageDao;

    /**
     * Tested service.
     */
    private MessageServiceBean service;

    /**
     * Context for mocking.
     */
    private Mockery context;

    /**
     * Creates new instance.
     */
    public MessageServiceBeanTest() {
    }

    @Override
    protected void setUp() throws Exception {
        context = new Mockery();

        messageDao = context.mock(MessageDao.class);

        service = new MessageServiceBean.Builder().
                messageDao(messageDao).
                build();
    }

    /**
     * Tests for for formatting http post messages.
     */
```

```
    public void testFormatHttpPost() {
        ...
    }

    ... other test methods
    ... private utility methods

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
}
```

Business logic tests

Business logic tests are done with jMock framework (equivalents are EasyMock or MockIt). Except the framework knowledge the following rules applies:

- Use full equals
- Use sequences

Use full equals means that when there is an expectation for call to the lower layer, then it must be written with equal expectation (not with any). **Use sequences** means that whenever methods are required to be called in particular sequence, then this must be checked. More you can see in the example below.

Business logic test

```
/**
 * Test which proves message service bean is working.
 *
 * @author radek.hecl
 *
 */
public class MessageServiceBeanTest extends TestCase {

    ... properties

    /**
     * Creates new instance.
     */
    public MessageServiceBeanTest() {
```

```
}

@Override
protected void setUp() throws Exception {
    context = new Mockery();

    messageDao = context.mock(MessageDao.class);
    datesProvider = context.mock(DatesProvider.class);
    uniqueCodeGenerator = context.mock(UniqueCodeGenerator.class);
    emailFormatter = context.mock(Formatter.class, "emailFormatter");
    httpPostFormatter = context.mock(Formatter.class, "httpPostFormatter");
    emailSender = context.mock(Sender.class, "emailSender");
    httpPostSender = context.mock(Sender.class, "httpPostSender");

    service = new MessageServiceBean.Builder().
            messageDao(messageDao).
            datesProvider(datesProvider).
            uniqueCodeGenerator(uniqueCodeGenerator).
            emailFormatter(emailFormatter).
            httpPostFormatter(httpPostFormatter).
            emailSender(emailSender).
            httpPostSender(httpPostSender).
            build();
}

/**
 * Tests for for formatting http post messages.
 */
public void testFormatHttpPost() {
    Map<String, String> inputData = ...;
    FormattedText expected = ...;
    ValueAddedReseller partner = ...;

    context.checking(new Expectations() {
        {
            Map<String, Object> data = new HashMap<>();
            data.put("target.name", "Seaman Partners");
            data.put("target.privateKey", "pk");
            data.put("practice.name", "Seaman Healthy");
            data.put("practice.npi", "0000000001");

            allowing(httpPostFormatter).format(with(equal(MessageType.DETECTED_INCIDENT_NOTIFICATION)),
```

```
with(equal(data)));
                will(returnValue(new FormattedText.Builder().
                        subject("hello").
                        text("world").
                        build()));
            }
        });

        assertEquals(expected, service.formatHttpPost(MessageType.DETECTED_INCIDENT_NOTIFICATION, inputData, partner));

        context.assertIsSatisfied();
    }
```

```
    ...

}
```

Data access layer tests

Data access layer is the testing of the parts where we cannot mock the lower layer by jMock. These are the lowest layers within our code and doesn't contributing to the business logic. Examples are database and payment gateway. Database can have test instance on everyone's local machine and therefore all tests can be written and run together with unit tests. Payment gateway is system which lives outside of our environment and it is not guaranteed that this system is running at all. Therefore we can write unit tests, but these unit tests are usually commented. In both cases tests for data access layer are simple. Just execute set of actions and compare results. There is a single rule - **ALWAYS COMPARE THE FULL RESULT.** For example if function returns the list of objects, then comparing only list sizes is not enough. Comparing only some properties is also not enough. Comparing deeply 2 lists is enough. And this coding standards make it easy to do just by one assert (as all DTO objects have deep equal by definition). See the examples below.

Comparing list of domain objects

```
SecurityAuditLog rec1 = ...;
SecurityAuditLog rec2 = ...;
SecurityAuditLog rec3 = ...;

