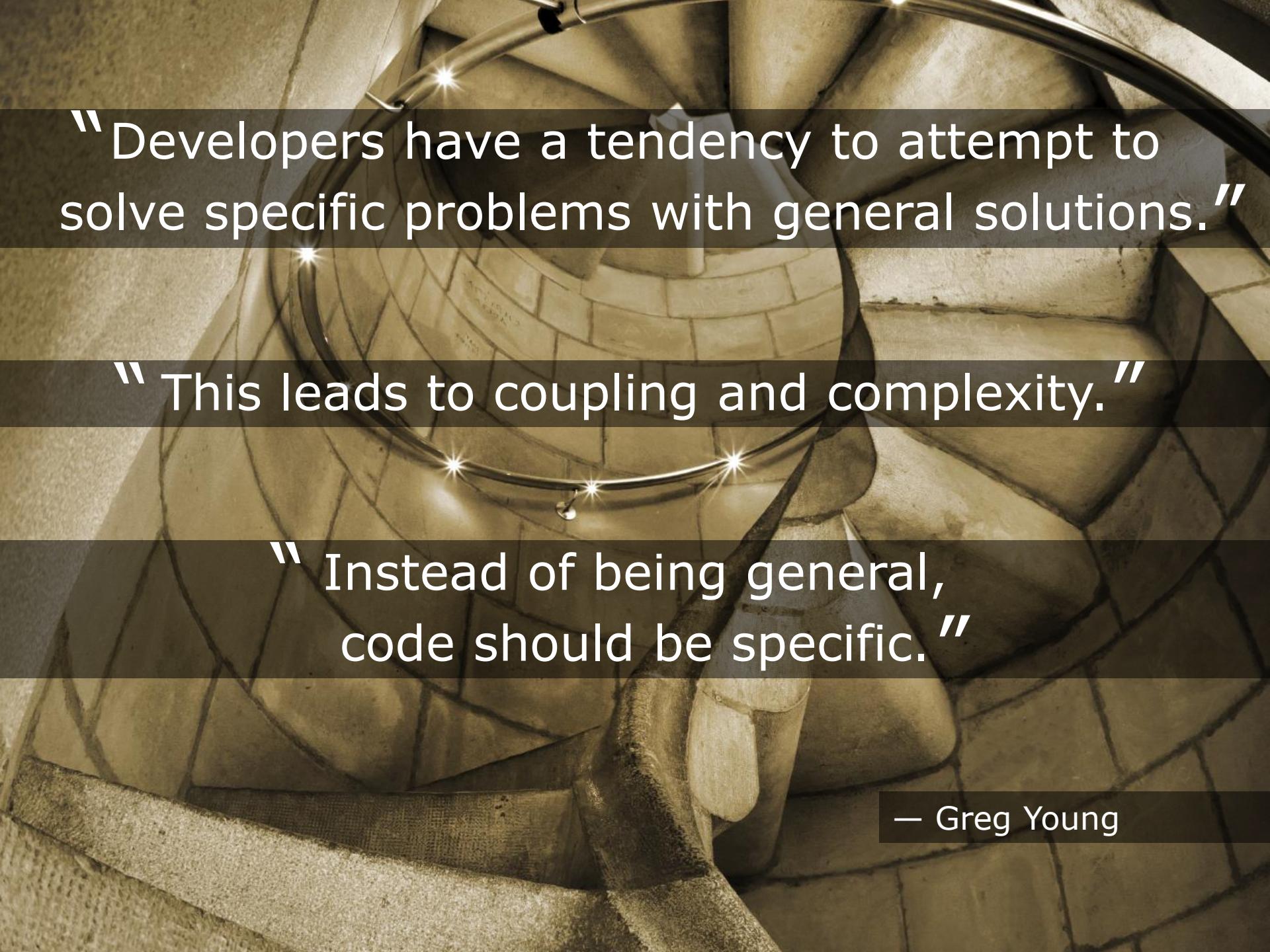


SOLID concrete

The background of the image is a dark, solid black. Overlaid on this background is a complex, abstract pattern of white and light blue smoke or steam. The smoke is highly detailed, showing intricate swirls and delicate wisps that fill the frame. It appears to be coming from the bottom left and rising towards the top right, creating a sense of movement and depth.

