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)?

```
static {
Loader.load();
System.loadLibrary("jniCalc");
}
```

- ○ 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?**

```
public static void loadLibrary
static {
AccessController.doPrivileged( new PrivilegedAction() {
 public static void init() {
Loader.load();
System.loadLibrary("jniCalc");
}} }
```

- ○ 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)?

```
String join(String separator, Iterable strings) {
String string = "";
for (String s : strings) {
string += (string.length() > 0 ? separator : "") + s;
}
return string;
}
```

- ○ 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
○	○	○	○	○	○

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

```
String join(String separator, Iterable<String> strings) {
String string = "";
if (s !== null)){
for (String s : strings) {
string += (string.length() > 0 ? separator : "") + s;
}}
return string;
}
```

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

*** 14. Task:**

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

```
public class AdapterTest {
static native @StdString String testStdString(@StdString String str);
static native @StdString BytePointer testStdString(@StdString BytePointer str);
static native @StdWString CharPointer testStdWString(@StdWString CharPointer str);
static native @StdWString IntPointer testStdWString(@StdWString IntPointer str);
static native String testCharString(String str);
static native @Cast("char*") BytePointer testCharString(@Cast("char*") BytePointer str);
static native CharPointer testShortString(CharPointer str);
static native IntPointer testIntString(IntPointer str);
static native @Const @ByRef @StdString byte[] getConstStdString();
static class SharedData{
SharedData(Pointer p) { super(p); }
SharedData(int data) { allocate(data); }
native void allocate(int data);
native int data(); native SharedData data(int data);
static native @SharedPtr SharedData createSharedData();
static native void storeSharedData(@SharedPtr SharedData s);
static native @SharedPtr SharedData fetchSharedData();
static class UniqueData {
UniqueData(Pointer p) { super(p); }
UniqueData(int data) { allocate(data); }
native void allocate(int data);
native int data(); native UniqueData data(int data);
@Function static native @UniquePtr UniqueData createUniqueData();
static native void createUniqueData(@UniquePtr UniqueData u);
static native void storeUniqueData(@Const @UniquePtr UniqueData u);
static native @Const @UniquePtr UniqueData fetchUniqueData();
static native int constructorCount();
static native void constructorCount(int c);
static native int destructorCount();
static native void destructorCount(int c);
static native @StdVector IntPointer testStdVectorByVal(@StdVector IntPointer v);
static native @StdVector IntPointer testStdVectorByRef(@StdVector IntBuffer v);
static native @StdVector int[] testStdVectorByPtr(@StdVector int[] v);
static native @Cast("const char**") @StdVector PointerPointer testStdVectorConstPointer(@Cast("const char**") @StdVector PointerPointer v);
}
```

- ○ 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?

```
static class TestStd{
static native @StdString String testStdString(@StdString String str);
static native @StdString BytePointer testStdString(@StdString BytePointer str);
static native @StdWString CharPointer testStdWString(@StdWString CharPointer str);
static native @StdWString IntPointer testStdWString(@StdWString IntPointer str);
static native String testCharString(String str);
static native @Cast("char*") BytePointer testCharString(@Cast("char*") BytePointer str);
static native CharPointer testShortString(CharPointer str);
static native IntPointer testIntString(IntPointer str);
static native @Const @ByRef @StdString byte[] getConstStdString();
static native @StdVector IntPointer testStdVectorByVal(@StdVector IntPointer v);
static native @StdVector IntPointer testStdVectorByRef(@StdVector IntBuffer v);
static native @StdVector int[] testStdVectorByPtr(@StdVector int[] v);
static native @Cast("const char**") @StdVector PointerPointer testStdVectorConstPointer(@Cast("const char**") @StdVector PointerPointer v);
}

static class SharedData{
SharedData(Pointer p) { super(p); }
SharedData(int data) { allocate(data); }
native void allocate(int data);
native int data();
native SharedData data(int data);
static native @SharedPtr SharedData createSharedData();
static native void storeSharedData(@SharedPtr SharedData s);
static native @SharedPtr SharedData fetchSharedData();
}

static class UniqueData {
UniqueData(Pointer p) { super(p); }
UniqueData(int data) { allocate(data); }
native void allocate(int data);
native int data();
native UniqueData data(int data);
@Function static native @UniquePtr UniqueData createUniqueData();
static native void createUniqueData(@UniquePtr UniqueData u);
native void storeUniqueData(@Const @UniquePtr UniqueData u);
```

```
static native @Const @UniquePtr UniqueData fetchUniqueData();
}
```

- ○ 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)?

```
public class BoolPointer extends Pointer {
private native void allocateArray(long size);
public native BoolPointer put(long i, boolean b);
}

public class BooleanPointer extends Pointer {
private native void allocateArray(long size);
public native boolean get(long i);
public native BooleanPointer put(long i, boolean b);
public native BooleanPointer get(boolean[] array, int offset, int length);
public native BooleanPointer put(boolean[] array, int offset, int length);
}

public class CLongPointer extends Pointer {
private native void allocateArray(long size);
public native long get(long i);
public native CLongPointer put(long i, long l);
}

public class Pointer{
private native void allocate(Buffer b);
private native void deallocate(long ownerAddress, long deallocatorAddress);
private native ByteBuffer asDirectBuffer();
public static native Pointer malloc(long size);
public static native Pointer calloc(long n, long size);
public static native Pointer realloc(Pointer p, long size);
public static native void free(Pointer p);
public static native Pointer memchr(Pointer p, int ch, long size);
public static native int memcmp(Pointer p1, Pointer p2, long size);
public static native Pointer memcpy(Pointer dst, Pointer src, long size);
public static native Pointer memmove(Pointer dst, Pointer src, long size);
public static native Pointer memset(Pointer dst, int ch, long size);
}
```

