The Representable/Valid Principle (RVP)

Making Invalid States Impossible

Doug Slater

Contents

Quiz from last talk

- Q: Code units A and B are coupled if, whenever ___ changes, ___ is also likely to change.
  - **A: A, B**
- Q: Your program was written via some tree of ___ and ___.
  - **A: assumptions, decisions**
- Q: The easier it is to ___ the design from the code, the more ___ it is.
  - **A: reverse-engineer, EDP-compliant** (Embedded Design Principle)
- Q: The ___ checks if code matches its natural language description
  - **A: Plain English Test**
- Q: ___ Antipatterns describe poor choice of words in the code
  - **A: Linguistic**

Goals

- Define state, abstract state, and concrete state
- Define the Representable/Valid principle
- Learn MIRO: The 4 incorrect mappings between abstract state and concrete state
- Make bugs impossible

Intro

> "Make illegal states unrepresentable"

We've all heard that.

What does it mean?

State

What is state?

State is information about the world that can change.

Examples:

- The wifi is connected.
- The patient's heart rate is 62.
- The wind speed is 100 knots.
- The car dashboard message says "LOW TIRE PRESSURE".

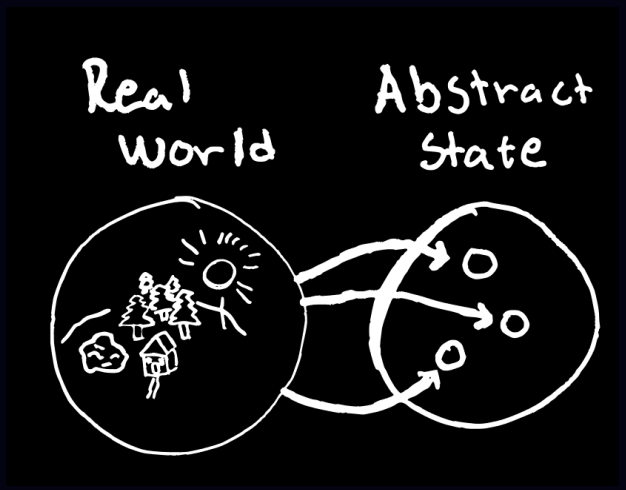
Abstract State

An **abstraction** is a mapping from a complex set to a simpler one.

Q: What is abstract state?

▒▒ Abstract state

- An **abstract state** is a simpler but useful model of reality



Q: How would you model a smart thermostat?

You might include:

- Whether the heat is on
- Whether the cooling is on
- The heat setpoint
- The cool setpoint
- The inside temperature
- The outside temperature

This is an abstract state for a smart thermostat.

The abstract state discards information about reality that is not useful:

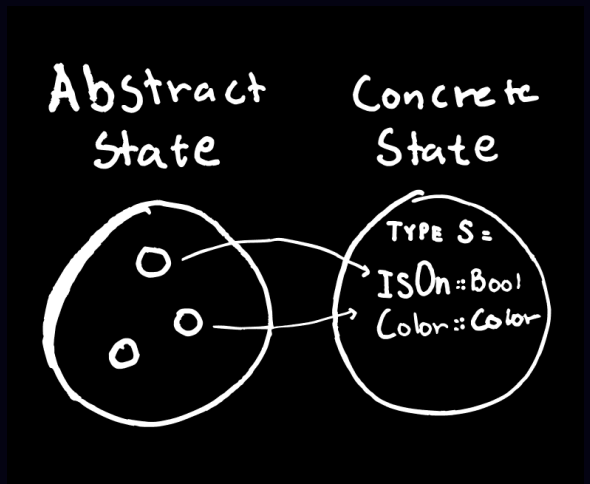
- Voltage in the control wires
- Whether the heater is gas or electric
- The cost of gas or electricity
- Whether the outside temperature comes from a sensor or a web service

Concrete State

Q: What is concrete state?

▒▒ Concrete state

- The **concrete state** is the data types in code that represent the abstract state.

