Assignment: The Three Levels Of Software

Here are some interesting examples of errors of modular reasoning from previous students, with some near misses at the end

Question:
Give another example of a defect of modular reasoning ("level 3") that 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.

Good Examples

Hashes not guaranteed to be consistent across executions

C++ has a feature called Run Time Type Information (RTTI), which provides information for types (both primitive and user defined).
To get the information for a type you use the typeid keyword, which returns a reference to a type_info struct. This struct
has a hash_code member function that returns the hash for the type. You can get a hash like this:

```
auto int_hash = typeid(int).hash_code();
```

I've seen people use the hash information when serializing and de-serializing objects between executions of a program.
The serialization code might look something like this:

```
template<typename T>
ostream& serialize(ostream& os, const T& x) {
    return os << typeid(T).hash_code() << " " << x << '\n';
}
```

The de-serialization code might look like this:

```
template<typename T>
T deserialize(istream& is) {
    T local_t;
    std::size_t t_hash;
    is >> t_hash;
    if (typeid(T).hash_code() != t_hash)
        throw ...
    is >> local_t;
    return local_t;
}
```

This code contains an error of modular reasoning. The reason is that C++ standard only guarantees that the hashes for a type will be consistent within a single invocation of the program. This usage of hash_code() expects more than the function guarantees, specifically that it returns a value that will be unique for the type ACROSS program executions. What's worse, is that the C++ runtime doesn't randomize this across executions, so this code might be fine for years!!! However, as soon as compiler vendors decide to make a change to the RTTI system, the code could suddenly stop working and you've now got a hard-to-fix bug.

Non-guaranteed non-blocking

1)
```
class BusinessProcessor {
    def process(): Result = { /* .. */ }
}

// ...

def serializableProcess(processor: BusinessProcessor): Result = {
    globalLock.lock()
    try {
        processor.process()
    } finally {
        globalLock.unlock()
    }
}
```

2) One of the important preconditions for the function is that `BusinessProcessor.process` should not block for an unspecified amount of time.

The violation of this precondition might lead to:
* Performance degradation
* Livelocks and deadlocks

3) If BusinessProcessor is a concrete class that at the moment doesn't do any blocking calls in process method, then `serializableProcess` will be working fine.
One thing that I find interesting: if the BusinessProcessor was a trait(an interface) - then the potential problems would be more visible as interface implies potentiality for unrestricted (in this case) new implementations.

Comment

This highlights the importance of specifying this kind of expectations on code behaviour.

Undocumented assumption of non-interference

This method is a simple iterator over the input object (passed by the reference).

Pseudocode -
```
// here input_list is passed by the reference
def list_iterator(input_list):
    n = len(input_list)
    for i in range(n):
        do_something(input_list[i])
```

The defect in the modular reasoning is the assumption that input parameter stays unchanged throughout the execution of the method (or the fact that the object is passed by the reference from outside and there is possibility that it could change during the execution is not considered in the logical design level). If the referenced object is not modified by other methods, there will be no code or runtime failures.
If we consider only the code, and consider certain inputs, the defect will not be exhibited, because as we walkthrough this code, the assumption is that input stays unchanged. But if in the future, the input_list is modified while this method is running, there is a possibility of exceptions, such as out of bounds array access.
To fix this, in the "level 3" precondition/predicate, it should be stated that the input_list must remain unmodified until the method completes.

Global assumption about frequency of a call

Consider a browser based weather app.

This weather app has a state container that splits off each category of global state into separate stores, such as a user's account details which also stores the list of locations they want to track, as well as a separate store that stores all queried location weather data. Currently, the code is implemented such that anytime the account details store is updated that the weather API is called to get new data.

This implementation currently works because the Account store only gets updated whenever the user was changed (e.g. login) or if their tracked locations changed. In the future, a new developer wanted to add a logout timer and store this information in the Account details store. They decided to add and update this timer every second in the account details store. The spec hasn't changed, but since the Account store is updated more frequently, and the app is now overusing our API data limits - essentially a bug because the component that was calling this API was making the assumption that the Account store would update on direct user inputs or logins/logouts.