- ○ 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?**

```
public class BooleanPointer extends Pointer {
public native boolean get(long i);
public native BooleanPointer put(long i, boolean b);
public native BooleanPointer put(long i, boolean b);
public native BooleanPointer get(boolean[] array, int offset, int length);
public native BooleanPointer put(boolean[] array, int offset, int length);
}

public class CLongPointer extends Pointer {
public native long get(long i);
public native CLongPointer put(long i, long l);
}

public class Pointer{
private native void allocate(Buffer b);
private native void deallocate(long ownerAddress, long deallocatorAddress);
private native void allocateArray(long size);
private native ByteBuffer asDirectBuffer();
public static native Pointer malloc(long size);
public static native Pointer calloc(long n, long size);
public static native Pointer realloc(Pointer p, long size);
public static native void free(Pointer p);
public static native Pointer memchr(Pointer p, int ch, long size);
public static native int memcmp(Pointer p1, Pointer p2, long size);
public static native Pointer memcpy(Pointer dst, Pointer src, long size);
public static native Pointer memmove(Pointer dst, Pointer src, long size);
public static native Pointer memset(Pointer dst, int ch, long size);
}
```

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

*** 24. <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 boolean trimMemory();
protected
P deallocator(Deallocator deallocator) {
if (deallocator != null && !deallocator.equals(null)) {
DeallocatorReference r = deallocator instanceof DeallocatorReference ?
(DeallocatorReference)deallocator : new DeallocatorReference(this, deallocator);
this.deallocator = r;
int count = 0;
long lastPhysicalBytes = maxPhysicalBytes > 0 ? physicalBytes() : 0;
```

```
synchronized (DeallocatorThread.class) {
try {
while (count++ < maxRetries && ((maxBytes > 0 && DeallocatorReference.totalBytes + r.bytes > maxBytes)
|| (maxPhysicalBytes > 0 && lastPhysicalBytes > maxPhysicalBytes))) {
if (logger.isDebugEnabled()) {
logger.debug("Calling System.gc() and Pointer.trimMemory() in " + this);
}
System.gc();
Thread.sleep(100);
trimMemory();
lastPhysicalBytes = maxPhysicalBytes > 0 ? physicalBytes() : 0;
}
} catch (InterruptedException ex) {
Thread.currentThread().interrupt();
}
}
}
```

- ○ 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?**

```
private static native boolean trimMemory();
protected <P extends Pointer> P deallocator(Deallocator deallocator) {
if (deallocator != null && !deallocator.equals(null)) {
DeallocatorReference r = deallocator instanceof DeallocatorReference ?
(DeallocatorReference)deallocator : new DeallocatorReference(this, deallocator);
this.deallocator = r;
int count = 0;
long lastPhysicalBytes = maxPhysicalBytes > 0 ? physicalBytes() : 0;
synchronized (DeallocatorThread.class) {
try {
while (count++ < maxRetries && ((maxBytes > 0 && DeallocatorReference.totalBytes + r.bytes > maxBytes)
|| (maxPhysicalBytes > 0 && lastPhysicalBytes > maxPhysicalBytes))) {
if (logger.isDebugEnabled()) {
logger.debug("Calling System.gc() and Pointer.trimMemory() in " + this);
```

```
}
System.gc();
Thread.sleep(100);
lastPhysicalBytes = maxPhysicalBytes > 0 ? physicalBytes() : 0;
}
trimMemory();
} catch (InterruptedException ex) {
Thread.currentThread().interrupt();
}
}
}
```

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

*** 29. Task:**

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

```
public static native int memcmp(Pointer p1, Pointer p2, long size);
public static native Pointer memcpy(Pointer dst, Pointer src, long size);
public static native Pointer memmove(Pointer dst, Pointer src, long size);

public
P put(Pointer p) {
if (p.limit > 0 && p.limit < p.position) {
throw new IllegalArgumentException("limit < position: (" + p.limit + " < " + p.position + ")");
}
int size = sizeof();
int psize = p.sizeof();
long length = psize * (p.limit <= 0 ? 1 : p.limit - p.position);
position *= size;
p.position *= psize;
memcpy(this, p, length);
position /= size;
p.position /= psize;
return (P)this;
}
```

- ○ 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?**

```
public static native Pointer memcpy(Pointer dst, Pointer src, long size);
public <P extends Pointer> P put(Pointer p) {
if (p.limit > 0 && p.limit < p.position) {
throw new IllegalArgumentException("limit < position: (" + p.limit + " < " + p.position + ")");
}
int size = sizeof();
int psize = p.sizeof();
long length = psize * (p.limit <= 0 ? 1 : p.limit - p.position);
position *= size;
p.position *= psize;
memcpy(this, p, length);
position /= size;
p.position /= psize;
return (P)this;
}
```

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

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

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

```
String[] merge(String[] ss, String s) {
if (ss != null && s != null) {
ss = Arrays.copyOf(ss, ss.length + 1);
ss[ss.length - 1] = s;
} else if (s != null) {
ss = new String[] { s };
}
return ss : new String[0];
}
```

○ 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?**

```
String[] merge(String[] ss, String s) {
if (ss != null && s != null) {
ss = Arrays.copyOf(ss, ss.length + 1);
ss[ss.length - 1] = s;
} else if (s != null) {
ss = new String[] { s };
}
if (ss != null ){
return ss != null ? ss : new String[0];
}}
```

- ○ 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.