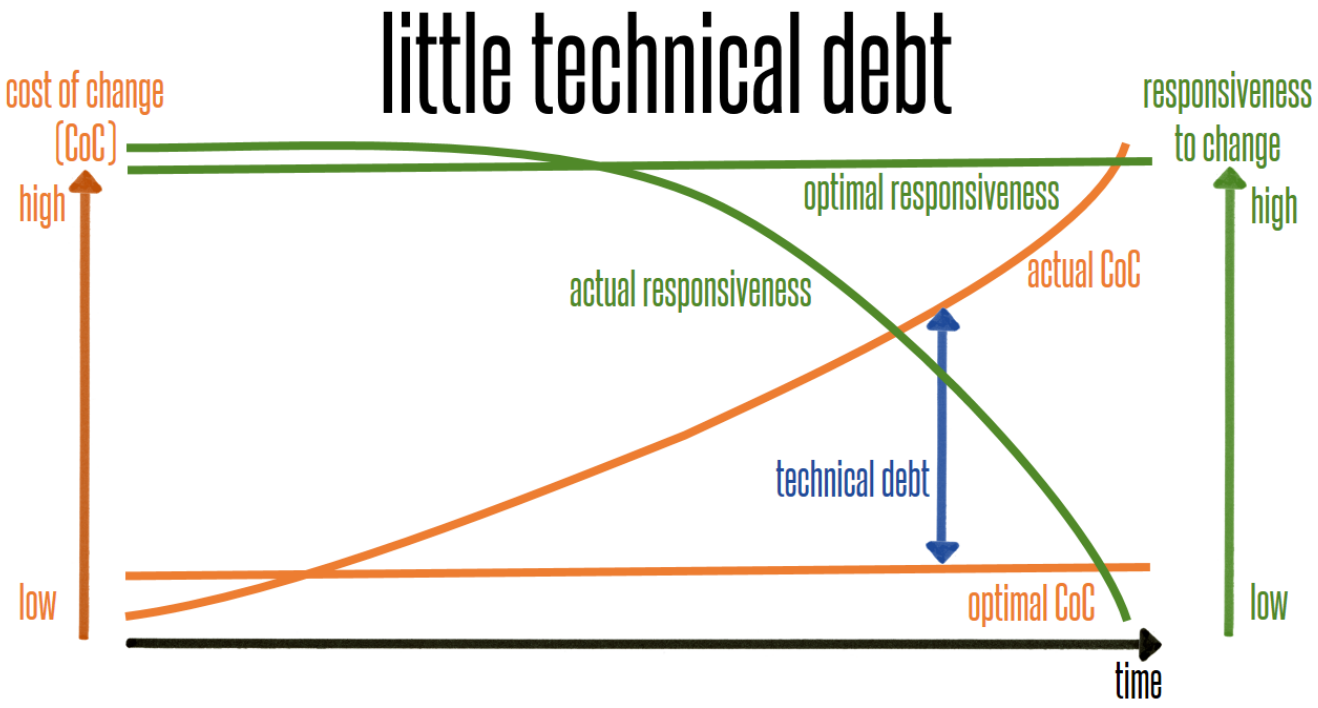
Clean Code

Applicability	EBX Add-ons	QMS Level	3
Purpose	Make easier to read and understand, to change, improve or refactor code	**Page Status**	PUBLISHED
Standards & Compliance	ISO 27001: A.14.2.1	**Publish Date**	09 Sep 2020
Owner/Author Email	Minh Tran Quang	**Version**	1.0
Classification	For internal use only - TIBCO Confidential		
Page Editing	This page limits editing; suggestions and changes are welcome and can be made via the page comments or contact the owner(s) of this page.		