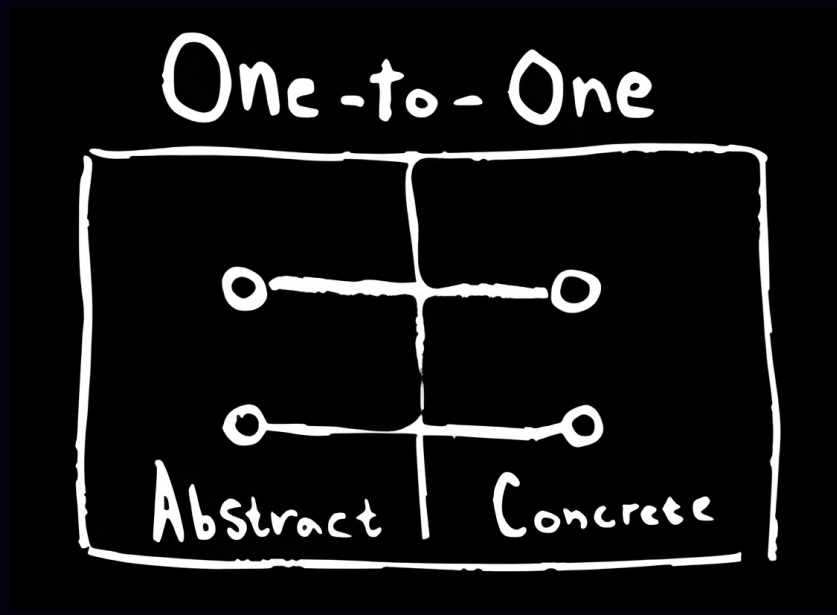



```
bool IsWifiConnected
int PatientHeartRateBpm
double WindSpeedKnots
string DashboardMessage
```

```
bool IsHeatOn
bool IsCoolOn
int HeatSetpoint
int CoolSetpoint
int InsideTemp
int OutsideTemp
```

▒▒ Representable/Valid Principle (RVP)

Keep a one-to-one correspondence between representable (concrete) and valid (abstract) states of the program.

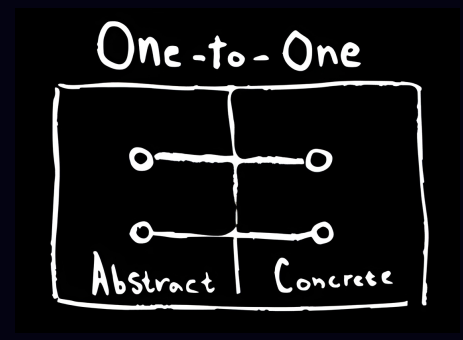


In other words, make it impossible to even represent bugs.

How do we do that?

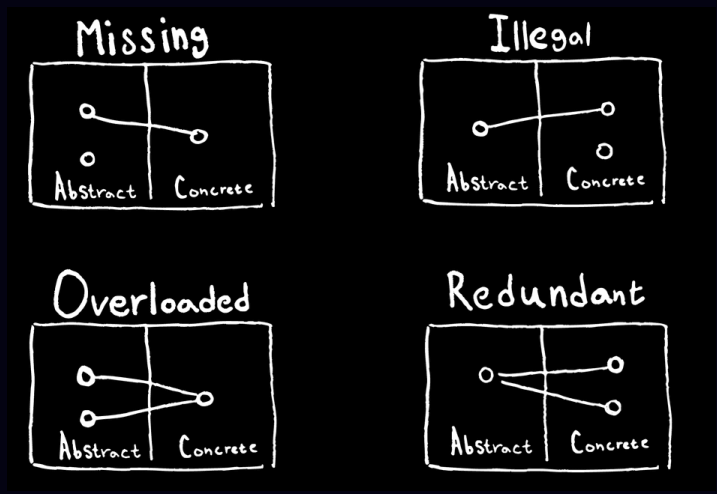
MIRO

In keeping with the RVP, we want a one-to-one mapping between the abstract state and concrete state.



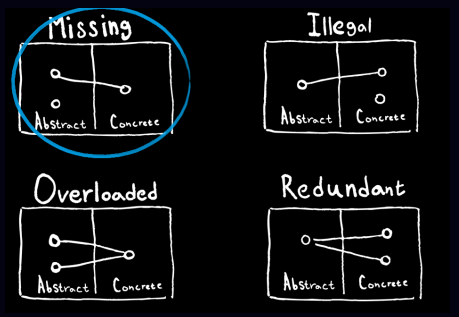
MIRO is an acronym for the 4 incorrect mappings:

- **M**issing
- **I**llegal
- **R**edundant
- **O**verloaded



Missing States

-> There are abstract states that the concrete states cannot express.



Example: Not having an `Error` or `Option` state

```
int OutsideTemperature { get; set; }
```

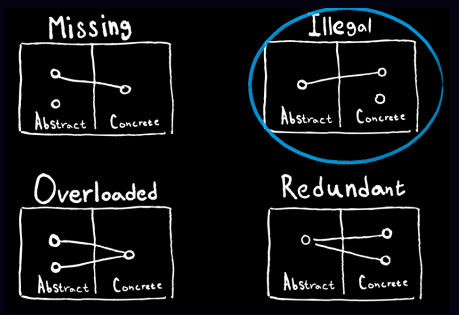
- What if the value is not known?
- There are two abstract states (valid states): value known, not known
- There is one concrete states (representable states): the value

Slightly better:

```
int? OutsideTemperature { get; set; } // null: Value not known
```

Illegal States

-> There are concrete states that have no mapping to an abstract state.