Needless
complexity



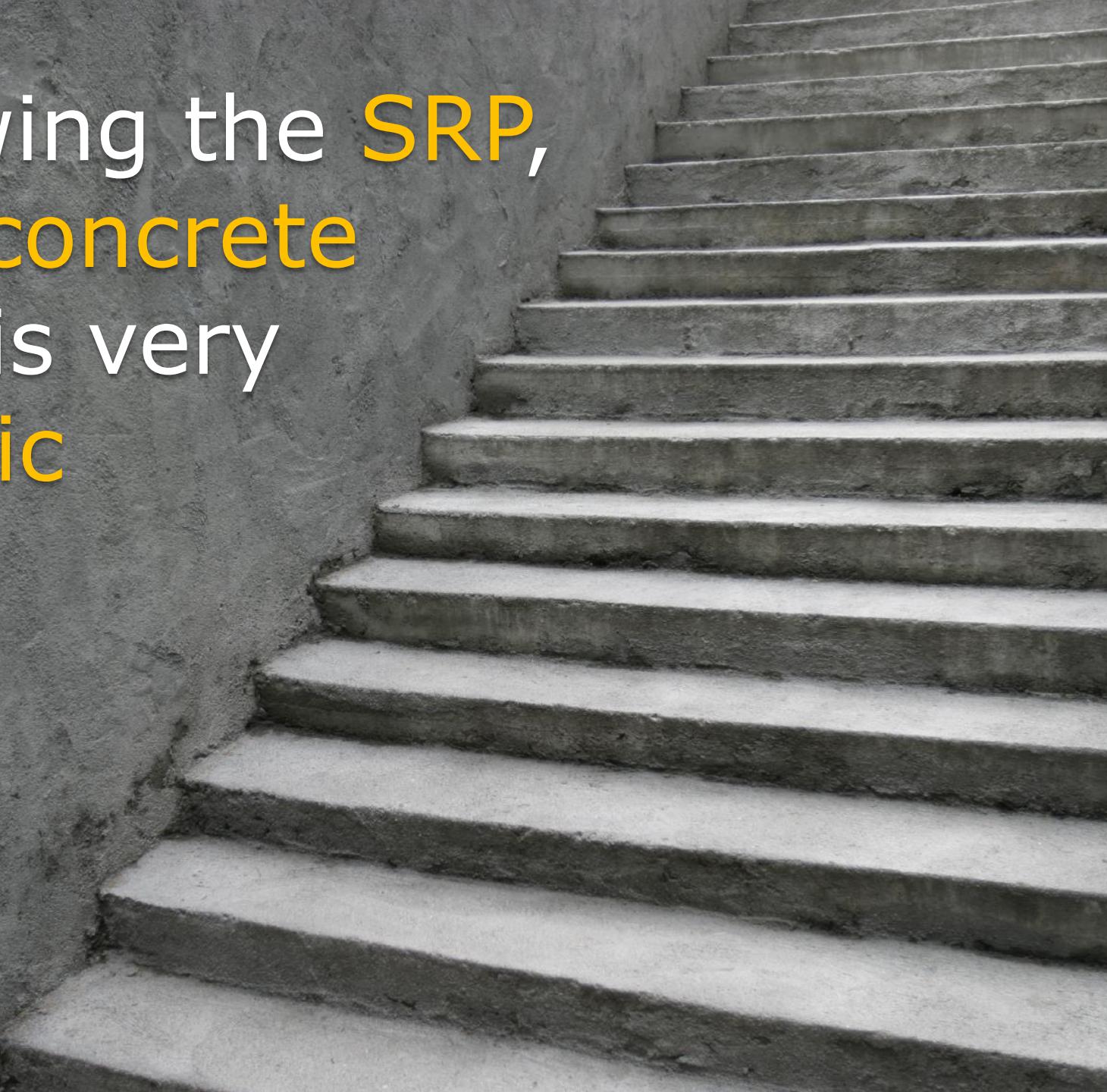
“Developers have a tendency to attempt to solve specific problems with general solutions.”

“This leads to coupling and complexity.”

“Instead of being general, code should be specific.”

— Greg Young

Following the SRP,
each concrete
class is very
specific



```
public class FileStore
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```



What if you need generality?

