Entropy Guidelines

Document #:	Draft 0.1
Date:	2019-04-07
Project:	Programming Language C++
Reply-to:	Jérémy Morosi <jeremymorosi@hotmail.fr>

1 Introduction

There must exist universal coding guidelines reflecting how nature works.

Entropy is a coding standard aimed to help you writing code easier to understand, maintain and unit test.

It works by defining a set of simple rules that encourage you to write code that tend to mimic how nature works. They emerged from the main idea that our code can be easier to understand when it reflects the fact that nature is composed of a multitude of independent systems each having little responsibilities rather than a single one system with a lot. Thus it can be seen as the single responsibility principle.

Of course we don't fully understand how nature works, but this is a k-approximation; and examples will prove that code written this way tend to be clearer. Ultimately it provides a way to measure entropy in your code and use it as a metric to identify and improve parts of your code that can be.

2 Motivation and Scope

These guidelines emerged from the fact that we often tend to write big monolithic codebase that we can hardly maintain and we can't remember/explain how it works days later. The goal is to prevent such things to happen by providing coding guidelines that - when applied correctly - help us to write code that mimic how nature works; thus producing a high amount of entropy.

But why entropy ? Think about the efforts required to keep a house clean. It naturally tend to be in disorder because this is simply how entropy works. Cleaning a house is like lowering the amount of entropy by reorganizing everything. Such process requires a lot of energy and it goes the same for your code.

If you are afraid of your code becoming obsolete or afraid of having someone else working on it because you know it's too complex for him to understand, it's because you are trying to maintain a codebase with too much functionalities, responsibilities or organization; which is equivalent to trying to keep it's amount of entropy low by spending a lot of efforts and energy.

Now, what would a codebase with a hight amount of entropy be like ? Functionalities would be written as separate and independent subprojects. Understanding how a functionality works and is written would require little effort and it would be easy to write unit tests covering as much code as possible. The best is that you wouldn't be afraid of having to rewrite your code if it became obsolete or broken.

This is what these guidelines are about. Not how writing high performance code in a specific language, but how organizing it to result into a high amount of entropy, requiring less efforts to maintain.

3 Core principle explained with `World` class

Imagine we are coding a multiplayer game and have a `World` class that contains all spawned entities stored by their id. Each entity must have an unique id and must be spawned both on server and client. One way to implement this class is as follow. Here is our `Entity` base class with an `enum` to identify the types of our entities over network:

```
class EntityType(IntEnum):
    NPC = 0

class Entity:
    def __init__(self, id):
        self.id = id

class NPC(Entity):
    def __init__(self, id, name):
        super().__init__(id)
        self.name = name

    def __str__(self):
        return "{}({})".format(self.name, self.id)
```

For the `World` class, as it is a multiplayer game, we have a `ClientWorld` class for client-side implementation and `ServerWorld` class for server-side implementation, but only the second one will be shown here:

```
class World:
    def __init__(self):
        self.entities = {}

class ServerWorld(World):
    GUID = 0

    def spawn(self, type, *args):
        entity = None
        if type == EntityType.NPC:
            entity = NPC(ServerWorld.GUID, *args)
        self.entities[ServerWorld.GUID] = entity
        ServerWorld.GUID += 1
        # Todo: send a packet to clients
        return entity
```

You can see that it simply instantiate a new entity based on the given type and register it to a map. If we were to test our `spawn` method it would produce the following output:

```
>>> world = ServerWorld()
>>> print(world.spawn(EntityType.NPC, "Spongebob"))
Spongebob(0)
>>> print(world.spawn(EntityType.NPC, "Spongebob"))
Spongebob(1)
```

It is working, but there is something definitely wrong with this code. The `ServerWorld` class has too much responsibilities for itself:

- Instantiating new entities

- Attributing free ids to them
- Keeping track of them
- Sending packets to spawn them on clients

All of this has been tied up in one place in a single effort to make everything work together. But such thing is against entropy and consequently will require a lot of effort to understand and maintain.

Also, how are we going to test these classes ? Should we write a single big unit test to check that spawning an entity on server would also spawn it on client with a unique id ? Or should we try to test each feature with four unit tests that will inevitably look redundant ?

Here follow guidelines that will answer these questions.

