Object-Orientation, Encapsulation, and Other Things That Are Wrong With Society

The great computer scientists of the past created entire operating systems without it. The great minds behind the moon landing did not need it either (Garry, 2014). Yet, today's so-called 'programmers' cannot even make a basic web app without it. Nowadays, object-orientation is lauded as the building blocks of computer science and software engineering, but the truth is, it is highly redundant, generally unnecessary, and sometimes actively harmful.

Primer: the Principles of Object-Orientation

Object-orientation is generally defined around 4 principles (Chandel, 2018):

1. **Encapsulation**. Encapsulation is concerned with the restricting of data, usually through the creation of getter-setter methods to manipulate fields.

2. **Abstraction**. Abstraction is concerned with the hiding of complexity, and in object-oriented programming, the division of units of meaning into classes.

3. **Inheritance**. Inheritance is concerned with expressing abstractions in terms of the relationships with each other, especially in terms of "parent-child" or "is-a" relations.

4. **Polymorphism**. Polymorphism is concerned with the morphing of various forms.

These principles are not inherently bad; however, object-orientation is almost never needed to achieve them, and the means to achieve these principles through object-orientation often overshadows the benefits of the principles to began with. Due to budgetary restrictions, this particular essay will only be examining encapsulation.

Problems of Encapsulation

Encapsulation, which advocates the hiding of data implementation for the purpose of abstraction, is perhaps the weakest link of object-oriented programming. This principle, in itself, is not bad, but encapsulation's manifestation in object-oriented code has usually resulted in code that is highly redundant at best, and actively embarrassing at worst.

Effective Java (Bloch, 2018, p. 78) states that "public classes should never expose mutable fields" and that "it is less harmful, though still questionable, for public classes to expose immutable fields". The book gives this example of a "degenerate" class:

```
class Point {
    public double x;
    public double y;
}
```

Effective Java considers this class to be "anathema," and proposes it be changed to the following in order to reflect object-oriented principles:

```
class Point {
    private double x;
    private double y;

    public double getX(){ return x; }
    public double getY(){ return y; }

    public void setX(double x){ this.x = x; }
    public void setY(double y){ this.y = y; }
}
```

Effective Java notes that, with the first example, "you can't change the representation without changing the API, you can't enforce invariants, and you can't take auxiliary action when a field is accessed". However, in the transformation of `Point` to a more object-oriented paradigm, none of these possible "benefits" of encapsulation are shown. In the second example, the API still must be changed if the representation is, no invariants are enforced, and no auxiliary action is taken for field accesses! And there is no reason why any of those would be needed for the example given. And so, here, *there is absolutely no benefit in using encapsulation*. The author is merely conforming to object-oriented ideology for the sake of conformity.

Not only is this juvenile principle-based masturbation useless, it can be argued that it is even harmful. The first example is obviously clearer and cleaner than the second, and the resulting boilerplate from the transformation is highly wasteful. Some languages do make some effort to reduce this boilerplate, such as in C#:

```
class Point {
    public double x { get; set; }
    public double y { get; set; }
}
```

Here, with this special declaration, C# labels `x` and `y` not as regular fields, but as “properties”, and in doing so, C# allows them to be treated as fields. But for what? The above example will function *exactly the same* if these properties *simply are fields*:

```
class Point {
    public double x;
    public double y;
}
```

It is clear the defensiveness of this irrational complexity has only come from psychological repression of programming language designers. The ego correctly desires an escape from this madness, but the superego merely moderates this obsession with encapsulation into what may be called encapsulation-without-encapsulation, as seen above, whereby designers suffering from cognitive dissonance attempt to avoid forced encapsulation as far as one can while still operating within it as framework.

Encapsulation as protection

Encapsulation can be a positive good when it actually fits the idea of encapsulation, abstracting away excessive detail or offering data protection. Classes will indeed sometimes act as more than structures or records.

However, it is not always clear that protections absolutely need to be added to fields that should not be mutated directly. The vast majority of classes are only used internally in projects. Is there anything more self-depreciating than trying to protect yourself from a mistake that you know you would not ever make?

