Semantic
& Immutable Types

Semantic Type

A type that captures
exactly one simple concept

Semantic Types in .NET

System

DateTime

DateTimeOffset

Timespan

Uri

Version

System.IO

FileInfo

DirectoryInfo

System.Numerics

Complex

Why?

What problem do they solve?

Example

Without Semantic Types

```
public void Save(string file)
{
    using (var stream = File.Create(file))
    {
        SaveToStream(stream);
    }
}
```

Crash during save causes previous file to be lost

```
public void Save(string file)
{
    var index = file.LastIndexOf('.');
    var backup = file.Substring(0, index) + ".bak";
    if (File.Exists(file))
    {
        File.Move(file, backup);
    }

    using (var stream = File.Create(file))
    {
        SaveToStream(stream);
    }
}
```

Crash during save causes previous file to be renamed

```
public void Save(string file)
{
    var folderIndex = file.LastIndexOf('\\');
    var extIndex = file.LastIndexOf('.');

    var folder = file.Substring(0, folderIndex + 1);
    var tempFile = folder + Guid.NewGuid().ToString("D");
    using (var stream = File.Create(tempFile))
    {
        SaveToStream(stream);
    }

    var backupFile = Path.ChangeExtension(file, "bak");
    if (File.Exists(file))
    {
        File.Move(file, backupFile);
    }

    File.Move(tempFile, file);
}
```

Save crashes if backup file already exists

```
public void Save(string file)
{
    var folderIndex = file.LastIndexOf('\\');
    var extIndex = file.LastIndexOf('.');

    var folder = file.Substring(0, folderIndex + 1);
    var tempFile = folder + Guid.NewGuid().ToString("D");

    using (var stream = File.Create(tempFile))
    {
        SaveToStream(stream);
    }

    var backupFile = Path.ChangeExtension(file, "bak");
    if (File.Exists(backupFile))
    {
        File.Delete(backupFile);
    }

    if (File.Exists(file))
    {
        File.Move(file, backupFile);
    }

    File.Move(tempFile, file);
}
```

Save to missing folder gives misleading error

```
public void Save(string file)
{
    var folderIndex = file.LastIndexOf('\\');
    var extIndex = file.LastIndexOf('.');

    var folder = file.Substring(0, folderIndex + 1);
    if (!Directory.Exists(folder))
    {
        throw new IOException($"Folder {folder} does not exist.");
    }

    var tempFile = folder + Guid.NewGuid().ToString("D");
    using (var stream = File.Create(tempFile))
    {
        SaveToStream(stream);
    }

    var backupFile = Path.ChangeExtension(file, "bak");
    if (File.Exists(backupFile))
    {
        File.Delete(backupFile);
    }

    if (File.Exists(file))
    {
        File.Move(file, backupFile);
    }

    File.Move(tempFile, file);
}
```

How easy is reading this code?

Example

With Semantic Types

```
public void Save(FileInfo file)
{
    using (var stream = file.Open(FileMode.Create))
    {
        SaveToStream(stream);
    }
}
```

Crash during save causes previous file to be lost

```
public void Save(FileInfo file)
{
    var backup = file.ChangeExtension("bak");
    if (file.Exists)
    {
        file.MoveTo(backup);
    }

    using (var stream = file.Open(FileMode.Create))
    {
        SaveToStream(stream);
    }
}
```

Crash during save causes previous file to be renamed

```
public void Save(FileInfo file)
{
    var tempFile = file.Directory.CreateTemporaryFile();
    using (var stream = tempFile.Open(FileMode.Create))
    {
        SaveToStream(stream);
    }

    var backup = file.ChangeExtension("bak");
    if (file.Exists)
    {
        file.MoveTo(backup);
    }

    tempFile.MoveTo(file);
}
```

Save crashes if backup file already exists

```
public void Save(FileInfo file)
{
    var tempFile = file.Directory.CreateTemporaryFile();
    using (var stream = tempFile.Open(FileMode.Create))
    {
        SaveToStream(stream);
    }

    var backup = file.ChangeExtension("bak");
    if (file.Exists)
    {
        file.MoveAndOverwrite(backup);
    }

    tempFile.MoveTo(file);
}
```

Save to missing folder gives misleading error

```
public void Save(FileInfo file)
{
    var tempFile
        = file.Directory.MustAlreadyExist()
            .CreateTemporaryFile();

    using (var stream = tempFile.Open(FileMode.Create))
    {
        SaveToStream(stream);
    }

    var backup = file.ChangeExtension("bak");
    if (file.Exists)
    {
        file.MoveAndOverwrite(backup);
    }

    tempFile.MoveTo(file);
}
```

How easy is reading this code?

