#### Survey on Multi-language Design Smells\_JavaCpp

### 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 <a href="https://drive.google.com/file/d/1aZfHRCr0bEX0i331\_oQHIS9ui9h6rlC5/view?usp=sharing">https://drive.google.com/file/d/1aZfHRCr0bEX0i331\_oQHIS9ui9h6rlC5/view?usp=sharing</a>

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

<u>Study Design:</u> 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	A value is returned to the other language without being checked. Thus, the interaction between both languages may
Value	not be correctly performed
Excessive Inter-language	eA wrong partitioning in both languages leads to many calls in a way or the other. It adds complexity takes more time
Communication	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
	When different libraries are needed depending on the operating system, they are not loaded with conditions on the
Hard Coding Libraries	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
Local Nelelelloes Abuse	references
Memory Management	Reference types passed from one language to another are not released in a language that does not handle the
Mismatch	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
0	A library is loaded using only the name not the path. It cannot be accessed in the same way from everywhere
Not Osing Nelative Fath	A whole object is passed as an argument, although only some of the fields were needed, and it would have been
Excessive Objects	better for the system performance to pass only these fields
Unused Method	better for the system performance to pass only these neits
Declaration	A method is declared in the host language but not implemented in the foreign language
Unused Method	A method is declared in the host language and implemented in the foreign language, but never called from the host
Implementation	language
Unused Parameters	
Unused Farameters	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).

- 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

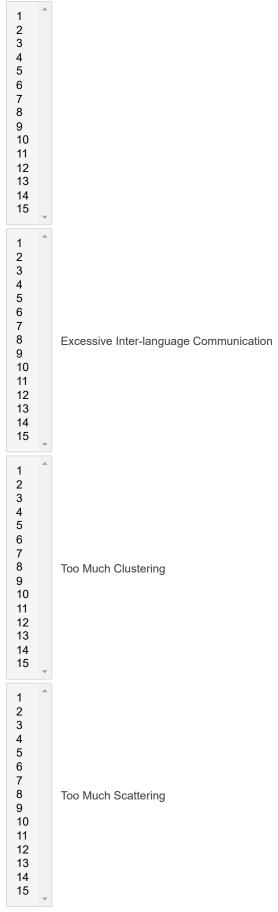
	1 Very Often	2 Often	3 Rarely	N/A
Not Handling Exceptions	0			0
Assuming Safe Return Value				
Excessive Inter-language Communication			0	
Too Much Clustering				
Too Much Scattering				
Hard Coding Libraries				
Local References Abuse				
Memory Management Mismatch				
Not Caching Objects				
Not Securing Libraries				
Not Using Relative Path				
Excessive Objects				
Unused Method Declaration			0	
Unused Method Implementation				
Unused Parameters			0	

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

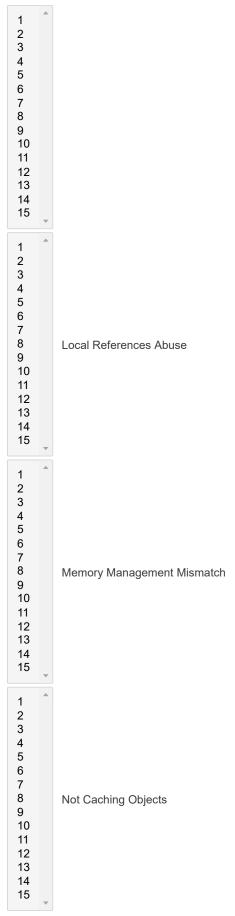
	Expandability	y Simplicity	Reusability	Learnability	Understandabilit	y Performance	Modularity	N/A
Not Handling Exceptions								
Assuming Safe Return Value								
Excessive Inter-language Communication								
Too Much Clustering								
Too Much Scattering								
Hard Coding Libraries								
Local 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

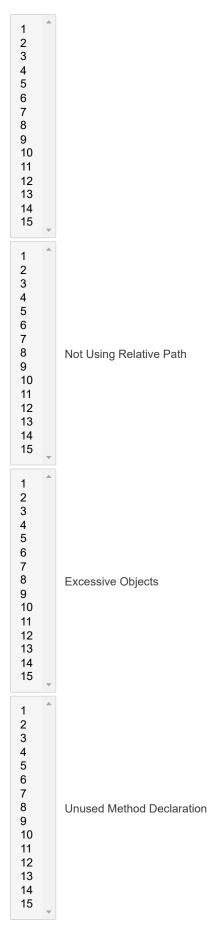
(Most harmful to the less harmful: 15 -> 1)



