#OOP is dead, long live OOP
##Blog post summary
https://www.gamedev.net/blogs/entry/2265481-oop-is-dead-long-live-oop/

- OO … object oriented
- OOP … OO programming
- OOD … OO design
- ECS … Entity Component Systems

This post written by Hodgman is a reaction to a publication of Aras Pranckevičius where he shows some terrible OOP code and then the relation model as a great alternative solution.

Hodgman criticizes that it is not a fair comparison because the OOP code presented "violates all sorts of core OO rules". He fixes the OOD violations in the code and gets to a result with similar performance improvement as the "ECS"-code presented by Aras. Additionaly it uses also less RAM and requires less lines of code.

> Code that uses OOP language features, but does not follow OOD design rules **is not OO code**

###4 tools of OOP

Hodgman desribes the "4 tools of OOP":

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

These are powerful tools which can be used to solve problems but, he says, it is important to know when you should use them and not simply learn how to use them.

###Five core rules

- **Single responsibility principle**: if class *A* has two responsibilities, create class *B* and *C* to handle each of them and then compose *A* out of *B* and *C*.
- **Open/closed principle**: put parts likely to change into concrete classes and build interfaces around the parts unlikely to change
- **Liskov substitution principle**: any algorithm that works on a interface should continue to work for all implementations
- **Interface segregation principle**: Keep interfaces as small as possible
- **Dependency inversion principle**: instead of having implementations communicating

directly, decouple them by formalizing their communication interface as a third class.
- **Composite reuse principle**: Composition is the right default, inheritance should be reserved for use when it is absolutely required.

###Inheritance types
The interfaces' purpose is to hide the implementation. There are two types of inheritance:

- **interface inheritance**: abstract base classes with pure-virtual functions
- **implementation inheritance**: occurs any time a base classes contains anything besides pure-virtual functions. This should be treated as a bit of a code smell.

###Terrible OOP education

####About hierarchies/inheritance
Let's say you have a university app that contains a directory of Students and Staff. We can make a Person base class, and then a Student class and a Staff class that inherit from Person!

NO, NO, NO! If we don't know what kind of algorithms are going to be operating on *Students* and *Staff* then it's downright irresponsible to dive in and start designing class hierarchies.

> You have to know the algorithms **and** the data first

About inheritance type

Let's say you have a shape class. We could also have squares and rectangles as sub-classes. Should we have square is-a rectangle, or rectangle is-a square?

Good example to distinguish between implementation-inheritance and interface-inheritance.

#####Attempt 1: implementation-inheritance

```
struct Square { int width; };
struct Rectangle : Square { int height; };
```

By inheriting from square, the rectangle class (according to the LSP) must obey the rules of square's interface. **Any algorithm that works correctly with a square, must also work correctly with a rectangle.**

```
std::vector<Square*> shapes;
int area = 0;
for(auto s : shapes)
  area += s->width * s->width;
```

This will work correctly for squares (producing the sum of their areas), but will not work for rectangles. Therefore, Rectangle violates the Liskov substitution principle expained above.

#####Attempt 2: interface-inheritance

If you're using the interface-inheritance mindset, then neither Square or Rectangle will inherit from each other. The interface for a square and rectangle are actually different, and one is not a super-set of the other.

#####Solution: Implementation-inheritance
OOD says that composition is the right way to go!

```
struct Shape
{
  virtual int area() const = 0;
};

struct Square : public virtual Shape
{
  virtual int area() const {
    return width * width;
  };
  int width;
};

struct Rectangle : private Square, public virtual Shape
{
  virtual int area() const {
    return width * height;
  };
  int height;
};
```

> your OOP class told you what inheritance was. Your missing OOD class should have told you not to use it 99% of the time!

After this introduction Hodgman shows how Aras code can be improved with OOP. This code is re-implementing the existing language feature of composition as a runtime library instead of a language feature.

