Survey on Multi-language Design Smells

Thank you for agreeing to participate, it will take around 30 minutes to complete.

Study Policy:

- Participation in this study is completely voluntary. If you decide not to participate there will not be any negative consequences. If you decide to participate, you may stop participating at any time and withdraw entirely your participation or you may decide not to answer any specific question.
- Your identity and the data collected thanks to your participation will remain anonymous and will never be released to the public. Only anonymous data (aggregated or not) will be published in scientific articles, ensuring that the data cannot be linked back to a particular participant. The data will be kept by the principal investigator for five years before being destroyed.
- By submitting this survey, you are indicating that you have read the description of the study, are over the age of 18, and that you agree to the terms and consent as described in https://drive.google.com/file/d/1aZfHRCr0bEX0i33l_oQHlS9ui9h6rlC5/view?usp=sharing

If you have any questions, please contact us at mouna.abidi@polymtl.ca

Study Design: The purpose of this study is to investigate the prevalence of design smells related to multi-language systems. These systems are developed using more than one programming language. We aim to investigate the perceived prevalence and impact of the design smells detailed below. Our main goal is to improve the quality of those systems.

Definition of terminologies:

Not Handling Exceptions	The exceptions are not handled, developers generally rely on the exceptions provided by the other language
Assuming Safe Return Value	A value is returned to the other language without being checked. Thus, the interaction between both languages may not be correctly performed
Excessive Inter-language Communication	A wrong partitioning in both languages leads to many calls in a way or the other. It adds complexity takes more time to run and may indicate a bad separation of concerns
Too Much Clustering	The multi-language code is concentrated in a few classes, regardless of their concerns and responsibilities.
Too Much Scattering	Many classes are scarcely used in multi-language communication
Hard Coding Libraries	When different libraries are needed depending on the operating system, they are not loaded with conditions on the operating system, but for instance, with a try-catch mechanism, making it hard to know which library has really been loaded
Local References Abuse	The developer does not manage the memory in the native space properly and does not release local and global references
Memory Management Mismatch	Reference types passed from one language to another are not released in a language that does not handle the management of memory causing memory leaks
Not Caching Objects	A method is called to retrieve a field every time this field is needed, although the field's ID or value could have been cached.
Not Securing Libraries	The code loads a foreign library without any security check or restriction privilege
Not Using Relative Path	A library is loaded using only the name not the path. It cannot be accessed in the same way from everywhere
Excessive Objects	A whole object is passed as an argument, although only some of the fields were needed, and it would have been better for the system performance to pass only these fields
Unused Method Declaration	A method is declared in the host language but not implemented in the foreign language
Unused Method Implementation	A method is declared in the host language and implemented in the foreign language, but never called from the host language
Unused Parameters	Some arguments of a function are used neither in its body nor in the other language.

(Khomh, F., & Gueheneuce, Y. G. (2008, April). Do design patterns impact software quality positively?. In Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on (pp. 274-278).

IEEE.)

− Expandability: The degree to which the design of a system can be extended.

− Simplicity: The degree to which the design of a system can be understood easily.

− Reusability: The degree to which a piece of design can be reused in another design.

− Learnability: The degree to which the code source of a system is easy to learn.

− Understandability: The degree to which the code source can be understood easily.

− Performance: The degree to which the code meets its requirements for timeliness.

− Modularity: The degree to which the implementation of the functions of a system is independent of one another.

Thank you.

Best regards,

*** 1. How often do you encounter the following design smells in your project(s)?**

Please check the definitions provided above before answering this questions

	1 Very Often	2 Often	3 Rarely	N/A
Not Handling Exceptions	○	○	○	○
Assuming Safe Return Value	○	○	○	○
Excessive Inter-language Communication	○	○	○	○
Too Much Clustering	○	○	○	○
Too Much Scattering	○	○	○	○
Hard Coding Libraries	○	○	○	○
Local References Abuse	○	○	○	○
Memory Management Mismatch	○	○	○	○
Not Caching Objects	○	○	○	○
Not Securing Libraries	○	○	○	○
Not Using Relative Path	○	○	○	○
Excessive Objects	○	○	○	○
Unused Method Declaration	○	○	○	○
Unused Method Implementation	○	○	○	○
Unused Parameters	○	○	○	○

*** 2. How do you evaluate the impact of the following design smells in those software quality attributes?**

Please carefully read the definition of the smells provided bellow and the reference provided.
(VN: Very Negative, N: Negative, NS: Not significant/Neutral, P: Positive, and VP: Very Positive)