```
public class FileStore : IStoreWriter
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

```
public class FileStore : IStoreWriter, IStoreReader
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

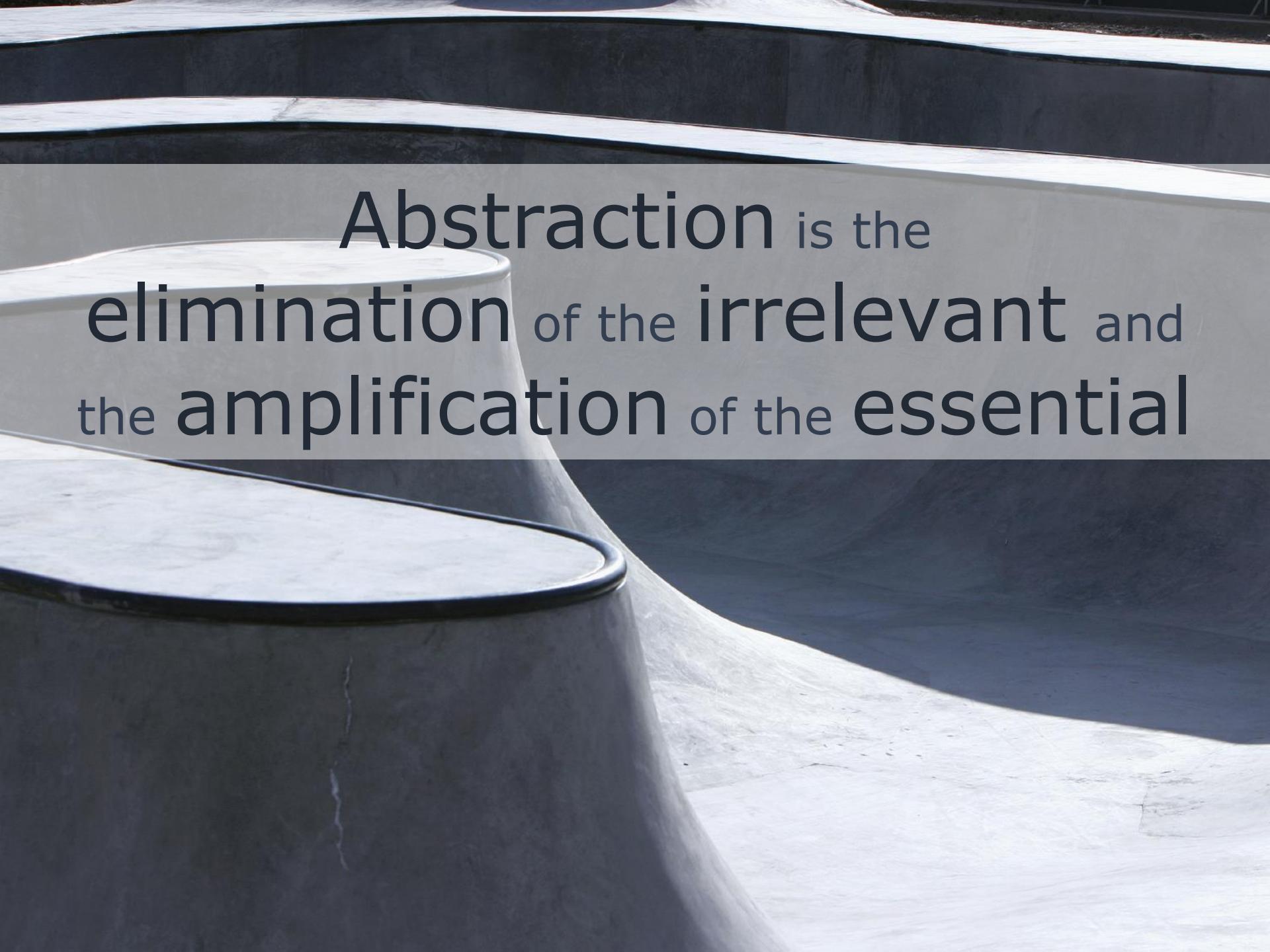
    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

```
public class FileStore : IStoreWriter, IStoreReader, IFileLocator
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

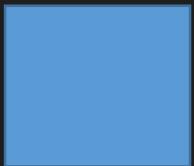
    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