1.What is clean code?

is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently

- Easy to read and understand
- Easy to change, improve or refactor



```
protected void handleFKQueries(IndexingDefinition indexDefinition)
{
    // TODO Recursive FK cache, MultiHop cache is better, refactor latter
    List<FieldIndexingDefinition> fkQueries = new ArrayList<FieldIndexingDefinition>
    for (FieldIndexingDefinition query : indexDefinition.getFieldIndexingDefinitions().va
    {
        if (!TeseSearchUtil.getInstance()
            .isProgrammaticLabel(query.getSchemaNode(), indexDefinition.getLocale())
            && query.getSchemaNode().getFacetOnTableReference() != null)
        {
            fkQueries.add(query);
        }
    }
```

2. Why Clean Code?

- Improve yourself
    - Seniority doesn't depend on how many years of experience
- Bad code is error and buggy more effort to develop new features
- Readable Maintainable easy to change or extend Don't have to rework any addon

3. General rules

- Follow standard conventions/Design pattern/Popular data structure
- Boy scout rule: leave the campground cleaner than you found it.
    - Never add more dirty code
- Always find root cause: always look for the root cause of a problem.
    - Dirty fixs always take more time to maintain

```
final MatchingTableConfig currentConfigRecord = MatchingTableConfigDAO.instance()
    .getBySchemaNode(currentTable);
```

- Keep it simple stupid: Simpler is always better, reduce complexity as much as possible.
- Avoid negative conditionals

```
if (!equalsIgnoreLength(this, other))
    return false;
return true;
```

```
return equalsIgnoreLength(this, other);
```

```
if (other.currentLength == length)
{
    for (int i = 0; i < length; i++)
        if (doubleMetaphoneToken.primary[i] != other.primary[i])
            return false;
}
else
{
    return false;
}
```

```
if (other.currentLength != length)
    return false;
for (int i = 0; i < length; i++)
    if (doubleMetaphoneToken.primary[i] != other.primary[i])
        return false;
```

```
Set<String> rawValue= rawValues.get(fieldIndexingDefinition.getSchemaNode());
if (rawValue == null)
   rawValue = MapSetBuilder.get().getSet();
```

```
Set<String> rawValue;
if (!rawValues.containsKey(fieldIndexingDefinition.getSchemaNode()))
   rawValue = MapSetBuilder.get().getSet();
else
   rawValue = rawValues.get(fieldIndexingDefinition.getSchemaNode());
```

4. Functions rules

- Small, do one thing (and do it really well!!!)
- Use descriptive names.
- Prefer fewer arguments.
- Have no side effects.
- Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

```
private StringBuilder buildQuery(
   final boolean isManualMergeOnly,
   final String configTableKey,
   boolean hasActiveMergePolicy)
{
```

Query for what purpose?

flag argument

flag argument

do many things then don't know what should return

do many things

```
/**
 * Scan queries for surrogate, handle FK queries, calculate BaseScore. This function also set FKExtendedQueries for
 */
public void scanStandardQueries(
   TeseConfiguration configuration,
   List<TeseQuery> standardQueries,
   MapForSearchManager mapForSearch,
   IndexSearchContext context,
   OLD_ScoringPolicy policy,
   List<TeseQuery> queriesForSurrogate)
```

which will be modified???

```
private List<?> getRecordDataList(
    final Adaptation record,
    final Path rootPath,
    final Field field,
    final Locale locale,
    final Map<String, List<Adaptation>> linkedRecords,
    final UIServiceComponentWriter uiComponentWriter)
{
    final Object fieldValue = record.get(rootPath.add(field.getRelativePath()));

    final int subRecordCount = ((List<?>) fieldValue).size();

else if (field.getMaxOccurs() > 1)
{
    return this.getRecordDataList(
        record,
        rootPath,
        field,
        locale,
        linkedRecords,
        uiComponentWriter);
}
```

name of method should mention about multiple values scenario

WTF????

surprise!!!