	Expandability	Simplicity	Reusability	Learnability	Understandability	Performance	Modularity	N/A
Not Handling Exceptions								
Assuming Safe Return Value								
Excessive Inter-language Communication								
Too Much Clustering								
Too Much Scattering								
Hard Coding Libraries								
Local References Abuse								
Memory Management Mismatch								
Not Caching Objects								
Not Securing Libraries								
Not Using Relative Path								
Excessive Objects								
Unused Method Declaration								
Unused Method Implementation								
Unused Parameters								

*** 3. Please rank the following design smells from the most harmful to the less harmful**
(Most harmful to the less harmful: 15 -> 1)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Not Handling Exceptions

Assuming Safe Return Value

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Excessive Inter-language Communication

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Too Much Clustering

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Too Much Scattering

Hard Coding Libraries

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Local References Abuse

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Memory Management Mismatch

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Not Caching Objects

Not Securing Libraries

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Not Using Relative Path

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Excessive Objects

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Unused Method Declaration

Unused Method Implementation

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Unused Parameters

*** 4. Task:**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
jmethodID JNI_getStaticMethodIDOrNull(jclass clazz, const char* name, const char* sig)
{
jmethodID result;
BEGIN_CALL
result = (*env)->GetStaticMethodID(env, clazz, name, sig);
END_CALL
return result;
}
```

- ○ Yes
- ○ No

5. b) If YES, please provide an explanation or specify the design smell(s) involved?

6. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 7. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 8. e) If YES, would you apply this refactored solution?**

```
jmethodID JNI_getStaticMethodIDOrNull(jclass clazz, const char* name, const char* sig)
{
jmethodID result;
jobject exh;
BEGIN_CALL
result = (*env)->GetStaticMethodID(env, clazz, name, sig);
if(result == 0) {
exh = (*env)->ExceptionOccurred(env);
}
END_CALL
return result;
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 9. <u>Task:</u>**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
static bool check_enabled(bool *newval, void **extra, GucSource source)
{
if ( initstage < IS_PLJAVA_ENABLED )
return true;
if ( *newval )
return true;
GUC_check_errmsg(
"too late to change \"pljava.enable\" setting");
return false;
}
```

- ○ Yes
- ○ No

10. b) If YES, please provide an explanation or specify the design smell(s) involved?

11. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 12. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 13. e) If YES, would you apply this refactored solution?**

```
static bool check_enabled(bool *newval)
{
if ( initstage < IS_PLJAVA_ENABLED )
return true;
if ( *newval )
return true;
GUC_check_errmsg(
"too late to change \"pljava.enable\" setting");
return false;
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 14. <u>Task:</u>**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
jshort* JNI_getShortArrayElements(jshortArray array, jboolean* isCopy)
{
jshort* result;
BEGIN_JAVA
result = (*env)->GetShortArrayElements(env, array, isCopy);
END_JAVA
return result;
}
```

- ○ Yes
- ○ No

15. b) If YES, please provide an explanation or specify the design smell(s) involved?

16. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 17. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A

○	○	○	○	○	○

*** 18. e) If YES, would you apply this refactored solution?**

```
jshort* JNI_getShortArrayElements(jshortArray array, jboolean* isCopy){
jshort* result;
result = (*env)->GetShortArrayElements(env, array, isCopy);
releaseShortArrayElements(env, array, isCopy);
return result;
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 19. Task:**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
static jobject obtainUDTHandle( jmethodID which, jclass clazz, char *langName, bool trusted)
{
jstring jname = String_createJavaStringFromNTS(langName);
jobject result = JNI_callStaticObjectMethod(s_Function_class,
which, clazz, jname, trusted ? JNI_TRUE : JNI_FALSE);
JNI_deleteLocalRef(jname);
return result;
}
```

- ○ Yes
- ○ No

20. b) If YES, please provide an explanation or specify the design smell(s) involved?

21. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 22. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 23. e) If YES, would you apply this refactored solution?**

```
static jobject obtainUDTHandle( jmethodID which, jclass clazz, char *langName, bool trusted)
{
jstring jname = String_createJavaStringFromNTS(langName);
jobject result = JNI_callStaticObjectMethod(s_Function_class,
which, clazz, jname, trusted ? JNI_TRUE : JNI_FALSE);
JNI_deleteLocalRef(jname);
if result !== null)){
return result;
}}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 24. Task:**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
public class TriggerData{
private static native Relation _getRelation(long pointer) throws SQLException;
private static native Tuple _getTriggerTuple(long pointer) throws SQLException;
private static native Tuple _getNewTuple(long pointer) throws SQLException;
private static native String[] _getArguments(long pointer) throws SQLException;
private static native String _getName(long pointer) throws SQLException;
private static native boolean _isFiredAfter(long pointer) throws SQLException;
private static native boolean _isFiredBefore(long pointer) throws SQLException;
private static native boolean _isFiredForEachRow(long pointer) throws SQLException;
private static native boolean _isFiredForStatement(long pointer) throws SQLException;
private static native boolean _isFiredByDelete(long pointer) throws SQLException;
private static native boolean _isFiredByInsert(long pointer) throws SQLException;
private static native boolean _isFiredByUpdate(long pointer) throws SQLException;