Even when exported, these kinds of protections are not necessary, since *improper use is unlikely to happen when there are no reasons to improperly use a field* and, in most cases, *can simply be prevented with proper documentation.* Programmers need not to assume that those that use their structures are monkeys typing away, mutating fields at random to cause undefined behavior.

Consider Java’s `ArrayList` class. Internally, `ArrayList` uses a private integer called `size` to keep track of the number of elements in the list. What would be the difference if this field was public? If public, `size` would then be mutable, which if mutated, could lead to undefined behavior. Yet, who would mutate it? There is no obvious reason to do so outside of the methods specifically designated to. It would defy all reasoning to do so.

When it comes to the use of mouse traps, it is obvious that mouse traps should only be used for certain obvious purposes, like trapping rats, as well as creating Rube Goldberg machines. When used outside of those purposes, it can be said that the use of a mouse trap may result in undefined behavior, such as pinching one's nose. Yet, it is obvious to all that one should not put their nose in a mouse trap, or undesired behavior may occur. Thus, this rarely happens, unless causing undefined behavior is intentional, as may be in a YouTube reaction video.

Perhaps instead of obsessing over authoritarian methods to create state safety, programmers should simply write less complicated code.

Achievement of Encapsulation Outside of Object-Orientation

Regardless of how one feels about all of the aspects of encapsulation, it is worthwhile to note that it can be achieved outside of object-orientation. Consider C, a notably non-object oriented language.

C allows for methods and variables inside of a file to be declared as *static*, which disallows files outside of it to access and use them. Individual members of structures cannot be declared as static; however, C does have “opaque data types”, which allow for some level of abstraction. Essentially, the structure is declared in the header file, and defined in the source file. Thus, when working with such as a header file, the implementation details of a structures are not a concern.

Below is a rough example of how an array list may be implemented in C:

```c
// array_list.h
// Note that "ArrayList" is what is used directly.
// _ArrayList is used internally to match the declaration with the definition.
typedef struct _ArrayList ArrayList;

ArrayList* ArrayList_create();
void ArrayList_delete(ArrayList* list);
int ArrayList_get(ArrayList *list, int index);
void ArrayList_add(ArrayList *list, int val);
int ArrayList_size(ArrayList *list);


// array_list.c
// This is the real definition of the structure
typedef struct _ArrayList {
    int size;
    int capacity;
    int **data;
} ArrayList;

// Used only locally to grow the capacity of the ArrayList.
static void grow(ArrayList *list);

ArrayList* ArrayList_create(){
    // Implementation redacted.
}

void ArrayList_delete(ArrayList* list){
    // Implementation redacted.
}

int ArrayList_get(ArrayList *list, int index){
    // Implementation simplified.
    return list->data[index];
}

void ArrayList_add(ArrayList *list, int val){
    // Implementation redacted.
}

static void grow(ArrayList *list){
    // Implementation redacted.
}

int ArrayList_size(ArrayList *list){
    return list->size;
}
```

The user of an array list would interface with it through including (importing) `array_list.h`, and thus could not access any fields of the `ArrayList` structure directly, nor the `grow` method associated with it, which would be used internally by the `ArrayList_add` method to increase the capacity of the array list.

Conclusion

Encapsulation in the object-oriented context results in ridiculous, unnecessary code, which becomes harder to read and understand. Encapsulation may not be necessary to begin with, especially in internally-facing code, and regardless, object-orientation is not needed to implement encapsulation. Needless to say, object-oriented programming in its present form is the opium of the people, and as a society, we can and should do better.

References

1. Garry, C. L. (2014, April 3). Apollo-11. Retrieved April 11, 2021, from https://github.com/chrislgarry/Apollo-11

2. Chandel, M. (2018, August 15). What are four basic principles of Object Oriented Programming? Retrieved April 11, 2021, from https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

3. Bloch, J. (2018). Effective Java (3rd ed.). Boston, MA: Addison-Wesley. Retrieved April 11, 2021, from https://www.kea.nu/files/textbooks/new/Effective%20Java%20%282017%2C%20Addison-Wesley%29.pdf