Testing on the Toilet Presents... ***Healthy Code on the Commode***

Make Interfaces Hard to Misuse

We all try to avoid errors in our code. But what about errors created by callers of your code? A good interface design can make it easy for callers to do the right thing, and hard for callers to do the wrong thing. **Don't push the responsibility of maintaining invariants required by your class on to its callers.**

Can you see the issues that can arise with this code?

```
class Vector {
  explicit Vector(int num_slots);  // Creates an empty vector with `num_slots` slots.

  int RemainingSlots() const;  // Returns the number of currently remaining slots.
  void AddSlots(int num_slots);  // Adds `num_slots` more slots to the vector.
  // Adds a new element at the end of the vector. Caller must ensure that RemainingSlots()
  // returns at least 1 before calling this, otherwise caller should call AddSlots().
  void Insert(int value);
}
```

If the caller forgets to call `AddSlots()`, **undefined behavior might be triggered** when `Insert()` is called. The interface pushes complexity onto the caller, exposing the caller to implementation details.

Since maintaining the slots is not relevant to the caller-visible behaviors of the class, don't expose them in the interface; **make it impossible to trigger undefined behavior** by adding slots as needed in `Insert()`.

```
class Vector {
  explicit Vector(int num_slots);

  // Adds a new element at the end of the vector. If necessary, allocates new slots
  // to ensure that there is enough storage for the new value.
  void Insert(int value);
}
```

Contracts enforced by the compiler are usually better than contracts enforced by runtime checks, or worse, documentation-only contracts that rely on callers to do the right thing.

Here are other examples that could signal that an interface is easy to misuse:

- Requiring callers to call an initialization function (alternative: expose factory methods that return your object fully initialized).
- Requiring callers to perform custom cleanup (alternative: use language-specific constructs that ensure automated cleanup when your object goes out of scope).
- Allowing code paths that create objects without required parameters (e.g. a user without an ID).
- Allowing parameters for which only some values are valid, especially if it is possible to use a more appropriate type (e.g. prefer `Duration timeout` instead of `int timeout_in_millis`).

It is not always practical to have a foolproof interface. **In certain cases, relying on static analysis or documentation is necessary** since some requirements are impossible to express in an interface (e.g. that a callback function needs to be thread-safe).

Don’t enforce what you don’t need to enforce - avoid code that is too defensive. For example, extensive validation of function parameters can increase complexity and reduce performance.

More information, discussion, and archives: `testing.googleblog.com`