3.1 Removing the global `GUID`

First thing first, we will remove this global `GUID` variable. The `ServerWorld` shouldn't have to know how ids are attributed or care about it. Also, it force us to only have one single instance of this class because two different instances would share the variable:

```
>>> world1 = ServerWorld()
>>> print(world1.spawn(EntityType.NPC, "Spongebob"))
Spongebob(0)
>>> print(world1.spawn(EntityType.NPC, "Spongebob"))
Spongebob(1)
>>> world2 = ServerWorld()
>>> print(world2.spawn(EntityType.NPC, "Spongebob"))
Spongebob(2)
```

For us this look fine because we know there will only be one world in the whole program. But we shouldn't care about this limitation or force it on anyone else. More important, this is against entropy to force such things. So we are simply going to remove the `GUID` variable and pass a function returning new ids to our `ServerWorld` instead:

```
class ServerWorld(World):
    def __init__(self, get_free_id):
        super().__init__(self)
        self.get_free_id = get_free_id

    def spawn(self, type, *args):
        entity = None
        if type == EntityType.NPC:
            entity = NPC(self.get_free_id(), *args)
        self.entities[entity.id] = entity
        # Todo: send a packet to clients
        return entity
```

Now we can instantiate our world with any function or object that can return free ids which remove this responsibility from `ServerWorld` and just make everyhing simpler:

```
class IdAttributor:
    def __init__(self):
        self.next_id = 0

    def __call__(self):
        id = self.next_id
```

```
        self.next_id += 1
        return id

>>> world1 = ServerWorld(IdAttributor())
>>> print(world1.spawn(EntityType.NPC, "Spongebob"))
Spongebob(0)
>>> print(world1.spawn(EntityType.NPC, "Spongebob"))
Spongebob(1)
>>> world2 = ServerWorld(IdAttributor())
>>> print(world2.spawn(EntityType.NPC, "Spongebob"))
Spongebob(0)
```

We can now have as many world instances as we want and have total control over how ids are attributed without modifying the `ServerWorld` class. We disorganized our code a little which increased its entropy and made it easier to maintain. But we are not done yet.

3.2 Adding an `EntityFactory`

In our code we have an `enum` corresponding to the types of our entities:

```
class EntityType(IntEnum):
    NPC = 0
```

This is required for telling the client what type of entity must be spawned and to know how to instantiate it. But this is weird that the `ServerWorld` has to know about it as it just wan't to manage living entities and not to know how many different type there exist. Furthermore, there are too many ways our `ServerWorld` can become broken by modifying constructors of our entities and adding new values to the `enum`.

This is why we are going to rely on a separate factory to instantiate entities:

```
class ServerEntityFactory:
    def __init__(self, get_free_id, on_spawned):
        self.get_free_id = get_free_id
        self.on_spawned = on_spawned

    def spawn(self, type, *args):
        entity = None
        if type == EntityType.NPC:
            entity = NPC(self.get_free_id(), *args)
        # Todo: send a packet to clients
        self.on_spawned(entity)
        return entity
```

This is our factory server-side. You can see that it takes a function `get_free_id` that return the next free id and a callback `on_spawned` that is called when a new entity is instantiated. It will be used to automatically add spawned entities to our world. We will also remove the `ServerWorld` class and rewrite the `World` class as follow:

```
class World:
    def __init__(self):
        self.entities = {}

    def add(self, entity):
```

```
        self.entities[entity.id] = entity
        print("Entity {} added".format(entity.id))
```

To test this code, we will simply create a `World` instance and bind the `add` function of the world to the `on_spawned` callback of the factory:

```
>>> world = World()
>>> factory = ServerEntityFactory(
...     IdAttributor(),
...     lambda entity: world.add(entity)
... )
>>> print(factory.spawn(EntityType.NPC, "Spongebob"))
Entity 0 added
Spongebob(0)
```

Now, the world is simply a `World` whose only responsibility is managing living entities, not instantiating them, attributing free ids, or sending packets to clients.

