



Seeing the bigger picture

Or, “Can you see the bug?”

Part 2

Andrzej Krzemieński

Setting the stage

- W&B calculator: “Is thus loaded aircraft safe to fly?”
- Life-critical system
 - Wrong results can cause injuries or deaths
- Not a real-time system
 - If the program goes down, we can calculate manually



Agenda

- Example – magic value
 - Find the bug
 - Find the root cause
 - Take precautions
- Exercise on perceptiveness
- Example – exceptions
- Advice

Dealing with contingency

- Bugs usually reveal in contingent cases
- These execution paths are toughest to get right
- They are given least thought
- They are least tested
- Bug can hide there undetected for long time

Find the bug

```
double Flight_plan::weight() {
    if (Impl* impl = get_impl()) { // lazy init?
        return impl->compute_weight();
    }
    else {
        // TODO: need to handle it
    }
}
```

Can you see the bug?

Find the bug

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

Can you see the bug?

Find the bug

```
bool too_heavy(Flight_plan const& p)
{
    return p.weight() > p.aircraft().max_weight();
}
```

Inadvertent consequence:

- lazy load problem?
→ aircraft never too heavy

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

Find the root cause

“Bogus weight”:

- A documented interface
- Useful optimization
- Following the design of `atoi()`

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

The caller:

- Didn't read the instructions
- Is not cautious enough

```
bool too_heavy(Flight_plan const& p)
{
    return p.weight() >
        p.aircraft().max_weight();
}
```


Find the root cause

“Bogus weight”:

- A documented interface
- Useful optimization
- Following the design of `atoi()`

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

The caller:

- Didn't read the instructions
- Is not cautious enough

```
bool too_heavy(Flight_plan const& p)
{
    return p.weight() >
        p.aircraft().max_weight();
}
```

WRONG!

Find the root cause

“bogus weight”:

- Returns two different things: *either* weight *or* error code

Declaration:

```
double Flight_plan::weight();
```

- How am I supposed to see it?
- “weight”, “double” --> I should be able to compare with <
- The design is *counterintuitive, deceptive*

Taking precautions

What if we returned NaN?

Same problem:

```
(NaN > max_weight()) == false
```

- Comparisons with NaN always return false
- NaN has better be never used (never divide 0/0)

What if we returned Infinity?

- Comparison would return “too heavy”, but don’t do that...
- We will cover it later

Taking precautions

- Irregularity of the results should be reflected in the interface.
- Can't afford changing the interface? Use exceptions.

Taking precautions

Boost.Optional:

```
optional<double> Flight_plan::weight();
```

- Clear message: why is it not just double?

But:

- `sizeof(optional<double>) > sizeof(double)`
- No information about the cause of failure
- Accidental comparison still works:

```
(optional<double>() > max_weight()) == false
```

Taking precautions

Expected<T>:

```
expected<double> Flight_plan::weight();
```

- Clear message.
- Contains info about the cause of failure

But:

- Bigger than `double`
- Accidental comparison may still work

Taking precautions

Improve the trick:

```
class OptionalQuantity {
    double val;
public:
    explicit OptionalQuantity(double v) : val(v) {}
    // precondition: v >= 0.0
    OptionalQuantity() : val(NaN) {};
    bool has_value() const { return !isnan(val); }
    double quantity() const { return val; }
    // precondition: has_value()
};
```

Taking precautions

Improve the trick:

- Still a `double`, but with different interface
- Using type system!
- `sizeof(OptionalQuantity) == sizeof(double)`
- No possibility of accidental comparison

```
class OptionalQuantity {
    double val;
public:
    explicit OptionalQuantity(double v) : val(v) {}
    // precondition: v >= 0.0
    OptionalQuantity() : val(NaN) {};
    bool has_value() const { return !isnan(val); }
    double quantity() const { return val; }
    // precondition: has_value()
};
```


Taking precautions

Clever ideas:

- Is it intuitive (to anyone else but you)?
- Are your colleagues convinced?

Exercise – perceptiveness

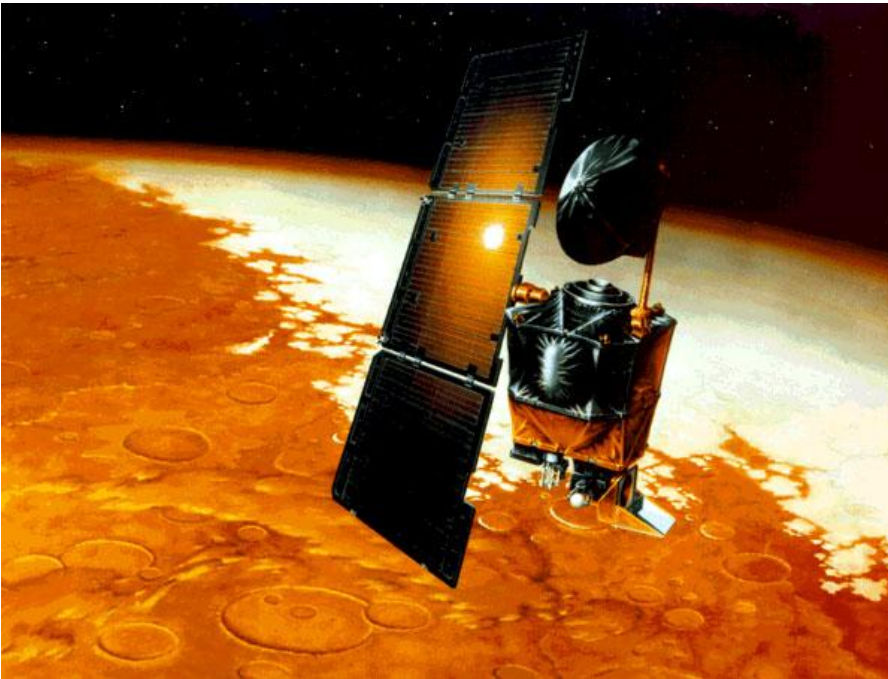
```
double Flight_plan::weight() {  
    if (Impl* impl = get_impl())  
        return impl->compute_weight();  
    else  
        throw Internal_error();  
}
```

