Modular Reasoning

A software quality skill

Doug Slater

About Me

- Coding since 1999
- BS Computer Science 2013
- MS Computer Science 2014
- C#, C++, Python, more
- 10 years medical device software
  - robust QA
- 4 months at Climavision

Contents

Goals

- Teach design-level thinking
- Improve code quality

Motivating Quiz

Question: As part of a large application, you are reading an input and need to escape all single quotes. Which of these two options is better and why? Assume both do the exact same thing.

```
x = sanitize(readInput())
```

```
x = readInput().replaceAll("'", "\\'")
```

- A: Option 1 because it's shorter
- B: Option 1 because option 2 is incorrect, even though it always works
- C: Option 2 because it's more clear what it does
- D: Option 2 because it's more efficient, avoiding the function call overhead

3...

2...

1...

Answer: B

Question: How could something that always works be incorrect?

Answer: Errors in modular reasoning.

Source: https://mirdin.com/quizzes/software-design-quiz/

Motivating Quiz

Question: Which is correct?

```
write(1, "foo", len)
```

```
write(stdout, "foo", len)
```

Answer: The second

Question: Why?

Answer: The first is incorrect because

- It is not possible to know that it works without making assumptions about the rest of the system.
- The number `1` is a secret to another module.
- Even though it works now, this secret could change without notice.
- The idea of the number `1` and the idea of the stream `stdout` are different, even though they are the same in implementation right now.

The implementation/interface distinction states that the guarantees of what a function must do are different from what it actually does.

Motivating Quiz

```
x = sanitize(readInput())
```

```
x = readInput().replaceAll("'", "\\'")
```

The general idea of a sanitized string and the specific idea of a string with all single quotes escaped by a backslash are different.

The exact sanitization rules should be considered a secret to its module which could change without notice.

What is a Module?

- From PL (Programming Language) Theory
- A unit of organization that can keep a secret
- The secret is shared internally
- The secret is concealed externally

- local constants, calculations

```
def sanitize(input_string):
  # the sanitization rules are a secret to this function
  return input_string.replace("'", "\\'")
```

- `private`, `protected` members

```
public class UsTaxCode : ITaxCode
{
  // A secret to the class
  private const int _retirementAge = 65;

  public bool IsRetirementAge(int age) => age > _retirementAge;
}
```

```
public class AsciiTools
{
  // another secret
  private const int _minChar = 65;

  public bool IsAlphanumeric(int c) => c > _minChar;
}
```

Question: These secrets are identical. Should they be merged?

Answer: No, because they have different reasons to change.

- e.g. .dll, .so files
- `internal` members

```
namespace MyProject;

// Cannot be referenced outside MyProject.dll
internal class ThisAssemblyOnly
{
  ...
}
```

- Sometimes literally named "Modules"
  - Early languages: ALGOL, OCaml, ML
  - Newer languages: JavaScript, Rust, C++20
- Sometimes called "Packages" (Dart, Go, Java)

What is Modular Reasoning?

The ability to make decisions about a module while looking only at its implementation and the specification (i.e. interface, i.e. contract) of other modules.

Modular reasoning lets an engineer reason about the correctness of a module without reading the rest of the program.

Question: How big should a module be?

Answer: From cognitive science [1][2], humans can hold about 7 pieces of information in short-term memory. Modules should hold up to that amount.

- [1] https://www.oreilly.com/library/view/code-that-fits/9780137464302/
- [2] https://www.manning.com/books/the-programmers-brain

Design-Level Thinking

- **Key idea**: Design is apart from the code.
- Design is about a shared fiction
- Just like Democracy does not exist
  - US, Russia, and Ukranian Sovereignty
- Newton's Law of Gravity is just a model
  - and only approximately true at the scale of humans
- These are patterns imposed by our minds on the world
- Similarly, software design is a narrative over physical code
  - **There can be many code implementations that satisfy a design**

> There can be many code implementations that satisfy a design

Here's a specification.

```
public interface ICalculator
{
  // Returns the integer quotient
  public int Divide(int x, int y);
}
```

What's the difference between these two implementations?

```
public class X : ICalculator
{
    public int Divide(int x, int y) => x / y;
}
```

```
public class Y : ICalculator
{
    public int Divide(int x, int y) => y switch
    {
        0 => int.MaxValue,
        _ => x / y
    };
}
```

> There can be many code implementations that satisfy a design

Here's a specification.

```
public interface ICalculator
{
  // Returns the integer quotient
  public int Divide(int x, int y);
}
```

What's the difference between these two implementations?

```
public class RawCalculator : ICalculator
{
    public int Divide(int x, int y) => x / y;
}
```

```
public class SafeCalculator : ICalculator
{
    public int Divide(int x, int y) => y switch
    {
        0 => int.MaxValue,
        _ => x / y
    };
}
```

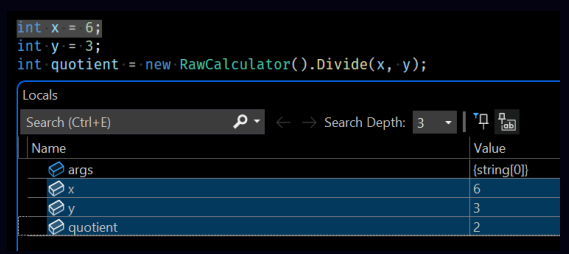
The Three Levels of Program Correctness

Level 1: Runtime

- Considers only a specific execution of a program

```
int quotient = new RawCalculator().Divide(6, 3);
// { quotient == 2 }
```

- What you see at a breakpoint in a debugger



Level 1 says:

> A program is incorrect if it runs and produces a wrong result.