What about side by side?

```
public void Save(string file)
{
    var folderIndex = file.LastIndexOf('\\');
    var extIndex = file.LastIndexOf('.');

    var folder = file.Substring(0, folderIndex + 1);
    if (!Directory.Exists(folder))
    {
        throw new IOException($"Folder {folder} does not exist.");
    }

    var tempFile = folder + Guid.NewGuid().ToString("D");
    using (var stream = File.Create(tempFile))
    {
        SaveToStream(stream);
    }

    var backupFile = Path.ChangeExtension(file, "bak");
    if (File.Exists(backupFile))
    {
        File.Delete(backupFile);
    }

    if (File.Exists(file))
    {
        File.Move(file, backupFile);
    }

    File.Move(tempFile, file);
}
```

```
public void Save(FileInfo file)
{
    var tempFile
        = file.Directory.MustAlreadyExist()
            .CreateTemporaryFile();

    using (var stream = tempFile.Open(FileMode.Create))
    {
        SaveToStream(stream);
    }

    var backup = file.ChangeExtension("bak");
    if (file.Exists)
    {
        file.MoveAndOverwrite(backup);
    }

    tempFile.MoveTo(file);
}
```

What did we gain?

Readability
Shorter, easier to read

Mutability
Easier to change

Declarative
"What" not "How"

Reusable
Captured common concepts

Semantic Types

Single Responsibility

Define a single concept well
Simple and testable

Maintenance

Greater consistency
Higher level of abstraction

Enforcement

Compile time checks
Runtime validation

Communication

Common Vocabulary
Shared Understanding

Not actually a new idea

"When (SmallTalk) is used...
the developer extends Smalltalk, creating
a domain specific language by
adding a new vocabulary of
language elements..."

Why Smalltalk? Adele Goldberg
pp 105-107 Communications of the ACM v38 #10 (October 1995)

Write your own

Some tips

```
IEquatable<T> {
    bool Equals(T other) { … }
}

override bool Equals(object obj) { … }

override int GetHashCode() { … }

static bool Equals(T left, T right) { … }

static bool operator ==(T left, T right) { … }
static bool operator !=(T left, T right) { … }
```

Equality

```
IComparable<T> {
    int CompareTo(T other) { … }
}

static int Compare(T left, T right) { … }

static bool operator <(T left, T right) { … }
static bool operator >(T left, T right) { … }
static bool operator <=(T left, T right) { … }
static bool operator >=(T left, T right) { … }
```

Comparison

```
[DebuggerDisplay]

override string ToString() { … }

IFormattable
{
    string ToString(
        string format,
        IFormatProvider formatProvider) { … }
}

T Parse(string s) { … }
bool TryParse(string s, out T value) { … }

Option<T> TryParse(string s) { … }
```

Of Strings and things

+ - * / %

++ --

! ~ & |

true false

^

<< >>

== !=

< >

<= >=

explicit implicit

Operators

Immutable Type

A type where the instances
cannot be modified after creation

Why?

What's wrong with mutable types?

Problems with Mutable Types

Complex

State evolves over time
Hard to reason about

Size

Growth in field count
& number of possible states

Concurrency

Often cannot be shared
across threads

Unverifiable

Often difficult to
verify behaviour

Example

A well behaved Immutable Type

```
public struct Range<T>
    : IEquatable<Range<T>>,
        IComparable<Range<T>>
where T : struct,
    IEquatable<T>,
    IComparable<T>
{
    public T? Upper { get; }
    public T? Lower { get; }

    public Range(T? lower, T? upper)
    {
        Lower = lower;
        Upper = upper;
    }
}
```