Lazy effects

Consider the following api in Clojure

```clj
;; a business logic function
(defn revoke-user-permissions
 [db user-ids]
 (map (fn [user-id] (clojure.java.jdbc/delete! :UserPermissions ["userId = " user-id])) user-ids))

;; called when the endpoint  POST /users/revoke is hit
(defn revoke-user-permissions-endpoint-handler
 [request services]
 (let [user-ids (:user-ids request)
       db (:database services)]
   (revoke-user-permissions db user-ids)))
```

The postcondition for `revoke-user-permissions` is intended to be that there does not exist any row in the UserPermissions table with `userId = u` for any u in the parameter `user-ids`.
But, this postcondition isn't actually satisfied because in Clojure `map` is lazy and will only do work when forced. Whether `revoke-user-permissions` does what you want

depends on whether you wind up forcing the result at the call site or not. In the code above, the endpoint will always work as intended, because `revoke-user-permissions` is forced by returning it from the handler function. Ultimately, something trying to convert the list to json will iterate through it, forcing the thunk. However, making a change like below will break the endpoint:

```clj
(defn revoke-user-permissions-endpoint-handler
 [request services]
 (let [user-ids (:user-ids request)
       db (:database services)]
   (revoke-user-permissions db user-ids)
   (make-successful-response user-ids))
```

Because now `map` isn't forced to run.

Ignoring the specification

I create a predictor function that will tell a user if a Cake (which is defined in an external Cake library) with certain ingredients will be a fan favorite:

Here is the Cake object that is defined in an external Cake library:

```
type struct Cake {
    sugarAmount: number;
    butterAmount: number;
    carrot: {
        shapedLikeCarrot: boolean;
    };
}
```

Here is my function:

```
function willFansLikeThisCake(cake: Cake) {
   if (cake.carrot.shapedLikeCarrot) return true;
   if (sugarAmount > 50) return true;
   if (butterAmount > 50) return true;

   return false
}
```

My design spec holds the assumption that willFansLikeThisCake should work for all Cakes.

However, the Cake library spec states that, due to changing consumer tastes and lack of development resources, maintaining specific cake attributes within the Cake object is not guaranteed for future versions of the cake library, and that we should guard against these changes wherever possible.

This function works, up until whatever version in the future where cake.carrot is removed. This breaks the code implementation as I am now continuing to access the property shapedLikeCarrot on an undefined object.

Implementation looser than the spec

The public API design states input X must be of type number.

The implementation just checks that X can be resolved to a number, so strings like "34" , and "i45" all work in the current implementation.

In a future version this could be rewritten much more strictly to just accept integer types and all clients relying on the previous implementation will start throwing errors.

Comment

Hyrum's law states that any observable behavior of an API will become depended on by someone. This highlights why the second half of Postel's law (be generous about what you accept) is setting things up for disaster. To retain sanity, the implementation should aim to brutally awaken non-spec-compliant users before their bad habits become entrenched.

Exploiting undefined behavior

Using unions for type punning in C++ violates the C++ spec but is a commonly-used technique for eg reinterpreting the bits of a 32-bit float as a uint32_t for serialization. Some compilers implement non-standard extensions for them.

This:

```
union int_float {
  uint32_t i;
  float f;
};

uint32_t f() {
  union int_float val;
  val.f = 3.0f;
  return val.i;
```

}

is explicitly supported by GCC and happens to work in many other compilers. See https://stackoverflow.com/questions/25664848/unions-and-type-punning. People in the "Standard C" camp discussed here would certainly consider this a failure of modular reasoning, and in the future, compilers might decide to compile code involving type punning differently while still following the C++ spec. More generally, undefined behavior is an interesting case study in people disagreeing about the design to consider. Many high performance application developers working to develop code for specific platforms would prefer to program in semi-portable C/C++ where most "undefined behavior" is instead implementation-defined for the platforms they are targeting. However, out of necessity, they tend to use compilers developed by compiler authors whose objective is to take maximum advantage of official C/C++ specs to emit the most optimal machine code. In practice, this has led to security problems resulting from safety checks removed by compilers exploiting undefined behavior.

Looking beyond the indirection

This just makes me think of all the code out there that relies on specific, not technically guaranteed or documented API behaviors that could easily break on minor change updates. An obvious example of this I came across recently is a web API who's DNS happens to resolve a set of arbitrary domains to the same server. A client module could rely on a specific, technically not supported host. As long as the DNS config never changes, the client will always work.