3.3 Conclusion

Final version of the code:

```
class EntityType(IntEnum):
    NPC = 0

class Entity:
    def __init__(self, id):
        self.id = id

class NPC(Entity):
    def __init__(self, id, name):
        super().__init__(id)
        self.name = name

    def __str__(self):
        return "{}({})".format(self.name, self.id)

class IdAttributor:
    def __init__(self):
        self.next_id = 0

    def __call__(self):
        id = self.next_id
        self.next_id += 1
        return id

class ServerEntityFactory:
    def __init__(self, get_free_id, on_spawned):
        self.get_free_id = get_free_id
        self.on_spawned = on_spawned

    def spawn(self, type, *args):
```

```
        entity = None
        if type == EntityType.NPC:
            entity = NPC(self.get_free_id(), *args)
        # Todo: send a packet to clients
        self.on_spawned(entity)
        return entity

class World:
    def __init__(self):
        self.entities = {}

    def add(self, entity):
        self.entities[entity.id] = entity
        print("Entity {} added".format(entity.id))

>>> world = World()
>>> factory = ServerEntityFactory(
...     IdAttributor(),
...     lambda entity: world.add(entity)
... )
>>> print(factory.spawn(EntityType.NPC, "Spongebob"))
Entity 0 added
Spongebob(0)
```

We greatly increased entropy of our code by splitting the features into three different and totally independent classes. Advantages are that it's now easier to:

- Write lightweight and understandable unit tests
- Ensure that a feature is working
- Write documentation or explain our classes and functions
- Maintain or rewrite a feature
- Reuse our classes and functions

This is the kind of code you will write if you follow guidelines presented here. And the last point is important because `IdAttributor` is used here to attribute ids to entities but could now be used in any other context. It could also be in its own external library, including its unit test. The same goes for the `World` class that can now be used both on client and server with no distinction as it only require to know what an `entity` is, not how it is instantiated.

4 Entropy as a metrics

Now that we understand how entropy is related to coding, the next step is to find a way to measure how much entropy is produced by our code. Such metrics would help us to identify and improve parts of our code that reduce entropy. Why ? Because the more entropy is produced, the more our code is maintainable.

Fundamentally, we know that:

- Measuring entropy is equivalent to measuring how much our code is organized because spending energy into keeping something organized is equivalent to lowering its entropy.
- Maximum entropy is reached in a system at equilibrium

So, we are going to define a set of rules that describe how parts of the code can either be designed as organized or disorganized. Each rule will simply define a theorical maximum entropy reached when the code is fully disorganized (at equilibrium) or null if it is fully organized.

With this we will be able to compute the theorical maximum entropy our program can reach and improve it to tend toward this goal. And it will also be possible to identify parts of the code that are far from the equilibrium.

Each rule will also define a threshold code analysis tools can use to display a warning message.

5 Primitives

List of rules that depend on no other rules. They usually have a maximum entropy that can be configured by user depending on preferences.

5.1 Function has too many responsibilities

A function shouldn't have responsibilities beyond its scope.

Hints can be:

- A name stating `x_or_y`
- Too many calls to other functions
- Difficulty to explain how it works

Yes:

```
def get_user(login, password):
    return db.find_user(login, password)

def create_user(login, password):
    user = User(login, password)
    db.insert_user(user)
    return user

check_login(login)
check_password(password)
user = get_user(login, password)
if not user:
    user = create_user(login, password)
```

No:

```
def get_or_create_user(login, password):
    check_login(login)
    check_password(password)
    user = db.find_user(login, password)
    if not user:
        user = User(login, password)
        db.insert_user(user)
    return user
```

Measure:

```
'''
1 when n <= eps
0 when n >= max
'''
def compute(fun, max):
    n = count_calls_from(fun)
    return 1 - clamp(max / n, 0, 1)
```

Notes:

- Calls to lambda functions passed as arguments shouldn't be considered because they are expected to be called

5.2 Function parameters could be packed

Multiple parameters that are always passed together should be packed together.

Hints can be:

- Too many parameters in the signature
- Passing most of them from one function to another

Yes:

```
class Rect:
    def __init__(x, y, width, height):
        ...

def area(rect):
    return rect.width * rect.height

def compare(rect1, rect2):
    return compare(area(rect1), area(rect2))
```

No:

```
def area(w, h):
    return w * h

def compare(w1, h1, w2, h2):
    return compare(area(w1, h1), area(w2, h2))
```

Measure:

```
'''
1 when n <= eps
0 when n >= max
'''
def compute(fun, max):
    n = eps
    for call in get_calls_from(fun):
        n = max(n, count_same_args(call, fun))
     return 1 - clamp(max / n, 0, 1)
```

Note:

- This is in no way an indicator of functions having too many parameters

6 References