5. Functions side effects

- Main effects: returning a value to the invoker of the operation
- Side effects: modifies some state variable value(s) outside its local environment (local variables, parameters)
- Problems:
    - Concurrency
    - Hard to read and control

```
public void add(Adaptation record)
{
    RecordIndex recordIndex = this.getRecordFactory()
        .buildRecordIndex(record, this.getIndexingDefinition());
    this.tableIndex.put(recordIndex.getPrimaryKeyAsString(), recordIndex);
    Filter filter = this.definition.buildFilter(record);
    this.addRecordIndexToCache(filter, recordIndex);
}
```

a side effect

not a side effect

2 operations

```
private RecordIndex createRecordAndPutToCache(Adaptation adaptation)
{
    String primaryKey = adaptation.getOccurrencePrimaryKey().format();
    RecordIndex record = new RecordIndex(adaptation, this);
    this.data.put(primaryKey, record);
    return record;
}
```

2 operations

plural

```
private void addPreviewAndSummaryStep(List<MergeStep> mergeSteps, int step, Locale locale)
{
```

6. Comments rules

- Always try to explain yourself in code.
- Don't be redundant.
- Don't add obvious noise.
- **Don't comment out code.** Just remove.
- Use as explanation of intent.
- Use as clarification of code.
- Use as warning of consequences.

7. Source code structure rules (not mandatory)

- Declare variables close to their usage **readable and avoid redundant initialization**
- Dependent functions and similar functions should be close.
- Place functions in the downward direction.
- Keep lines short.
- Don't use horizontal alignment.
- Use white space to associate related things and disassociate weakly related.
- Don't break indentation.

8. Objects and data structures

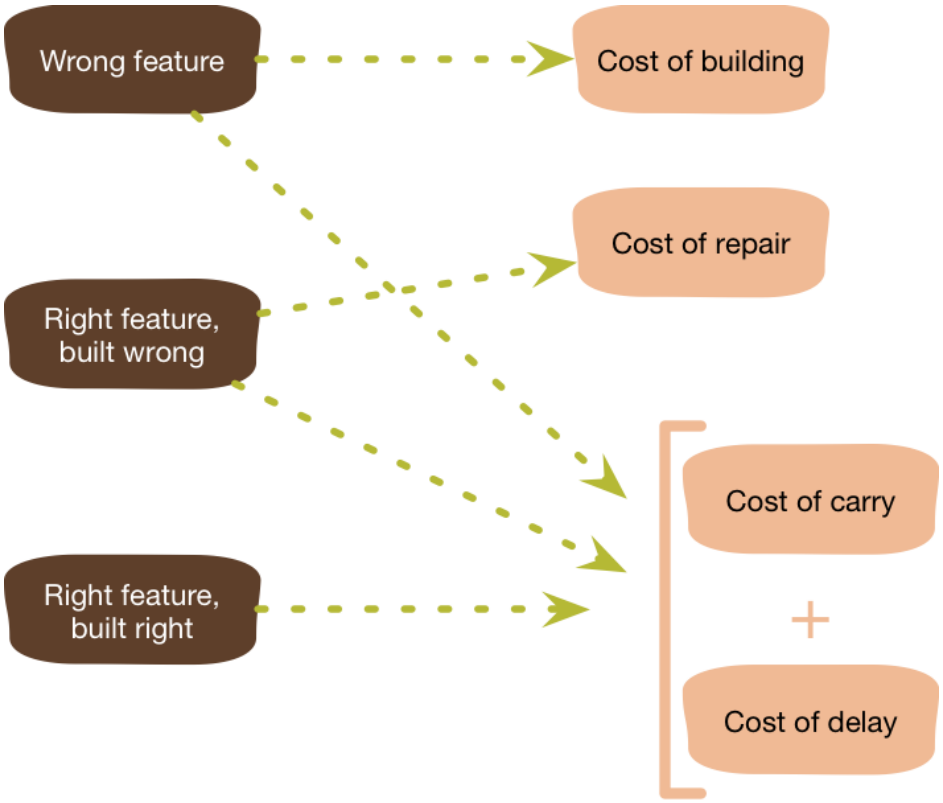
- Hide internal structure **reduce coupling and easy to refactor**
- Should be small and do one thing **clear and reusable**
- Small number of instance variables.
    - Waste memory if don’t use all variables
    - Purpose of object isn’t clear
- Base class should know nothing about their derivatives **extendable**
- Better to have many functions than to pass some code into a function to select a behavior a function **“Should be small and do one thing”**
- Prefer non-static methods to static methods.
    - Static methods cannot polymorphism
    - Static methods don’t have context
    - Static methods should be stateless