Comment

A more obvious variation might be the assumption that two of those arbitrary domains resolve to the same server, although any assumption about the particular location provided by a DNS lookup is an error of modular reasoning. (Obviously, your assumption should still be that you will reach a functioning instance of that service)

Assuming constant environment

Working on an app where the spec requires notifying users to continue using the app at 9am every wednesday. (let's imagine the server, developers and users are in the same location.)

the developer in charge of the projects write a cron job to run at 9am every wednesday "0 9 * * WED"

the code will work fine, with all of the tests because the company is still a startup with <100 users all in the same location and every attempt to test this feature will all pass.

the problem will arise when a user has to move out of the current timezone, or if the server location is changed to reduce cost or the company grows and acquires more users in other timezones.

Race condition

My example is autosave feature combined with a load next level feature.
Something like:

```
auto_save()
load_next_level()
```

Where the `load_next_level` routine loads the next level after the currently saved one.
Since these two pieces of code are separate `load_next_level` implicitly depends on `auto_save` completing before `load_next_level` runs.
Put in Hoare terms:

```
{true}
auto_save()
{auto_save called}
{auto_save complete}
load_next_level()
{level loaded}
```

Written with Hoare triples the issue is obvious,
but for every runtime value and possibly every test system this runs on,
that suble breach of condition may never surface.

But with an overloaded database or slower computer,
`auto_save` may not be complete by the time `load_next_level` is called,
creating a race condition bug with no change to the input data or code.

(This is loosely based on a real bug I found at work.
That code worked perfectly fine on local test machines,
and sometimes, but not always, created issues in production.)

Assuming consistent order

Recently I used a REST API which accepts an array of ids and returns a json array of objects corresponding to those ids. I ran a sample request and the returned array had the same order as the requested array. My initial idea was to use this property to zip the input and output arrays together. However, this would be an error of modular reasoning, because the order of returned items was not specified, and therefore could change in the future. The code would currently work correctly for any input, but was still wrong on the logical level.

What, this can error?

1. An early implementation of a MailService uses a Java executor with a threadpool to process jobs asynchronously. It worked fine for many years, while the system didn't need to send more than 32 emails at the same time.

```
MailService.java

public class MailService {

        protected static ThreadPoolExecutor executorService = new ThreadPoolExecutor(
        1, // keep at least one thread in the pool
        32, // but no more than 32, which actually seems excessive to me
        5L, TimeUnit.MINUTES, // reclaim threads if they have been idle for more than 5 minutes
        new SynchronousQueue<Runnable>());
        );

        public void sendEmail(MailMessage message) {
         ...
         executorService.execute(new MailSender(api, message));
        }

        ...
        ...

        static class MailSender implements Runnable {
         MailSender(API api, MailMessage message) {
         ...
         }

        @Override
        public void run() {
                api.send(message);
```

```
        }
        }
}
```

2. However, the implicit guarantee of enough threads for sending emails is not strong enough in this implementation. The service only provides up to 32 threads, and `SynchronousQueue` data structures do not actually queue incoming jobs, but just hand over to available threads. When there are no more thread available, an `RejectedExecutionException` exception was thrown.

3. At the early stage of the application, the system rarely send emails to users. So inputs and scenarios in development, testing, and even production never revealed the code issue above with `SynchronousQueue`. However, after a few year, the application grew a lot more popular with a lot users, and new features of email reminders were added later that require sending many emails in a short period of time (sometimes more than 32 emails at the same time). Here the code that used to work broke after more than 10 years in production.

With a little help from my friends

Suppose you are trying to query an API, and the spec for the API says that you should limit yourself to a certain number of requests per second. Now, as it turns out, this API is new enough that the devs hadn't bothered *enforcing* rate limiting strictly. And suppose your API querying module depends on another module that has the effect of doing rate limiting, but in a way that is not guaranteed by the spec for that module.

In this scenario, your API querying code "cannot be shown correct by modular reasoning," since" that other module that happens to be implementing rate limiting is *not* doing so in a way that is *guranteed* by the spec (i.e., is not one of the module's post conditions), and since for the logic of your API querying app to be correct, it does need to respect, at the design / logical level, the requirement that there be some sort of rate limiting.

There is no concrete user input or scenario that would exhibit a defect, given the way the API's code and your app is currently set up: even if your app were to suddenly explode in popularity, that other module that is currently in effect implementing rate limiting would throttle things.