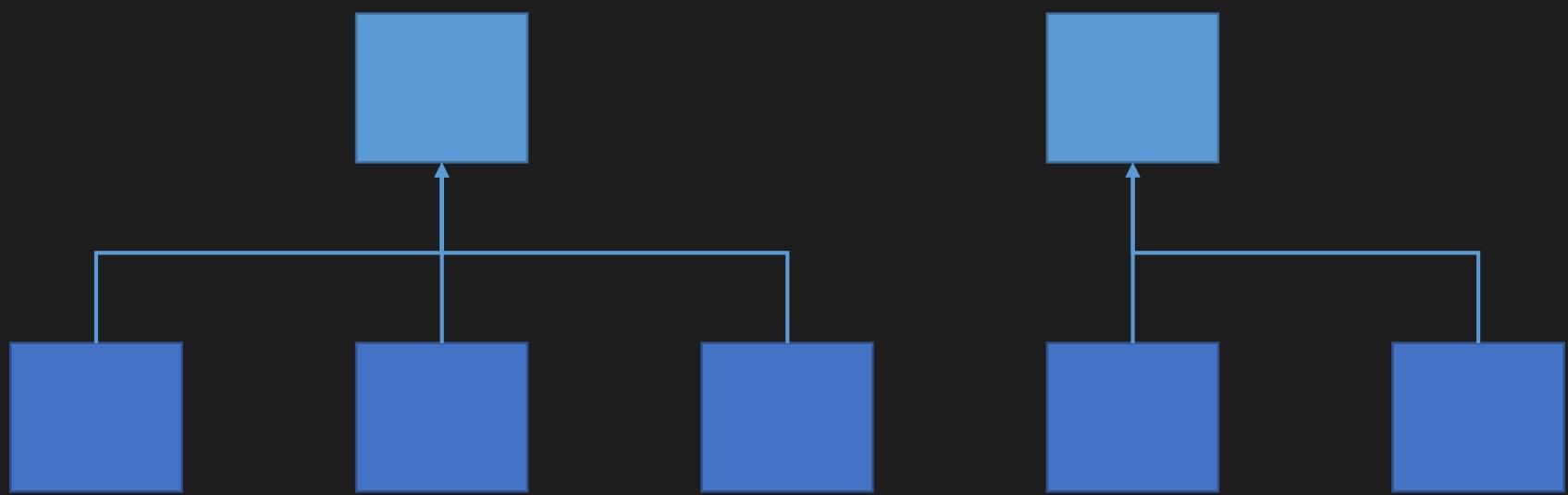
    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

Reused Abstractions Principle



Abstraction is the
elimination of the irrelevant and
the amplification of the essential







Start with **concrete behaviour**



Discover abstractions
as commonality emerges

A photograph of three large, cylindrical concrete cooling towers standing side-by-side against a bright blue sky with scattered white clouds. The towers have a light beige or cream-colored surface with vertical ribbing and a red and white striped band near the top. The central tower is the largest and most prominent, while the two flanking towers are slightly smaller.

