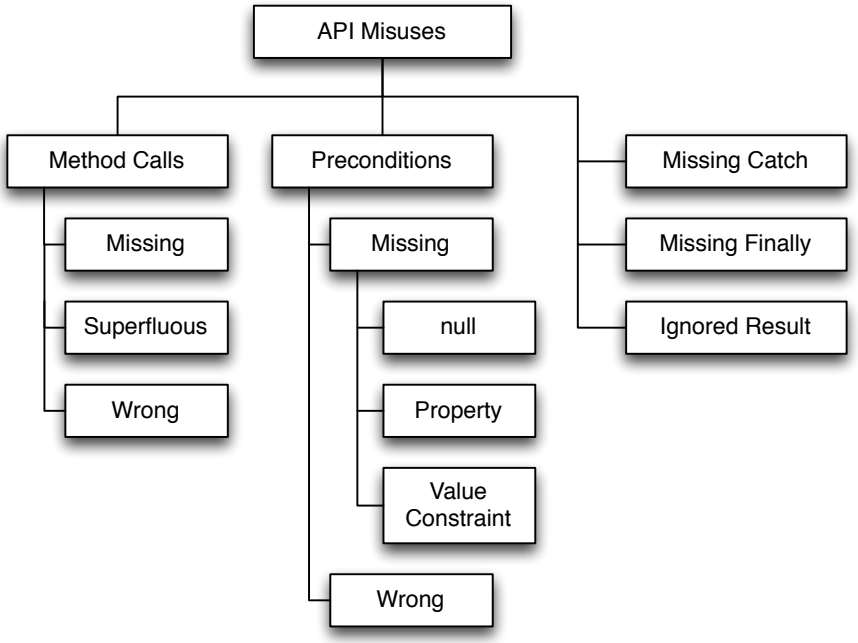

Figure 1: The Misuse Classification

1 The Misuse Classification

The Misuse Classification (MuC) is a taxonomy that categorizes API misuses. Figure 1 shows the classification hierarchy. In the following, we present and discuss the individual categories:

`missing call` describes a class of misuses where the developer missed to call a method that is required by the API to solve the task correctly. Examples from this class are missing to call `pack()` in an instance of `jFrame` to ensure its content are laid out and missing to call `close()` on a resource to prevent leaks.

`superfluous call` describes a class of misuses where the developer calls a method she should not call in this context. An examples from this class is accidentally calling a method twice on the same object, instead of once on that object and once on another object of the same type, as it is sometimes introduced as a consequence of copy and pasting statements.

`wrong call` describes a class of misuses where the developer calls a method with the right intention, but the wrong variant of it. Examples from this class are calling `File.mkdir()` instead of `File.mkdirs()`. We distinguish this from both a superfluous and a missing call, because of the difference in explaining the problem.

`missing precondition` describes a class of misuses where the developer failed to ensure a certain precondition for some (part of) a usage. We further differentiate three cases:

1. a `property` check on an object, e.g., `isEmpty()`,
2. a `null` check on a reference, before calling a method on it or passing it to a method that expects a non-`null` parameter, and
3. a `value constraint` check, e.g., that the number passed to `setFetchSize()` of a instance of `PreparedStatement` must be smaller or equal to the number passed to `setMaxRows()` on the same instance, if the number passed to `setMaxRows()` is larger than 0.

We distinguish the null checks, because the problem of forgetting them is so common that we detectors may well introduce special handling for this case, even if they do not generally capture precondition checks. We distinguish property checks from general value-constraint checks, because detectors that consider only information about method calls can detect missing property checks, but not value-constraint checks.

`wrong precondition` describes a class of misuses where the developer did a check with the right intention, but did a mistake in the boolean logic. An example from this class is missing to negate the result of `isEmpty()` before accessing a collection instance.

`missing catch` describes a class of misuses where the developer misses to handle a certain type of exception. Note that this includes the case where she does not handle any type of exception. Examples from this class are calls to `SortedMap.firstKey()`, which may throw the `RuntimeException` `NoSuchKeyException`, if the map is empty, and calls to `Cipher.init()`, which may throw an `InvalidKeyException`, if the passed public key is unusable in the given setting.

`missing finally` describes a class of misuses where the developer misses to enclose some part of a usage in a `finally` block to ensure its unconditional execution. Note that this includes cases where she does specifically handle exceptions. An examples from this class is calling the `close()` method of a resource outside a `finally` block.

`ignored result` describes a class of misuses where the developer misses to assign (or use) the return value of a method call. An example of this class is dropping the result of the pure method `BigInteger.setScale()`, which—despite its name—returns an adjusted copy of the current instance and leaves the current instance unchanged.