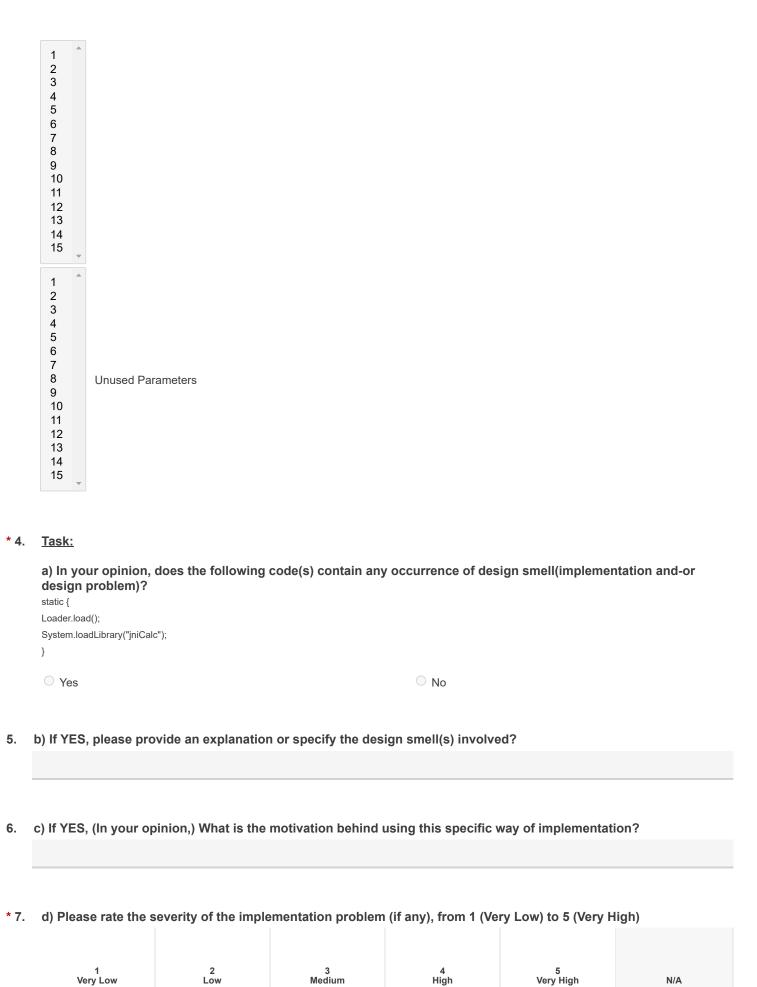
Hard Coding Libraries



Not Securing Libraries



Unused Method Implementation



N/A

publi station Acce publ	ic static void loadLibrar c {	u apply this refact  ded( new PrivilegedAction				
Syste	em.loadLibrary("jniCalc	");				
	Yes (Refactor with	hio colution)	O Yee	(Refactor with an alter	ro ativo polivijan)	
	No (No refactoring)	•	U 165	(Netactor with all alter	mative solution)	
Too	ale.					
Tas a) li		does the following	g code(s) contain any	v occurrence of de	sian smell(impleme	ntation and-or
des Strin	sign problem)?  In join(String separator,  In g string = "";		g 0000(0) 00111aiii uii.	, 00041101100 01 400		
for (S	String s : strings) {					
SITING						
}	g += (string.length() > 0	'? separator : "") + s;				
}	g += (string.length() > 0 rn string;	? separator : "") + s;				
} retur	n string;	? separator : "") + s;		○ No		
} retur }	n string;	? separator : "") + s;		○ No		
<pre>} retur } </pre>	n string; Yes		on or specify the des		ed?	
<pre>} retur }</pre>	n string; Yes		on or specify the des		ed?	
<pre>} retur } </pre>	n string; Yes		on or specify the des		ed?	
} retur }  o) If	rn string; Yes YES, please pro	vide an explanatio	on or specify the des	ign smell(s) involv		tion?
<pre>} retur } b) If '</pre>	rn string; Yes YES, please pro	vide an explanatio		ign smell(s) involv		tion?
<pre>} retur } b) If '</pre>	rn string; Yes YES, please pro	vide an explanatio		ign smell(s) involv		tion?
retur } o h	Yes YES, please pro	vide an explanation		ign smell(s) involv	way of implementat	
retur } original properties of the content of the c	Yes YES, please pro	vide an explanation	e motivation behind	ign smell(s) involv	way of implementat	
retur } original properties of the content of the c	Yes YES, please pro YES, (In your op	vide an explanation inion,) What is the everity of the imp	e motivation behind	ign smell(s) involves using this specific if any), from 1 (Ve	way of implementat ery Low) to 5 (Very F	High)
retur } o h	YES, please pro YES, (In your op	vide an explanation inion,) What is the everity of the imp	e motivation behind	ign smell(s) involv	way of implementat ery Low) to 5 (Very F	
retur } o h	Yes YES, please pro YES, (In your op	vide an explanation inion,) What is the everity of the imp	e motivation behind	ign smell(s) involves using this specific if any), from 1 (Ve	way of implementat ery Low) to 5 (Very F	High)