But if one day (i) that other module is changed so that it no longer has the unintended effect of implementing rate limiting and (ii) the API's implementors were to actually enforce rate limiting, your API querying code would obviously break.

This is a somewhat contrived example, but so it goes.

Near misses

Happy path coding

Consider a Node.js program that is meant to copy files to cloud storage via HTTP POST. The program runs in an environment that has credentials allowing write access to the cloud storage destination.

```
const xferToCloudStorage = (metadata, stream)=>storageFileLibrary.write(metadata, stream);

...
...
return xferToCloudStorage.then(respondAndCleanup).catch(handleError);
```

This works for all inputs tested. There is error checking on the metadata format, there is a rejected promise handler and retry is built into the library. It's working in production for months without issue.

Nobody remembers this thing is running because it does it's job so well.

Then the credential expires. The person responsible was away for two months and missed all email warnings. There was no group alerting in place. There was no automated monitoring. The program fails.

What's worse, the storageFileLibrary logs an error during credential failures, but it doesn't reject the promise, it's left unfulfilled due to a recent patch level change. The library documentation indicates that the function returns a promise, which it still does, but the documentation doesn't specify that the promise is always resolved. Technically the new patch is still working as designed. Practically, it's a nightmare.

Responses aren't provided, there's no cleanup done. Connections pile up, and thanks to auto-scaling, there's six months worth of compute time billed during the weekend of the failure and the half day it takes before anyone notices the system isn't working as expected.

During post mortem discussion, someone asks why tests didn't catch the issue. They're told that all of the inputs were valid and there was 100% code coverage. There was no input that could cause a test failure because the failure was in a third party library. Someone walks away muttering something about modular reasoning, saying that the library's guarantees weren't as strong as they had thought, making it impossible to show that the program was correct. Had the library made some specific claims about the resolution of the promise, the

program could have been shown to be correct (or not).’

Comment

Certainly there are many things that went wrong here, with pretty bad results, but it is not necessarily an error to focus on the happy path. It can be a very costly and never-ending rabbit-hole to try to cover every contingency. Often it is more sensible to do a risk analysis and make an estimate that you will be able to find out and fix problems before they cost too much.

The credentials expiring would have been something they should have thought about. Presumably it was felt that the email warnings would suffice, so there is no error there.

What about the third party library? It seems obvious that a precondition would be supplying correct credentials and when a precondition is not fulfilled, all bets are off, so no error of modular reasoning there either.

You might think you could reasonably expect returned promises to always be resolved, but that is not true. Carl Hewitt, inventor of the actor model, cautions us to always consider indeterminism. Even if you are 100% certain something will happen, it may take infinitely long before it does (where infinity is any time longer than you are prepared to wait). And of course in this case all bets were off anyway since the preconditions were not fulfilled.

So no error of modular reasoning here, just a risk assessment gone wrong.

Doesn’t cater to all inputs

Mac was asked to write a program to export users annual account balances as CSV files. The annual account balances are stored as list of tuples, e.g.

```
[(2021, 1000), (2022, 1100)]
```

Mac figures they might need to export other CSV files at some point, so he creates this clever abstraction:

```
def export_csv(data, filename):
  with open(filename, "w+") as f:
    for row in data:
      f.write(",".join(str(cell) for cell in row))
```

```

This code works great for the current use case, and would even work for a lot of other CSV data. However, this function won't work correctly if a cell contains a comma. The function name implies the function will export a valid CSV for any given data, not just data that doesn't contain any commas.

### Comment

This is rather an error on the level of code since it doesn’t work for all inputs it seems to handle. However, if the program was written so that this could only be run on account balances, and it can be shown that account balances can never have commas, then this would indeed be an error of modular reasoning.

## Invalid representations

This program listen to a stream of data (i.e. Kafka) and for every event it makes an http request to another service.

The events correspond to messages in a chat service between users and businesses: each event either originates from a user or from a business (at least for now...!!)