Example: A setting that enables other settings

```
record User(bool IsAdmin, bool CanSetUserPasswords);

User(true, true)    // OK
User(true, false)   // OK
User(false, true)   // illegal
User(false, false)  // OK
```

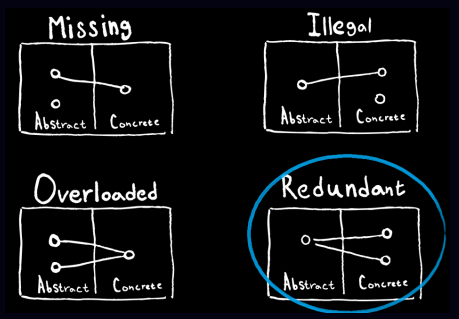
- `CanSetUserPasswords` should not be true when `IsAdmin` is `false`.
- There are 3 abstract states, 4 concrete

Better:

```
abstract record Role;
record Admin(bool CanSetUserPasswords) : Role;
record RegularUser : Role;
```

Redundant States

-> There is more than one way to represent an abstract state.



Example:

```
record Thermostat(bool IsOn, bool IsHeatOn);

Thermostat(false, false)  // redundant
Thermostat(false, true)   // illegal
Thermostat(true, false)   // illegal
Thermostat(true, true)    // redundant
```

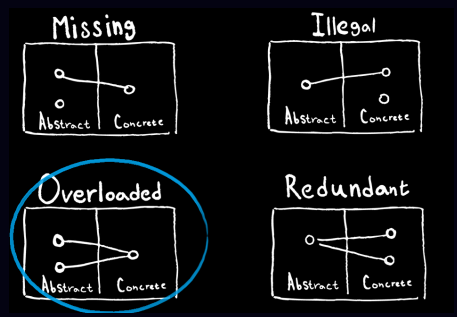
- There are two abstract states: on, off
- There are four concrete states. Two illegal, two redundant

Slightly Better:

```
record Thermostat(bool IsHeatOn)
{
    public bool IsOn => IsHeatOn;
}
```

Overloaded States

-> There is one concrete state representing two or more abstract states.



Example: A chat app that says "no messages", then suddenly shows them.

The app cannot represent the difference between the abstract states

- "haven't fetched messages yet"
- "fetching messages"
- "fetching messages failed"
- "there are no messages"

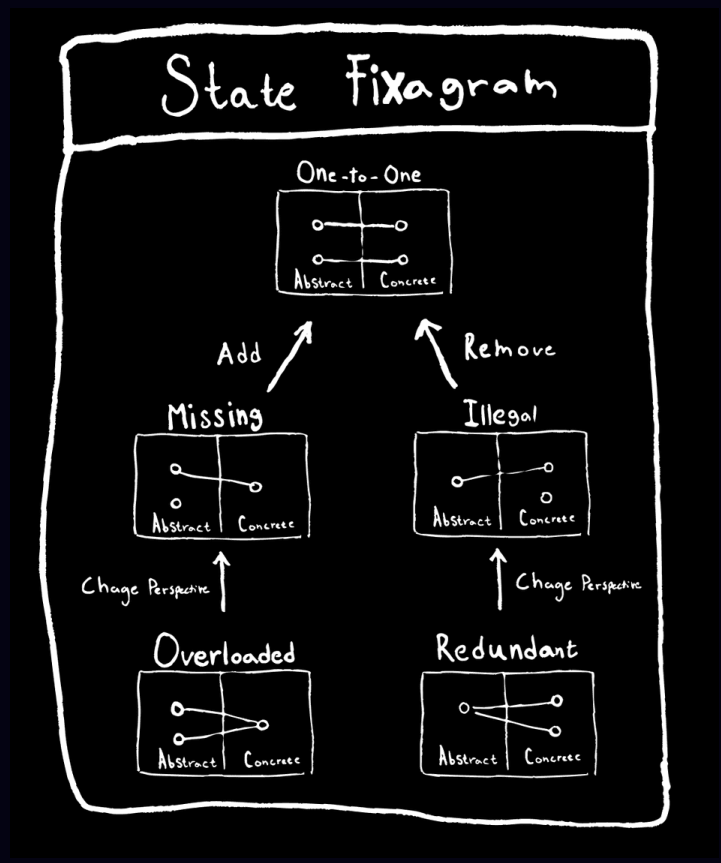
```
record State(string[] Messages)
```

- There are 4 abstract states: not fetched, fetching, fetched, failed
- There are 2 concrete states: empty list, nonempty list

Better:

```
abstract record State;
record NotFetched : State;
record Fetching : State;
record Fetched(string[] Messages) : State;
record Failed(string ErrorMessage) : State;
```

Algorithm for fixing invalid states:



Applying MIRO

Recall our abstract state for a thermostat:

- Whether the heat is on
- Whether the cooling is on
- The heat setpoint
- The cool setpoint
- The inside temperature
- The outside temperature

Consider this error-prone concrete state:

```
record BuggyThermostat(
    bool IsOn,
    bool IsHeatOn,
    bool IsCoolOn,
    int Setpoint,
    int InsideTemp,
    int? OutsideTemp = null);
```

What's wrong with it?