Range<T>

A range of values
with optional bounds

```
public bool Contains(T value)
{
    if (Lower.HasValue
        && Lower.Value.CompareTo(value) < 0)
    {
        return false;
    }

    if (Upper.HasValue
        && Upper.Value.CompareTo(value) > 0)
    {
        return false;
    }

    return true;
}
```

Contains()

```
public enum PositionIs
{
    Before,
    LowerBound,
    Within,
    UpperBound,
    After
}
```

PositionIs

```
public PositionIs PositionOf(T value)
{
    if (Lower.HasValue)
    {
        var lower = Nullable.Compare(Lower, value);
        if (lower > 0) return PositionIs.Before;
        if (lower == 0) return PositionIs.LowerBound;
    }

    if (Upper.HasValue)
    {
        var upper = Nullable.Compare(value, Upper);
        if (upper > 0) return PositionIs.After;
        if (upper == 0) return PositionIs.UpperBound;
    }

    return PositionIs.Within;
}
```

PositionOf()

Where is value in relation to the range?

```
public Range<T> WithoutLowerBound()
{
    if (Lower.HasValue)
    {
        return new Range<T>(null, Upper);
    }

    return this;
}

public Range<T> WithoutUpperBound() { ... }
```

With*()

Create variations of the range

```
public Range<T> Extend(T value)
{
    switch (PositionOf(value))
    {
        case PositionIs.Before:
            return new Range<T>(value, Upper);

        case PositionIs.After:
            return new Range<T>(Lower, value);

        default:
            return this;
    }
}
```

Extend()

Make the range bigger

```
public Range<T> ExtendUpperBound(T? value)
{
    if (value == null)
    {
        return WithoutUpperBound();
    }

    if (PositionOf(value.Value) == PositionIs.After)
    {
        return new Range<T>(Lower, value.Value);
    }

    return this;
}

public Range<T> ExtendLowerBound(T? value) { ... }
```

ExtendUpperBound()

Make the range bigger

```
public Range<T> Merge(Range<T> other)
{
    return ExtendUpperBound(other.Upper)
        .ExtendLowerBound(other.Lower);
}
```

Merge()

Join two ranges

Immutable Types

Queries

Functional, Predictable

Always return same result

Commands

Always return an instance

Might be shared

```
public interface IMutableStack<T>
{
    T Peek { get; }
    bool IsEmpty { get; }
    int Count { get; }
    void Push(T value);
    T Pop();
}
```

```
public interface IImmutableStack<T>
{
    T Peek { get; }
    bool IsEmpty { get; }
    int Count { get; }
    IImmutableStack<T> Push(T value);
    IImmutableStack<T> Discard();
}
```

```
public class ImmutableStack<T> : IImmutableStack<T>
{
    public T Peek { get; }

    public int Count { get; }

    private readonly IImmutableStack<T> _rest;

    public ImmutableStack(
        T value, IImmutableStack<T> rest)
    {
        Peek = value;
        _rest = rest;
        Count = rest.Count + 1;
    }

    public bool IsEmpty => false;

    public IImmutableStack<T> Push(T value)
        => new ImmutableStack<T>(value, this);

    public IImmutableStack<T> Discard() => _rest;
}
```

```
public class EmptyStack<T> : IImmutableStack<T>
{
    public T Peek
    {
        get { throw new InvalidOperationException(); }
    }

    public bool IsEmpty => true;

    public int Count => 0;

    public IImmutableStack<T> Push(T value)
        => new ImmutableStack<T>(value, this);

    public IImmutableStack<T> Discard()
    {
        throw new InvalidOperationException();
    }
}
```

Immutable Types

Simple

State defined on creation
Functional behaviour

Safe

Consistent behaviour
No side effects

Efficient

Shared data, cacheable
Can be faster

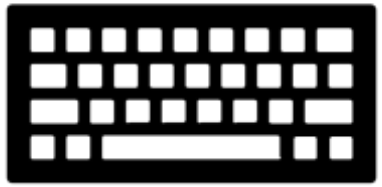
Testable

Functional style makes
testing easier

Thanks!

Bevan.Arps@outlook.com
@unrepentantgeek
www.nichesoftware.co.nz

Created by Mateo Zlatar
from Noun Project



Created by useiconic.com
from Noun Project

Created by blackspike
from Noun Project

Created by Cédric Stéphane Touati
from Noun Project

Image Credits