List<SecurityAuditLog> res = null;
List<SecurityAuditLog> expected = null;
Filter<SecurityAuditFilterColumns> filter = null;

filter = new Filter.Builder<SecurityAuditFilterColumns>().
        addNotEqualCriterium(SecurityAuditFilterColumns.PRACTICE_NPI, "9999999999").
        addDescendantOrder(SecurityAuditFilterColumns.EVENT_TIMESTAMP).
        setFrom(0).
        setLimit(1000).
        build();
res = provider.selectList(filter);
expected = Arrays.asList(rec3, rec2, rec1);
assertEquals(expected, res);
```

Comments

Comments are key for the product long time maintenance. For our standards comments means to write the Java doc. There is general rule for writing comments - **COMMENT EVERYTHING**. This means every class, method, field, enumeration, and others must be commented. Modifiers like private, static, protected... doesn't remove the duty for comment. Even the methods where functionality is obvious must be commented. The only exception is for methods which are marked as @Override and reason is, because the comment is inherited from parent. Now let's go into the details.

Note: If there is any time question similar to "should this XXX be commented", then answer is "yes, it should".

Definition: **Description** is sentence starting with capital letter and finished with period.

Definition: **Annotation description** is sentence starting with lower case letter and finished without period

General comment format is to have part with description, then one free line and finally the part with annotations. In addition expressions {@link #something} is prohibited to use. Now please see the examples of various comments.

Interfaces and classes

Minimal requirements:

- Write **description**.
- @author annotation is used only for people with significant contribution to the design or implementation. Author is always writtent as "first.last" pattern, everything lower case.

Interface comment

```
/**
 * Describes the database layer functionality for manipulation with graphs.
 *
 * @author radek.hecl
 *
 */
public interface GraphDao {
  ...
}
```

Class comment

```
/**
 * This implementation of graph management interface uses database as a storage.
 *
 * @author zakhar.amirov
 *
 */
@Repository
public class DbGraphDao implements GraphDao {
  ...
}
```

Inner class comment

```
/**
 * Describes the core information about detector.
 *
 * @author radek.hecl
 *
 */
public class DetectorInfo {

    /**
     * Builder object.
     */
    public static class Builder {
       ...
    }

    ...
}
```

Enumerations

Minimal requirements:

- **Description** on the enumeration definition
- **Description** on each of the enumerated items

Enumeration comment

```
/**
 * Enumerates the columns for the filtering and ordering graph groups.
 *
 * @author radek.hecl
 *
 */
public enum GraphGroupFilterColumns {

    /**
     * Code of the graph group.
     */
    CODE
}
```

Fields

Minimal requirements:

- **Description** on every field, no matter whether it is private, static, or anything else.
- If there is any special limitations for the field, then write it after the description, possibly with examples

Field comment

```
/**
 * Channel of this upload block.
 */
private Channel channel;
```

Field detailed comment

```
/**
 * Enabled flag of this upload block.
 * Value is true if all the following conditions are satisfied:
 * <ul>
 *   <li>ehr version is enabled</li>
 *   <li>ehr product is enabled</li>
 *   <li>ehr version is enabled</li>
 *   <li>practice is activated</li>
 * </ul>
 * Otherwise false.
 */
private Boolean enabled;
```

Methods

Minimal requirements:

- Every method which doesn't have @Override annotation is commented
- Method has **description**
- Every input parameter has **annotation description**
- Return statement has **annotation description**
- Every checked exception has **annotation description**
- Non checked exceptions have **annotation description** if there is expected behavior when they are supposed to be thrown. If this is not defined, then they have no place to be in the comment
- If there is required more sentences in **annotation description,** then **annotation description** is finished with period and next sentence starts with capital letter and is also finished with period. Then there can be as many sentences as needed

Constructor comment

```
/**
 * Creates new instance.
 *
 * @param builder builder object
 */
public UploadBlock(Builder builder) {
    ...
}
```

Simple getter comment

```
/**
 * Returns channel customization level.
 *
 * @return channel customization level
 */
public UploadBlockCustomizationLevel getChannelCustomizationLevel() {
    return channelCustomizationLevel;
}
```

Complex method comment

```
/**
 * Performs the section task. This method is expected to be overridden by subclasses.
 * This method does not produce any result. Also this class guarantee that only
 * single thread executes this method. Therefore there is no need for additional check and
 * no need to make this method thread safe.
 *
 * @param parameters parameters to specify the behavior. It is up to the implementation to decide how to behave for the given
parameters.
 * @throws Exception in case of problem. Implementation can throw any exception and this will be handled by the framework.
 */
protected abstract void runInternal(Map<String, String> parameters) throws Exception;
```

Unchecked exception comment

```
/**
 * Updates ehr version code for a specified practice.
 *
 * @param npi NPI number of the practice
 * @param ehrVersionCode new value of the ehr version code, can be null if this practice is not associated with any version
 * @throws IllegalArgumentException if npi doesn't match any existing practice or ehrVersionCode doesn't match any existing
version
 */
public void updateEhrVersionCode(String npi, String ehrVersionCode);
```