- Missing state
  - There is only one setpoint, but the UI lets the user set two.
- Illegal states
  - `IsHeatOn` and `IsCoolOn` could both be true
  - `IsOn` could be `false` but there is a setpoint
  - `IsOn` could be `false` while `IsHeatOn` or `IsCoolOn` are `true`
  - It's possible to mix celsius and fahrenheit.
  - It's possible to assign a temperature reading as the setpoint or vice versa.
- Redundant state
  - `IsOn` is implied by `IsHeatOn || IsCoolOn`
- Overloaded states
  - What does `null` mean?
    - Haven't fetched?
    - Currently fetching?
    - Server returned no data?

For maximum shenanigans:

```
new BuggyThermostat(
   IsOn: false,
   IsHeatOn: true,
   IsCoolOn: true,
   Setpoint: 25, // Comfortable in Celsius
   InsideTemp: 72,
   OutsideTemp: null); // Who knows why!
```

One solution:

- Use the C# type system to make these bugs unrepresentable.
- The bugs will not even compile.
- Bonus: It is EDP-compliant

```
abstract record Unit(int Value);
record Fahrenheit(int Value) : Unit(Value);
record Celsius(int Value) : Unit(Value);
```

```
abstract record Temperature(Unit Value);
record Setpoint(Unit Value) : Temperature(Value);

record Reading(Unit Value) : Temperature(Value);
```

```
abstract record RunState;
record OffState : RunState;
record HeatState(Setpoint Setpoint) : RunState;

record CoolState(Setpoint Setpoint) : RunState;
```

```
abstract record FetchState;
record NotFetched : FetchState;
record FetchInProgress : FetchState;
record FetchError(string Message) : FetchState;

record Fetched(Reading Temperature) : FetchState;
```

The new thermostat:

```
record BetterThermostat(
    RunState RunState,
    Reading InsideTemp,
    FetchState OutsideTemp);
```

Example use:

```
var thermostat = new BetterThermostat(
    new HeatState(new Setpoint(new Celsius(25))),

    new Reading(new Fahrenheit(72)),
    new FetchInProgress());
```

Applying MIRO

Q: Why is `BetterThermostat` better?

A: It's impossible to write a bug.

Let's look at validation logic for `BuggyThermostat` and `BetterThermostat`.

```
new BuggyThermostat(
   IsOn: false,
   IsHeatOn: true,
   IsCoolOn: true,
   Setpoint: 25, // Comfortable in Celsius
   InsideTemp: 72,
   OutsideTemp: null); // Who knows why!
```

```
record BetterThermostat(
    RunState RunState,
    Reading InsideTemp,
    FetchState OutsideTemp);
```

```
bool IsValid(BuggyThermostat t) =>
     (  (t.IsOn &&  (t.IsHeatOn || t.IsCoolOn))

     || (!t.IsOn && !(t.IsHeatOn || t.IsCoolOn))  )

   && !(t.IsHeatOn && t.IsCoolOn)
   && ((t.IsOn && t.Setpoint >= 0) || (!t.IsOn && t.Setpoint < 0));
```

```
bool IsValid(BetterThermostat t) => true;
```

- What are the chances there is a bug on the left?
- What are the chances there is a bug on the right?
- Design view: We moved risk from runtime to compile time

Quiz

- Q: An _____ is a mapping from a complex set to a simpler one.
  - **A: abstraction**
- Q: The _____ is a simpler but useful model of reality
  - **A: abstract state**
- Q: The ____ is the data types in code that represent the abstract state.
  - **A: concrete state**
- Q: The 4 incorrect mappings between abstract state and concrete state are...

  M____ Missing

  I____ Illegal

  R____ Redundant

  O____ Overloaded

- Q: The ____ keeps a one-to-one correspondence between representable and valid states of the program.
  - **A: Representable/Valid Principle (RVP)**

Summary

- Defensive code is a smell. It implies the possibility for bugs.

```
// I smell bugs
try {
  bool ok = DoThing();
  if (!ok) ...
}
catch (SomeException) {
 ...
}
catch (OtherException) {
  ...
}
```

- Often manifests as conditionals and `try`.
- A program with no representable invalid states cannot have bugs.

```
bool IsValid(BetterThermostat t) => true;
```

▓▓ Principle:

The power of a design isn't how much it can do, it's what it can't do.

Resources

- https://note89.github.io/state-of-emergency/
- https://github.com/nref/speaking/tree/main/representable_valid_principle

Discussion