}
```

- ○ Yes
- ○ No

25. b) If YES, please provide an explanation or specify the design smell(s) involved?

26. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 27. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 28. e) If YES, would you apply this refactored solution?**

```
public class Tuple{
private static native Relation _getRelation(long pointer) throws SQLException;
private static native Tuple _getTriggerTuple(long pointer) throws SQLException;
private static native Tuple _getNewTuple(long pointer) throws SQLException;
private static native String[] _getArguments(long pointer) throws SQLException;
private static native String _getName(long pointer) throws SQLException;
}
public class Fired{
private static native boolean _isFiredAfter(long pointer) throws SQLException;
private static native boolean _isFiredBefore(long pointer) throws SQLException;
private static native boolean _isFiredForEachRow(long pointer) throws SQLException;
private static native boolean _isFiredForStatement(long pointer) throws SQLException;
private static native boolean _isFiredByDelete(long pointer) throws SQLException;
private static native boolean _isFiredByInsert(long pointer) throws SQLException;
private static native boolean _isFiredByUpdate(long pointer) throws SQLException;
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 29. <u>Task:</u>**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
package org.postgresql.pljava.internal;

public class Tuple
{
private static native Object _getObject(long pointer, long tupleDescPointer, int index, Class type)
}

package org.postgresql.pljava.internal;
import java.sql.SQLException;

public class Relation
{
private static native String _getName(long pointer)
throws SQLException;
private static native String _getSchema(long pointer)
throws SQLException;
private static native TupleDesc _getTupleDesc(long pointer)
throws SQLException;
private static native Tuple _modifyTuple(long pointer, long original, int[] fieldNumbers, Object[] values)
throws SQLException;
}

package org.postgresql.pljava.internal;
public class Session
{
private static native boolean _setUser(AclId userId, boolean isLocalChange);
}

public final class AclId
{
private static native AclId _getUser();
private static native AclId _getOuterUser();
```

```
private static native AclId _fromName(String name);
private native String _getName();
private native boolean _hasSchemaCreatePermission(Oid oid);
private native boolean _isSuperuser();
}
```

- Yes
- No

30. b) If YES, please provide an explanation or specify the design smell(s) involved?

31. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 32. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 33. e) If YES, would you apply this refactored solution?**

```
package org.postgresql.pljava.internal;
import java.sql.SQLException;
public class Tuple
{
private static native Object _getObject(long pointer, long tupleDescPointer, int index, Class type)
static native String _getName(long pointer)
throws SQLException;
private static native String _getSchema(long pointer)
throws SQLException;
private static native TupleDesc _getTupleDesc(long pointer)
throws SQLException;
private static native Tuple _modifyTuple(long pointer, long original, int[] fieldNumbers, Object[] values)
throws SQLException;

}

package org.postgresql.pljava.internal;
public final class AclId
{
private static native AclId _getUser();
private static native AclId _getOuterUser();
private static native AclId _fromName(String name);
private native String _getName();
private native boolean _hasSchemaCreatePermission(Oid oid);
private native boolean _isSuperuser();
```

```
private static native boolean _setUser(AclId userId, boolean isLocalChange);
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 34. Task:**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
private static native Oid _getOid(long _this, int index) throws SQLException;
public Class getColumnClass(int index)
throws SQLException
{
if(m_columnClasses == null)
{
m_columnClasses = new Class[m_size];
doInPG(() ->
{
long _this = this.getNativePointer();
for(int idx = 0; idx < m_size; ++idx)
m_columnClasses[idx] = _getOid(_this, idx+1).getJavaClass();
});
}
return m_columnClasses[index-1];
}
```

- ○ Yes
- ○ No

35. b) If YES, please provide an explanation or specify the design smell(s) involved?

36. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 37. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 38. e) If YES, would you apply this refactored solution?**

```
private static native Oid _getOid(long _this, int index, int m_size) throws SQLException;
public Class getColumnClass(int index)
throws SQLException
```