The Three Levels of Program Correctness

Level 2: Concrete implementation / Code

- Considers all possible executions of a program
- What the current implementation could do given arbitrary inputs and an arbitrary environment.

```
public class RawCalculator : ICalculator
{
    public int Divide(int x, int y) => x / y;
}

int impossible = new RawCalculator(1, 0); // throws
```

Level 2 says:

> A program is incorrect if there exists some environment or input under which it produces a wrong result

The Three Levels of Program Correctness

Level 3: Logic

- Considers how the code is derived
- At this level, we consider only the abstract specification (i.e. interface, i.e. contract) of each module.
- We don't look at any particular implementation.
- We assume a module may be replaced at any time with a different implementation.
- We can then determine the module's correctness by only looking at its code and the contracts of its dependencies.

```
ICalculator calc = ...;
int quotient = calc.Divide(6, 0);
```

- Junior dev: "Let's ship it. It's ok because I know calc is a `SafeCalculator`."
- Senior dev: "That is not a stable guarantee. What if tomorrow we are given a `RawCalculator`?"
- Junior dev: "Ok. We need to either not pass `0` or handle `DivideByZeroException`."

Level 3 says:

> A program is incorrect if the reasoning for why it should be correct is flawed.

Stable Guarantees

▒▒ Stable Guarantee

- programming only to guarantees made by other modules' spec.

▒▒ Spec: Preconditions

- the facts a function assumes to be true before it runs

▒▒ Spec: Invariants

- the facts a function assumes to remain true while it runs
- for pure functions, invariants == preconditions

▒▒ Spec: Postconditions

- the facts a function guarantees to be true after it has run

Stable Guarantees

```
public class RawCalculator : ICalculator
{
    // Strong assumption: y is not 0
    // Weak guarantees: could throw
    public int Divide(int x, int y) => x / y;
}
```

```
public class SafeCalculator : ICalculator
{
    // Weaker assumption: y can be 0
    // Stronger guarantees: will not throw
    public int Divide(int x, int y) => y switch
    {
        0 => int.MaxValue,
        _ => x / y
    };
}
```

Stable Guarantees > Precise Advice

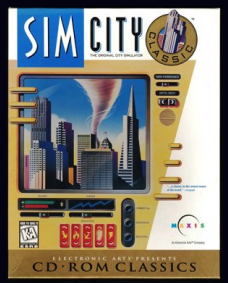
"You have heard it said, but I tell you..."

- Heard of "Defensive Coding"?
  - More precise: Program to stable guarantees
  - Think about preconditions, invariants and postconditions.
- Heard of "Encapsulation"?
  - More precise: think about modules and secrets.
- Heard of Arrange, Act, Assert?
  - More precise: the Hoare Triple, from Formal Methods
  - `{P}C{Q}`
  - https://en.wikipedia.org/wiki/Hoare_logic

Errors in Modular Reasoning

1. Depending on a stricter output (stronger postconditions) than the specification guarantees.
2. Depending on being able to use looser input (weaker preconditions) than the specification guarantees.

Errors in Modular Reasoning: Example



"the original Sim City had a use-after-free error. At the concrete implementation/code level, this was totally fine, since freed memory in DOS was valid until the next malloc, and so the program worked."

"At the logical level, this was a defect, because the spec for `free` says you need to act as if ... any future `free` implementation may actually [reallocate the memory]... once Windows 3.1 rolled around with a new memory manager, SimCity would start crashing.

"Microsoft had to add a special case to check if SimCity was running and switch to a legacy memory manager if so."

Sources

https://www.pathsensitive.com/2018/01/the-three-levels-of-software-why-code.html

https://www.joelonsoftware.com/2004/06/13/how-microsoft-lost-the-api-war/

```
[assembly: InternalsVisibleTo("Tests")]

public class UsTaxCode : ITaxCode
{
  // A secret to the class
  /*private*/ internal const int _retirementAge = 65;

  public bool IsRetirementAge(int age) => age > _retirementAge;
}
```

```
// In Tests.dll
[Fact]
public void RetirementAge_Is65()
{
  // Don't do this!
  new UsTaxCode()._retirementAge.Should().Be(65);
}
```

Q: Why is this an error in modular reasoning?

A: It isn't a stable guarantee.

But this is:

```
// In Tests.dll
[Theory]
[InlineData(64, false)]
[InlineData(65, false)]
[InlineData(66, true)]
public void IsRetirementAge_IsCorrect(int age, bool expected)
{
  // OK, uses only public API
  new UsTaxCode().IsRetirementAge(age).Should().Be(expected);
}
```

Review Quiz

- Q: How can a program that never goes wrong still be wrong?
  - **A: Errors in modular reasoning**
- Q: What kind of bug does an error in modular reasoning produce?
  - **A: A bug where a program works now but it might break in the future.**
- Q: Modular reasoning lets an engineer...
  - A. Make smaller classes, functions, and files.
  - B. Feel smarter
  - C. Reason about code correctness without reading the rest of the program.
  - **Answer: C**

Summary

- Our goal is not to just deliver correct software today.
- It's to continue to deliver correct software far into the future.
- We can do that by remembering Level 3, the layer of design and logic.
- This practice is as valuable or more than robust automated tests.

Resources

- https://mirdin.com
- https://mirdin.com/quizzes/software-design-quiz/
- https://www.pathsensitive.com/2018/01/the-three-levels-of-software-why-code.html
- https://www.pathsensitive.com/2018/01/the-design-of-software-is-thing-apart.html
- https://note89.github.io/pipe-dream/
- https://www.slater.dev/a-design-is-a-mold-for-code/

Discussion