###Aras' code

This is an "Entity/Component" framework (sometimes confusingly called an "Entity/Component system") -- but completely different to an "Entity Component System" framework. It formalizes

several "EC" rules:

- the game will be built out of featureless "Entities" called `GameObject`, which themselves are composed out of `Component`.
- `GameObject` fulfill the *service locator pattern* -- they can be queried for a child component by type.
- Components know to which `GameObject` they belong to -- they can locate sibling components by querying their parent `GameObject`.
- Composition may only be one level deep (Components may not own child components, GameObjects may not own child GameObjects).
- A `GameObject` may only have one component of each type (some frameworks enforced this, others did not).
- Every component (probably) changes over time in some unspecified way - so the interface includes "virtual void Update".
- GameObjects belong to a scene, which can perform queries over all GameObjects (and thus also over all Components).

####Evaluation

1. `GameObject::GetComponent` uses dynamic_cast. This is a code smell and a indication for a violation of the Liskov substitution principle

2. `GameObject` is kind of ok if you imagine that it's fulfilling the service locator pattern… but going beyond OOD critique for a moment, this pattern creates implicit links between parts of the project

3. `Component` is a violation of the Single responsibility principle because its interface (`virtual void Update(time)`) is too broad. The use of "virtual void Update" is pervasive within game development.

4. Even though the goal of the `Component` class is to enable composition, it's doing so via inheritance, which is a Composite reuse principle violation.

###C++ code

Before: https://github.com/hodgman/dod-playground/blob/3529f232510c95f53112bbfff87df6bbc6aa1fae/source/game.cpp

After: https://github.com/hodgman/dod-playground/blob/f42290d0217d700dea2ed002f2f3b1dc45e8c27c/source/game.cpp

####Comparison

Instead of this:

```
// create regular objects that move
for (auto i = 0; i < kObjectCount; ++i)
{
    GameObject* go = new GameObject("object");

    // position it within world bounds
    PositionComponent* pos = new PositionComponent();
    pos->x = RandomFloat(bounds->xMin, bounds->xMax);
    pos->y = RandomFloat(bounds->yMin, bounds->yMax);
    go->AddComponent(pos);

    // setup a sprite for it (random sprite index from first 5),
    // and initial white color
    SpriteComponent* sprite = new SpriteComponent();
    sprite->colorR = 1.0f;
    sprite->colorG = 1.0f;
    sprite->colorB = 1.0f;
    sprite->spriteIndex = rand() % 5;
    sprite->scale = 1.0f;
    go->AddComponent(sprite);

    // make it move
    MoveComponent* move = new MoveComponent(0.5f, 0.7f);
    go->AddComponent(move);

    // make it avoid the bubble things
    AvoidComponent* avoid = new AvoidComponent();
    go->AddComponent(avoid);

    s_Objects.emplace_back(go);
}
```

We now have this:

```
struct RegularObject
{
    PositionComponent pos;
    SpriteComponent sprite;
    MoveComponent move;
    AvoidComponent avoid;

    RegularObject(const WorldBoundsComponent& bounds)
        : move(0.5f, 0.7f)
        // position it within world bounds,
    // pos(RandomFloat(bounds.xMin, bounds.xMax),
        //     RandomFloat(bounds.yMin, bounds.yMax))
        // setup a sprite for it
    // (random sprite index from first 5),
    // and initial white color
```

```
        , sprite(1.0f,
                 1.0f,
                 1.0f,
                 rand() % 5,
                 1.0f)
    {
    }
};

...

// create regular objects that move
regularObject.reserve(kObjectCount);
for (auto i = 0; i < kObjectCount; ++i)
    regularObject.emplace_back(bounds);
```

The original code has a main loop algorithm that consists of just:

```
// go through all objects
for (auto go : s_Objects)
{
    // Update all their components
    go->Update(time, deltaTime);
```

It's completely obfuscating both the flow of control and the flow of data within the game. Better:

```
// Update all positions
for (auto& go : s_game->regularObject)
{
    UpdatePosition(deltaTime, go, s_game->bounds.wb);
}
for (auto& go : s_game->avoidThis)
{
    UpdatePosition(deltaTime, go, s_game->bounds.wb);
}

// Resolve all collisions
for (auto& go : s_game->regularObject)
{
    ResolveCollisions(deltaTime, go, s_game->avoidThis);
}
```

The downside of this style is that for every single new object type that we add to the game, we have to add a few lines to our main loop.