\* 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)

No (No refactoring)

Yes (Refactor with an alternative solution)
```

#### \* 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
                                                                                        O No
```

#### 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)



#### \* 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)
O No (No refactoring)
<u>Task:</u>
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 I);
}
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);
}

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

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

#### \* 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 I);
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. Task:

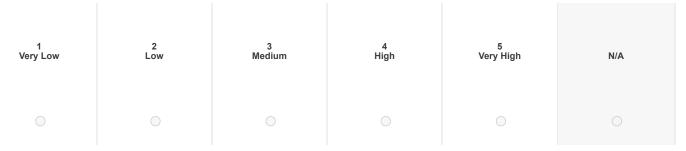
# a) In your opinion, does the following code(s) contain any occurrence of design smell(implementation and-or design problem)?

```
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;
```

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)



\* 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);
            || contact | contact |
```

	<pre>} System.gc(); Thread.sleep(100); lastPhysicalBytes = maxPh } trimMemory(); } catch (InterruptedException Thread.currentThread().inter } }</pre>	, -	rytes() : 0;			
	Yes (Refactor with No (No refactoring)	•	O Yes	(Refactor with an alterr	native solution)	
* 29.	design problem)?  public static native int mem  public static native Pointer  public static native Pointer  public  P put(Pointer p) {  if (p.limit > 0 && p.limit < p.position)	n("limit < position: (" + p.limit + " < " +	, long size); er src, long size); hter src, long size);	occurrence of des	ign smell(implemer	ntation and-or
	Yes			O No		
30.	b) If YES, please pro	ovide an explanation	n or specify the des	ign smell(s) involve	ed?	
31.	c) If YES, (In your op	pinion,) What is the	motivation behind	using this specific v	way of implementati	ion?
* 32.	d) Please rate the s	severity of the imple	ementation problem	(if any), from 1 (Ve	ry Low) to 5 (Very H	ligh)
	1 Very Low	2 Low	3 Medium	4 High	5 Very High	N/A

* 33.	e) If YES, would you public static native Pointer or public <p <p="" extends="" or="" pointer="" public="">if (p.limit &gt; 0 &amp;&amp; p.limit p.gize = p.sizeof(); int size = sizeof(); int psize = p.sizeof(); long length = psize * (p.limit position *= size; p.position *= psize; memcpy(this, p, length); position /= size; p.position /= psize; return (P)this; }</p>	nemcpy(Pointer dst, Pointer p) { P put(Pointer p) { vosition) { xception("limit < position: <= 0 ? 1 : p.limit - p.position:	ter src, long size); : (" + p.limit + " < " + p.positi tion);		ornativa polytian)	
	<ul><li>Yes (Refactor with the No (No refactoring))</li></ul>	his solution)	○ Ye	s (Refactor with an alt	ernative solution)	
* 34.	Task:  a) In your opinion, of design problem)?  String[] merge(String[] ss, S if (ss!= null && s!= null) {   ss = Arrays.copyOf(ss, ss.let ss[ss.length - 1] = s;   } else if (s!= null) {   ss = new String[] { s };   }   return ss: new String[0]; }	tring s) {	ງ code(s) contain ar		esign smell(implem	entation and-or
	O Yes			O No		
35. I	b) If YES, please pro	vide an explanatio	on or specify the de	sign smell(s) invol	ved?	
36. (	c) If YES, (In your op	inion,) What is the	e motivation behind	using this specific	c way of implement	ation?
<b>*</b> 37.	d) Please rate the s	everity of the impl	lementation proble	m (if any), from 1 (\	/ery Low) to 5 (Very	High)

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

#### \* 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)

No (No refactoring)

Yes (Refactor with an alternative solution)
```

## Your responses have been registered!

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