```
def call_endpoint(http_request_library, params):
    try:
        http_request_library(**params).result()
    except Exception as e:
        # log_error(e)
        raise e

def process(event_payload):
    new_payload = {
        "event_id": event_payload["id"],
    }
    if event_payload["user_id"]:
        new_payload["user_id"] = event_payload["user_id"]
        call_endpoint(HttpRequestLibrary, new_payload)
    if event_payload["business_id"]:
        new_payload["business_id"] = event_payload["business_id"]
        call_endpoint(HttpRequestLibrary, new_payload)
```

What we can see in the code above is the following:
```

- The developer assumed that the events will only originate from a user ('user_id') or from a business ('business_id').

However, the previous constraint is not 100% guaranteed: in the future there might be a requirement to send messages (events) that are common to both consumer and business (and therefore populate both properties user_id/business_id).

In that case, the endpoint would be called twice:

- once with an event containing 'user_id'
- once with an event containing 'business_id' and this would be wrong.

The events in the system for now only have either of those properties, so the code:

- works fine in production
- the tests added by the developer pass (they test both situations)
- the tests are missing the possibility of an event having both the 'user_id' and 'business_id' properties set.

Comment

What is the specification for the payload structure? What is the specification of the endpoint-calling behavior?

We could say that the tests are the observable artifact that describes the specification of the payload structure. In that case it would be clear that a payload should not contain both a ‘user_id’ and a ‘business_id’. It is a problem that it is possible to create an invalid payload and the way to overcome this is to make it impossible (according to the representation/valid principle), but it is not an error of modular reasoning.

On the other hand, we could say that the behavior of calling the endpoint twice is not specified. If a user of the module tries to send both a ‘user_id’ and a ‘business_id’ and relies on the fact that it calls the endpoint twice, that would be an error of modular reasoning which would break the calling code as soon as the ‘bug’ is ‘fixed’ to call it only once.

New major version

Let's suppose it's the early 2000's, and we have a Python program that tells me how to evenly distribute some chocolate chip cookies among some friends. But no one likes breaking cookies at the risk of dropping crumbs and making portions uneven, so everyone can only get a whole number of cookies. Such a program can look like this:

```
def cookies_per_person(num_cookies, num_people):
```

```
        if num_people <= 0:
                return 0
        return num_cookies / num_people
```

I run the program with 7 cookies and 2 people as arguments, and it correctly returns 3 cookies per person. Great, it works from the Level 1 layer of logic!

I now run this program for all integers, and it still returns correct values. Awesome, this works at the Level 2 layer too!

We've been running this program for decades and it's been reliable in keeping countless numbers of friendships well-fed. But now it's the year 2022, and it's time to upgrade to Python 3. But oh no! Now when I try to distribute 12 cookies among 5 friends, it now thinks I should give 2.4 cookies per person, and as stated earlier, giving fractional portions of cookie is forbidden. Chaos ensues, and all my friends leave me.

Now we have an example of a Level 3 defect, since when we upgraded to Python 3, our program is no longer "correct." As described in the article, at the level of logic, "a program is wrong if the reasoning for why it should be correct is flawed." When I was using Python 2, I relied on the assumption that integer division always returns an integer. I didn't count on Python 3 returning a float for this same operation.

Comment

While this may come as a rude surprise, it is not an error of modular reasoning since the code was correctly written according to the specification (and there was no other way to get integer division in Python 2 as far as I’m aware).

Narrow interpretation

```
class message:
        field sender    : Id
        field receiver  : Id
        field timestamp : Datetime
        field body      : Text

def mark_as_read(message):
        conversationId = make_conversation_id(message.sender, message.receiver)
        update_conversation_marker_in_db(message.receiver, message.timestamp)

def make_conversation_id(a, b):
        sort([a, b]).join(":")
```

```

`make_conversation_id()` will return a value that identifies a single conversation between the sender and receiver of a message. The function's correctness relies on the property that any two users may only communicate in a single, 'direct' conversation.

If this program is extended to support 'group' conversations, then `make_conversation_id` will no longer give a correct result. Given a message sent from user A to user B, it is no-longer knowable whether the message was sent in A's and B's 'direct' conversation, or if the message was sent in a 'group' conversation of which A and B are participants.

## Comment

Since the specification is that there only exists one conversation between a particular sender and receiver, this is not an error of modular reasoning, however requirements may change later.

Group conversations could be implemented without breaking the current code, simply by specifying that group messages are sent to the group or received from the group, maintaining the invariant that there is only one conversation between each pair of entities. A possible error of modular reasoning in this scenario could be the assumption that a sender or receiver must be a user.
```