Three levels of software

Give another example of a defect of modular reasoning ("level 3") which does not result in a code ("level 2") or runtime ("level 1**")** failure. Make sure your answer contains (1) a program or description of a program, (2) an explanation of why the program cannot be shown correct by modular reasoning(e.g.: function precondition not being met, a subroutine's guarantees are not strong enough to ensure desired behavior, etc), and (3) an explanation of why there is no concrete input or scenario that would exhibit a defect, but how a small future change could cause a sudden failure.

Answer

Here is a collection of answers given by earlier students that we like. These answers do not explicitly contain the three things asked for in the question, as the question's wording was changed in August 2022, although enough information is present to reconstruct them.

Jonathan Ray

> Bob calls an API and relies on the coincidence that it always returns a sorted list, even though the API does not promise to always return a sorted list.

Avery Pelle

> An example of a defect in modular reasoning that I have seen is reliance on the default ordering of rows in SQL (ex MySQL). For example, a program wants to retrieve a user's first 10 transactions in a system. The programmer then writes a SQL query to `select * from transaction limit 10` . However, without specifying an `order by` clause, this is a defect in modular reasoning. MySQL does not guarantee the order of rows in a database unless `order by` is specified, although many database engines will sort by insertion order and many programmers will never know that this circumstantial ordering is not guaranteed by the SQL spec because it will not produce unexpected behavior in many major DB engines.

The Design of Software is a Thing Apart

Give another example of when two different (or partially different) design intentions result in the same code, so that, if one changes, the other should likely not be changed. I.e.: a case where duplicated code is actually a good thing. Make sure your answer contains (1) a code snippet or a precise description of code, and (2) two alternative specifications for this code snippet, informally stated.

Answer

We've collected a large number of answers here: https://docs.google.com/document/d/1DvYcmeuadYEHY9uw8ZWpdXmem_pykgJUDDaCkd_VGJs/edit

Below are two example answers we like:

There is a profile page within a modal with two buttons, `back` and `close`. Both of them have click handlers that look like

```
<button onclick="window.history.back();">Go Back</button>
<button onclick="window.history.back();">Close</button>
```

These buttons could share a click handler to avoid duplication in the code. However, this wouldn't be prudent from a design perspective. In the future, the "close" button may want to check if there are unsaved changes on the page before closing the modal. However, since "go back" implies reverting to a previous state, this requirement might not apply to "go back", and thus the functionality would be divergent

Here is an example of what is asked for

```
# used by the manager to verify order details with the customer
def customer_phone(order)
    order.customer.phone
end

# used by the courier to arrange delivery
def delivery_contact_phone(order)
    order.customer.phone
end
```

While these two functions have exactly the same code now, in the future we might add a special column to orders table to store the contact phone that may be different from the customer's phone:

```
# used by the courier to arrange delivery
def delivery_contact_phone(order)
    order.delivery.contact_phone
end
```

Modules Matter Most

Pay particular attention to where he talks about the many-to-many relation between implementation and interface. This is another statement of the idea given in "The Design of Software is a Thing Apart." How would you express the idea of multiple interfaces for the same thing in the languages you use?

Answer

Languages have various levels of support for this. An incomplete list of options follows. Note that most of these only support having an "internal" interface that exposes everything and one or more "external" ones that expose some things. This covers the common case, but Bob Harper is speaking of the ability to give an arbitrary set of interfaces to a module, possibly with none dominating another in openness. Also, most of these require the module author to specify the set of allowed interfaces ahead of time. Further, most of these examples will actually be about objects/classes, where the ability to give multiple

interfaces only applies to one type at a time, as opposed to giving a different interface to a bundle of related types (like Regex and MatchData, or File and FileReader/FileWriter), .

Multiple interfaces, no coercion

Languages with explicit nominal interface implementation support restricting the set of allowed operations on an object by casting it to a weaker type. Languages with multiple inheritance do as well, to a lesser extent.

For example, the Integer type in Java can be passed around under many restricted interfaces such as Comparable and Serializable, in which most of its operations (such as addition) are unavailable.

Multiple interfaces with coercion

With this technique, one instead writes a function from an object to a new object with a restricted interface. This allows an object to be given an unanticipated interface. This commonly takes the form of the decorator pattern.

```
class ReadOnlyList<E> {
  private List<E> base;

  public ReadOnlyList(List<E> base) { this.base = base; }

  public E get(int i) { return base.get(i); }
}
```

This example is similar to Java's standard `Collections.unmodifiableList()` method, whose type signature returns a `java.util.List`, but which implicitly returns an object with a restricted interface.

Overlapping closed scopes

Consider this JavaScript:

```
function makeThing() {
  let privateVar = 0;
  let publicVar = 0;

  let privateState = {getPrivate: () => privateVar, getPublic() =>
  ↪ publicVar, setPrivate => …}

  let publicState = {getPublic() => publicVar};

  return [privateState, publicState];
}
```

Here is a Java analogue:

```
class PrivateThing {
   private int privateVar;
   private int publicVar;

   public int getPrivateVar() { return this.privateVar; }
   public int getPublicVar() { return this.publicVar; }
   class PublicThing {
     public int getPublicVar() { return PrivateThing.this.publicVar; }
   }
}
```

Re-exporting

In languages such as TypeScript and Haskell in which one has individual export control on every declaration in a file, one can build an implementation in the file MyModule.Internal, and then re-export only a public subset in the `MyModule` file.

Famed Haskell programmer Edward Kmett explains the use of `.Internal` modules in Haskell: https://www.reddit.com/r/haskell/comments/2nkiiq/testing_internals_without_exposing_them/cmfph90/

This approach is quite flexible, and does allow many different overlapping interfaces to be given to the same module. This approach also allows redefining a function at a more specific type (e.g.: re-exporting a function

```
map :: (a -> b) ->  AnyDataStructure a -> AnyDataStructure b
```

as

```
map :: (a -> b) -> List a -> List b)
```

However, without a true module system, one cannot then provide two files that export types and functions with the same names/signature and easily switch from using one to using another, such as switching between the stable sequential and experimental concurrent versions of the same feature. (Sometimes `#ifdef's` are used as a hack to do this.) It also does not allow for changing the opacity on types.

True module system

Some members of the ML family of languages (particularly OCaml and SML) support separate modules and module types, where one module can be ascribed to multiple signatures. You can think of this as being able to give multiple settings to the export visibility and type opacity of everything in a file. This is strictly better than any of the previous options because this control can apply to an entire

As of April 2022, there is an entry in the Hot Takes document (https://docs.google.com/document/d/11AF0ZXtALKoP6Q9513jkd0bIR4HcrMs0y7TPrzH5FNY/edit) on this reading, showcasing some OCaml code that can export a bundle of multiple types and functions at different interfaces.