Rule
of three

Open Closed Principle



Open for extensibility

Closed for modification

Bug fixing OK





Inheritance



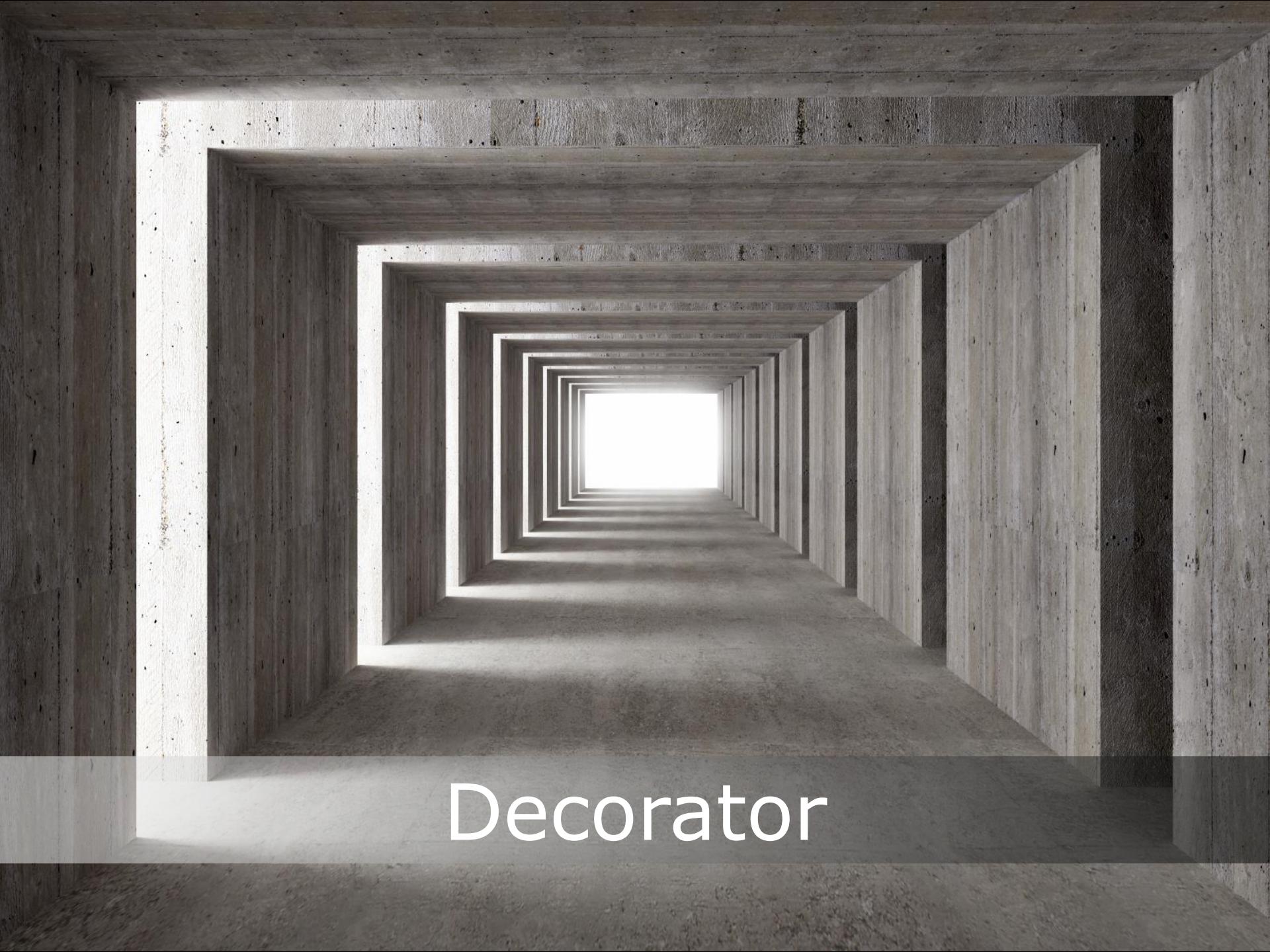
Favour **Composition** over inheritance

A photograph of a long, dark tunnel, likely a subway or high-speed rail track. The tunnel walls are made of light-colored concrete and feature a series of recessed lights along the ceiling. The floor is a smooth, light-colored surface. In the distance, the tunnel curves to the left, revealing more of the track and some equipment. The overall atmosphere is dim and industrial.

Strategy

A large stack of concrete pipes, likely manholes or culverts, is shown in an industrial or construction setting. The pipes are stacked in several layers, filling the frame. They are made of a light-colored concrete and have a smooth, slightly curved exterior. In the background, there are more pipes and some industrial structures under a clear sky.

Composite

A perspective view of a concrete corridor. The corridor is defined by multiple nested rectangular frames, creating a sense of depth and leading the eye towards a bright light at the far end. The walls and ceiling are made of rough, textured concrete. The floor is a lighter shade of concrete.

Decorator

```
public class FileStore
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

```
public class FileStore
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

```
public class FileStore
{
    public virtual void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public virtual string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public virtual FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

```
public class StoreCache
{
    private readonly ConcurrentDictionary<int, string> cache;

    public StoreCache()
    {
        this.cache = new ConcurrentDictionary<int, string>();
    }

    public void AddOrUpdate(int id, string message)
    {
        this.cache.AddOrUpdate(id, message, (i, s) => message);
    }

    public string GetOrAdd(int id, Func<int, string> messageFactory)
    {
        return this.cache.GetOrAdd(id, messageFactory);
    }
}
```

```
public class StoreCache
{
    private readonly ConcurrentDictionary<int, string> cache;

    public StoreCache()
    {
        this.cache = new ConcurrentDictionary<int, string>();
    }

    public void AddOrUpdate(int id, string message)
    {
        this.cache.AddOrUpdate(id, message, (i, s) => message);
    }

    public string GetOrAdd(int id, Func<int, string> messageFactory)
    {
        return this.cache.GetOrAdd(id, messageFactory);
    }
}
```

```
public class StoreCache
{
    private readonly ConcurrentDictionary<int, string> cache;

    public StoreCache()
    {
        this.cache = new ConcurrentDictionary<int, string>();
    }

    public virtual void AddOrUpdate(int id, string message)
    {
        this.cache.AddOrUpdate(id, message, (i, s) => message);
    }

    public virtual string GetOrAdd(int id, Func<int, string> messageFactory)
    {
        return this.cache.GetOrAdd(id, messageFactory);
    }
}
```