Can you see other potential bugs?

Exercise – perceptiveness

Physical quantity but no units!

- Newtons? Pound-force?



Courtesy NASA/JPL-Caltech.

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

Exercise – perceptiveness

Using floating point numbers.

- It is not same as “quantity”
- Limited precision
- Are rounding errors acceptable?

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

Exercise – perceptiveness

Defensive if's!

- You cannot tell the logic flow from contingency handling
- Difficult to understand
- They may become offensive
- Asks for trouble (like ours)

```
const double BOGUS_WEIGHT = -666.66;

double Flight_plan::weight() {
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT;
}
```

Exercise – perceptiveness

Improved solution:

```
Newton's Flight_plan::weight() {  
    Impl& impl = get_impl(); // may throw  
    return impl.compute_weight();  
}
```

- A dedicated type, no unit confusion, no fp problems
- A dedicated path for contingency: exceptions

Example – exceptions



Example – exceptions

Improved version:

```
Newton's Flight_plan::weight() {  
    if (Impl* impl = get_impl())  
        return impl->compute_weight();  
    else  
        throw Internal_error(); // poor...  
}
```

- If I have a contingency, I have to throw.
- But, doesn't someone need to catch it now?

Example – exceptions

```
bool too_heavy(Flight_plan const& p) {  
    try {  
        return p.weight() > p.aircraft().max_weight();  
    }  
    catch (Internal_error const& exc) {  
        // TODO: need to handle it  
    }  
}
```

- Almost a defensive if
- Inadvertently concealing an exception
- We have an undefined (random) behavior

Example – exceptions

But I followed a ‘good advice’: “Catch exceptions, or else `std::terminate()`.”

- Don’t just follow any good advice.
- Think.
- See the bigger picture.
- Understand the consequences.

Example – exceptions

```
bool too_heavy(Flight_plan const& p) {  
    try {  
        return p.weight() > p.aircraft().max_weight();  
    }  
    catch (Internal_error const& exc) {  
        logger.log(exc);  
        return true; // safest bet  
    }  
}
```

- Now, I can never give the 'unsafe' response
- Same effect as if returning +Infinity

Example – exceptions

End user perspective:

```
> load 7200kg in compartment 1  
aircraft maximum weight exceeded  
> load 7198kg in compartment 1  
aircraft maximum weight exceeded  
> load 7190kg in compartment 1  
aircraft maximum weight exceeded  
> load 7000kg in compartment 1  
aircraft maximum weight exceeded
```

Example – exceptions

End user perspective:

- We deceived the user. We let her think that by reducing the weight she will make the error disappear.
- We wasted her time (how much?),
- Caused delays on the airport,
- Incurred cost.

And it is likely, that you did that for nothing...

Example – exceptions

The top level code:

```
void process_user_input(string input) {  
    try {  
        variant<Command, Bad_input> command =  
            validate(input);  
        process(command);  
    }  
    catch (Internal_error const& exc) {  
        output(build_message(exc));  
    }  
}
```

Example – exceptions

End user perspective:

```
> load 7200kg in compartment 1  
! request ignored; internal error; try again?
```

“Something is wrong, I need to restart, or do work by hand.”

- We had the code that reports a malfunction correctly.
- We interfered with the mechanism.
- Why?

Example – exceptions

The fear for `terminate()`

- `terminate()` is not the worst thing that can happen.
- `terminate()` is a viable option (even if not best).
- “Not crashing” is never the primary goal.
- You can't afford to crash? Can you afford to lie?
- You cannot fix global problems locally.

Advice

```
bool too_heavy(Flight_plan const& p);
```

- Returns `true` or `false`. Neither answer is right on error.
- Having detected contingency is a vital info. Don't conceal it.

Advice

How not to forget:

```
double Flight_plan::weight() {
    if (Impl* impl = get_impl()) {
        return impl->compute_weight();
    }
    else {
        std::terminate();
        // TODO: improve in spare time
    }
}
```

Advice

Don't catch exceptions

Advice

Don't catch exceptions unless:

Advice

Don't catch exceptions unless:

- You only want to translate it
 - Change type
 - Add information
 - Turn into an error code
- You only do some local clean-up and re-throw
- You close the operation of a sealed module
 - Which does not affect other parts of the program
 - `main()`

Conclusion

- Even when working on small piece, consider the bigger picture
- We have responsibility
 - Sometimes we have to try hard to find it
- More life-critical software out there than you can imagine