9. Don't repeat yourself

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
    - Insert to table
    - Get business value
- Easy to modify/improve/refactor
    - A single place to control, we have to update all duplicated code when we update business
    - Newcomers will easy to catch up

10. YAGNI - You ain't gonna need it

- Feature aspect, not abstraction nor structure
- Unnecessary features will add unnecessary effort and unnecessary bugs
- "Code for today not tomorrow" - dla
- More information
- KISS: Keep it simple, stupid! can your code be more simple?



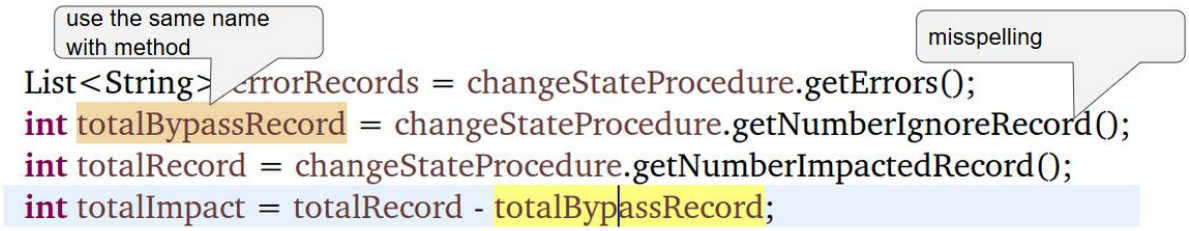

```
List<FieldSurvivor> remainingSurvivingFields = normalizedFields.stream()
    .filter(
        field -> survivingFields.stream().noneMatch(
            survivingField -> survivingField.getFieldPath()
                .equals(field.getAbsolutePath().format())))
    .filter(
        field -> manualMergePolicy.getAutoCreateNewGoldenMode() == DISABLED
            || !field.isPrimary() || hasAutoGoldenRecord)
    .filter(
        field -> manualMergePolicy.getAutoCreateNewGoldenMode() == DISABLED
            || AddonStringUtils.isEmpty(manualMergePolicy.getAutoCreateNewGoldenSource())
            || !field.getAbsolutePath()
                .format()
                .equals(matchingTableConfig.getSourceField()))
    .map(
        field -> new FieldSurvivor(
            field.getAbsolutePath().format(),
            survivingRecord.getOccurrencePrimaryKey().format()))
    .collect(Collectors.toList());
```

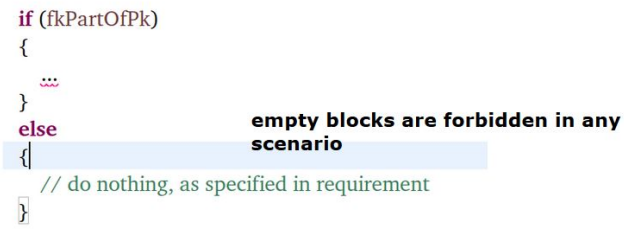
O(n^2) complexity

duplicated

duplicated

not duplicated but should cache this value





Sign-off History

Action	Name	Date
Prepared by	Thi Viet Phuong Luu	21 Feb 2020
Approved	Minh Tran Quang	09 Sep 2020

Revision History

Version	Date	Authors	Description
1.0	09 Sep 2020	Minh Tran Quang	Initial version