```
public class StoreLogger
{
    public void Saving(int id)
    {
        Log.Information("Saving message {id}.", id);
    }

    public void Saved(int id)
    {
        Log.Information("Saved message {id}.", id);
    }

    public void Reading(int id)
    {
        Log.Debug("Reading message {id}.", id);
    }

    public void DidNotFind(int id)
    {
        Log.Debug("No message {id} found.", id);
    }

    public void Returning(int id)
    {
        Log.Debug("Returning message {id}.", id);
    }
}
```

```
public class StoreLogger
{
    public void Saving(int id)
    {
        Log.Information("Saving message {id}.", id);
    }

    public void Saved(int id)
    {
        Log.Information("Saved message {id}.", id);
    }

    public void Reading(int id)
    {
        Log.Debug("Reading message {id}.", id);
    }

    public void DidNotFind(int id)
    {
        Log.Debug("No message {id} found.", id);
    }

    public void Returning(int id)
    {
        Log.Debug("Returning message {id}.", id);
    }
}
```

```
public class StoreLogger
{
    public virtual void Saving(int id)
    {
        Log.Information("Saving message {id}.", id);
    }

    public virtual void Saved(int id)
    {
        Log.Information("Saved message {id}.", id);
    }

    public virtual void Reading(int id)
    {
        Log.Debug("Reading message {id}.", id);
    }

    public virtual void DidNotFind(int id)
    {
        Log.Debug("No message {id} found.", id);
    }

    public virtual void Returning(int id)
    {
        Log.Debug("Returning message {id}.", id);
    }
}
```

```
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    public MessageStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }

    public DirectoryInfo WorkingDirectory { get; private set; }

    public void Save(int id, string message)
    {
        this.log.Saving(id);
        var file = this.GetFileInfo(id);
    }
}
```

The background image shows a large, cylindrical concrete structure with multiple circular openings along its top edge. The structure appears to be in a state of disrepair, with visible rust and weathering. The lighting is dramatic, coming from the openings, creating bright highlights against the dark, textured concrete walls.

Factory

Method

```
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    public MessageStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }

    public DirectoryInfo WorkingDirectory { get; private set; }

    public void Save(int id, string message)
    {
        this.log.Saving(id);
        var file = this.GetFileInfo(id);
    }
}
```

```
public void Save(int id, string message)
{
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    this.fileStore.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message);
    this.log.Saved(id);
}

public Maybe<string> Read(int id)
{
    this.log.Reading(id);
    var file = this.GetFileInfo(id);
    if (!file.Exists)
    {
        this.log.DidNotFind(id);
        return new Maybe<string>();
    }
    var message = this.cache.GetOrAdd(
        id, _ => this.fileStore.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(message);
}
```

```
        g( );
    return new Maybe<string>(message);
}

public FileInfo GetFileInfo(int id)
{
    return this.fileStore.GetFileInfo(
        id, this.WorkingDirectory.FullName);
}

}
```

```
        g( ) );
    return new Maybe<string>(message);
}

public FileInfo GetFileInfo(int id)
{
    return this.fileStore.GetFileInfo(
        id, this.WorkingDirectory.FullName);
}

protected virtual FileStore Store
{
    get { return this.fileStore; }
}

protected virtual StoreCache Cache
{
    get { return this.cache; }
}

protected virtual StoreLogger Log
{
    get { return this.log; }
}
}
```

```
        g( ) );
    return new Maybe<string>(message);
}

public FileInfo GetFileInfo(int id)
{
    return this.fileStore.GetFileInfo(
        id, this.WorkingDirectory.FullName);
}

protected virtual FileStore Store
{
    get { return this.fileStore; }
}

protected virtual StoreCache Cache
{
    get { return this.cache; }
}

protected virtual StoreLogger Log
{
    get { return this.log; }
}
}
```

```
        g( ) );
    return new Maybe<string>(message);
}

public FileInfo GetFileInfo(int id)
{
    return this.Store.GetFileInfo(
        id, this.WorkingDirectory.FullName);
}

protected virtual FileStore Store
{
    get { return this.fileStore; }
}

protected virtual StoreCache Cache
{
    get { return this.cache; }
}

protected virtual StoreLogger Log
{
    get { return this.log; }
}
}
```

Not the preferred way



Favour **Composition** over inheritance

When we're **done**,
we can make members **non-virtual** again