```
{
if(m_columnClasses == null)
{
m_columnClasses = new Class[m_size];
doInPG(() ->
{
long _this = this.getNativePointer();
this._getOid(_this, idx+1,m_size).getJavaClass();
});
}
return m_columnClasses[index-1];
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 39. Task:**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
private native void _heapFreeTuple(long pointer);

protected void javaStateUnreachable(boolean nativeStateLive)
{
assert Backend.threadMayEnterPG();
if ( nativeStateLive )
_freeTupleDesc(guardedLong());
}

private native void _freeTupleDesc(long pointer);

// C++

JNIEXPORT void JNICALL
Java_org_postgresql_pljava_internal_DualState_00024SingleFreeTupleDesc__1freeTupleDesc(JNIEnv* env, jobject _this, jlong pointer)
{
BEGIN_NATIVE_NO_ERRCHECK
Ptr2Long p2l;
p2l.longVal = pointer;
FreeTupleDesc(p2l.ptrVal);
END_NATIVE
}
```

- ○ Yes
- ○ No

40. b) If YES, please provide an explanation or specify the design smell(s) involved?

41. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 42. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 43. e) If YES, would you apply this refactored solution?**

```
protected void javaStateUnreachable(boolean nativeStateLive)
{
assert Backend.threadMayEnterPG();
if ( nativeStateLive )
_freeTupleDesc(guardedLong());
}

private native void _freeTupleDesc(long pointer);

// C++

JNIEXPORT void JNICALL
Java_org_postgresql_pljava_internal_DualState_00024SingleFreeTupleDesc__1freeTupleDesc(JNIEnv* env, jobject _this, jlong pointer)
{
BEGIN_NATIVE_NO_ERRCHECK
Ptr2Long p2l;
p2l.longVal = pointer;
FreeTupleDesc(p2l.ptrVal);
END_NATIVE
}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

*** 44. <u>Task:</u>**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
private static native int _getStatementCacheSize();
public static int getStatementCacheSize()
{
return doInPG(Backend::_getStatementCacheSize);
}


JNIEXPORT jint JNICALL Java_org_postgresql_pljava_internal_Backend__1getStatementCacheSize(JNIEnv* env, jclass cls)
{
return statementCacheSize;
}
JNIEXPORT jboolean JNICALL Java_org_postgresql_pljava_internal_Backend_isReleaseLingeringSavepoints(JNIEnv* env, jclass cls)
{
return pljavaReleaseLingeringSavepoints ? JNI_TRUE : JNI_FALSE;
}
```

- ○ Yes
- ○ No

45. b) If YES, please provide an explanation or specify the design smell(s) involved?

46. c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?

*** 47. d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

*** 48. e) If YES, would you apply this refactored solution?**

```
private static native int _getStatementCacheSize();
public static int getStatementCacheSize()
{
return doInPG(Backend::_getStatementCacheSize);
}

JNIEXPORT jint JNICALL Java_org_postgresql_pljava_internal_Backend__1getStatementCacheSize(JNIEnv* env, jclass cls)
{
return statementCacheSize;
}
```

○ Yes (Refactor with this solution)
○ Yes (Refactor with an alternative solution)
○ No (No refactoring)

*** 49. <u>Task:</u>**

a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
public void rollback(Savepoint savepoint) throws SQLException
{
if(!(savepoint instanceof PgSavepoint))
throw new IllegalArgumentException("Not a PL/Java Savepoint");
PgSavepoint sp = (PgSavepoint)savepoint;
Invocation.clearErrorCondition();
sp.rollback();
}
```

○ Yes
○ No

50. b) If YES, please provide an explanation or specify the design smell(s) involved?

51. **c) If YES, (In your opinion,) What is the motivation behind using this specific way of implementation?**

* 52. **d) Please rate the severity of the implementation problem (if any), from 1 (Very Low) to 5 (Very High)**

1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A
○	○	○	○	○	○

* 53. **e) If YES, would you apply this refactored solution?**

```
public void rollback(Savepoint savepoint) throws SQLException
{
if(!(savepoint instanceof PgSavepoint))
throw new IllegalArgumentException("Not a PL/Java Savepoint");
PgSavepoint sp = (PgSavepoint)savepoint;
if (sp !== null)){
Invocation.clearErrorCondition();
sp.rollback();
}}
```

- ○ Yes (Refactor with this solution)
- ○ Yes (Refactor with an alternative solution)
- ○ No (No refactoring)

Your responses have been registered!

Thank you for taking the time to complete the survey, your input is valuable to us.