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

Obsessed With Primitives?

Programming languages provide basic types, such as integers, strings, and maps, which are useful in many contexts. For example, a string can be used to hold everything from a person's name to a web page URL. However, **code that relies too heavily on basic types instead of custom abstractions can be hard to understand and maintain**.

Primitive obsession **is the overuse of basic ("primitive") types to represent higher-level concepts.** For example, this code uses basic types to represent shapes:

```
vector<pair<int, int>> polygon = ...
pair<pair<int, int>, pair<int, int>> bounding_box = GetBoundingBox(polygon);
int area = (bounding_box.second.first  - bounding_box.first.first) *
           (bounding_box.second.second - bounding_box.first.second);
```

`pair` is not the right level of abstraction because its generically-named `first` and `second` fields are used to represent *X* and *Y* in one case and *lower-left* (er, *upper-left*?) and *upper-right* (er, *lower-right*?) in the other. Worse, **basic types don't encapsulate domain-specific code** such as computing the bounding box and area.

Replacing basic types with higher-level abstractions results in clearer and better encapsulated code:

```
Polygon polygon = ...
int area = polygon.GetBoundingBox().GetArea();
```

Here are some other examples of primitive obsession:

- Related maps, lists, vectors, etc. that can be easily combined into a single collection by consolidating the values into a custom higher-level abstraction.

`map<UserId, string> id_to_name;`
`map<UserId, int> id_to_age;`
	`map<UserId, Person> id_to_person;`

- A vector or map with magic indices/keys, e.g. string values at indices/keys `0`, `1`, and `2` hold *name*, *address*, and *phone #*, respectively. Instead, consolidate these values into a higher-level abstraction.

`person_data[kName] = "Foo";`	`person.SetName("Foo");`

- A string that holds complex or structured text (e.g. a date). Instead, use a higher-level abstraction (e.g. `Date`) that provides self-documenting accessors (e.g. `GetMonth`) and guarantees correctness.

`string date = "01-02-03";`	`Date date(Month::Feb, Day(1), Year(2003));`

- An integer or floating point number that stores a time value, e.g. seconds. Instead, use a structured timestamp or duration type.

`int timeout_secs = 5;`	`Duration timeout = Seconds(5);`

It's possible for any type—from a lowly `int` to a sophisticated red-black tree—to be too primitive for the job. If you see code that uses a lot of basic types that would be clearer or better encapsulated by using a higher-level abstraction, refactor it or politely remind the author to *keep it classy*!

More information, discussion, and archives:

Copyright Google Inc. Licensed under a Creative Commons Attribution–ShareAlike 4.0 License (http://creativecommons.org/licenses/by-sa/4.0/).