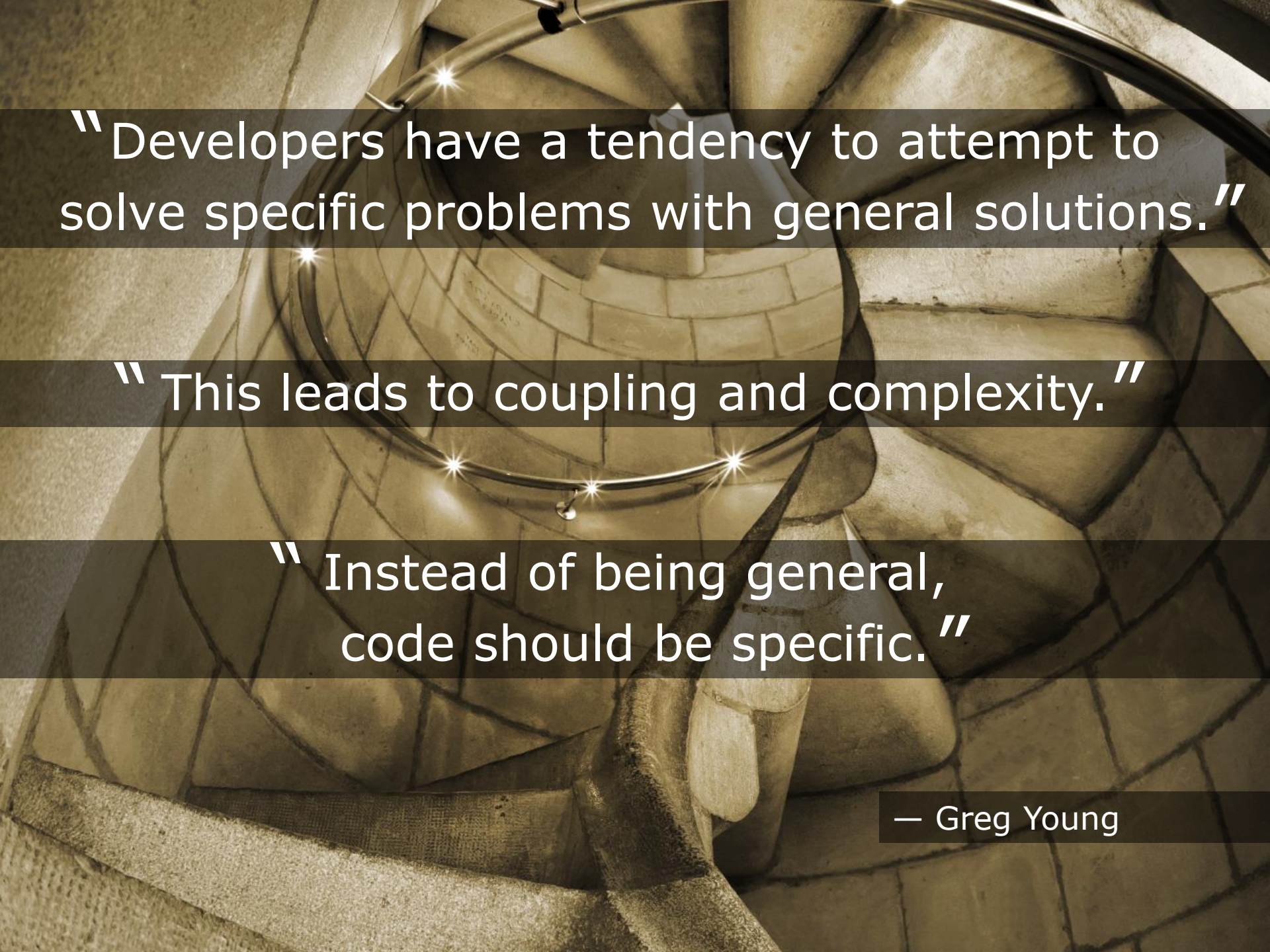
SOLID concrete


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"));
    }
}
```

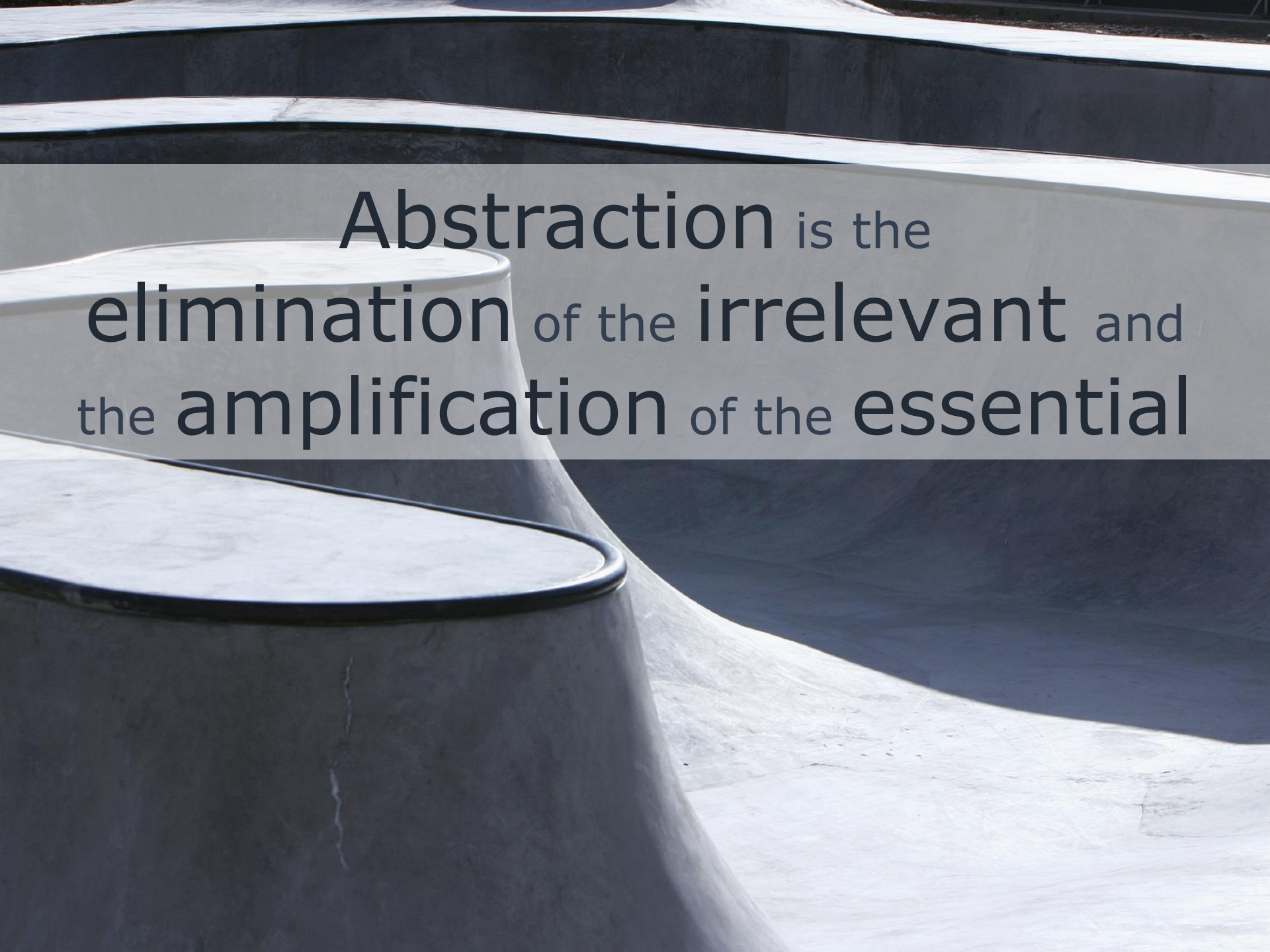

What if you need generality?

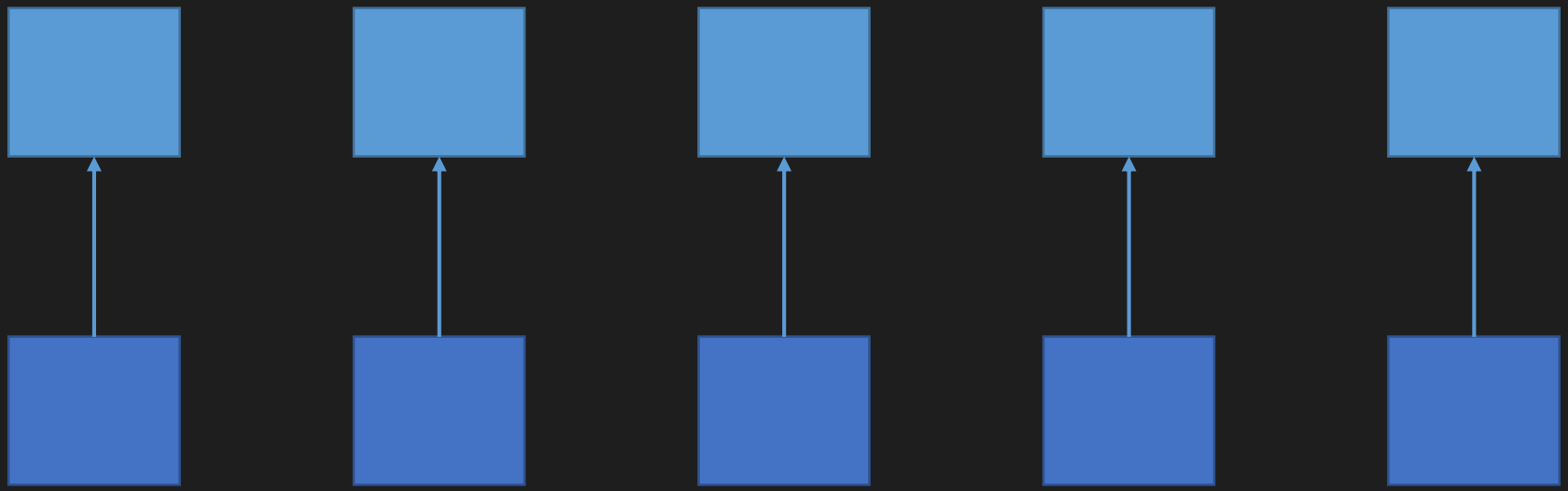
```
public class FileStore : IStoreWriter
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

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

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

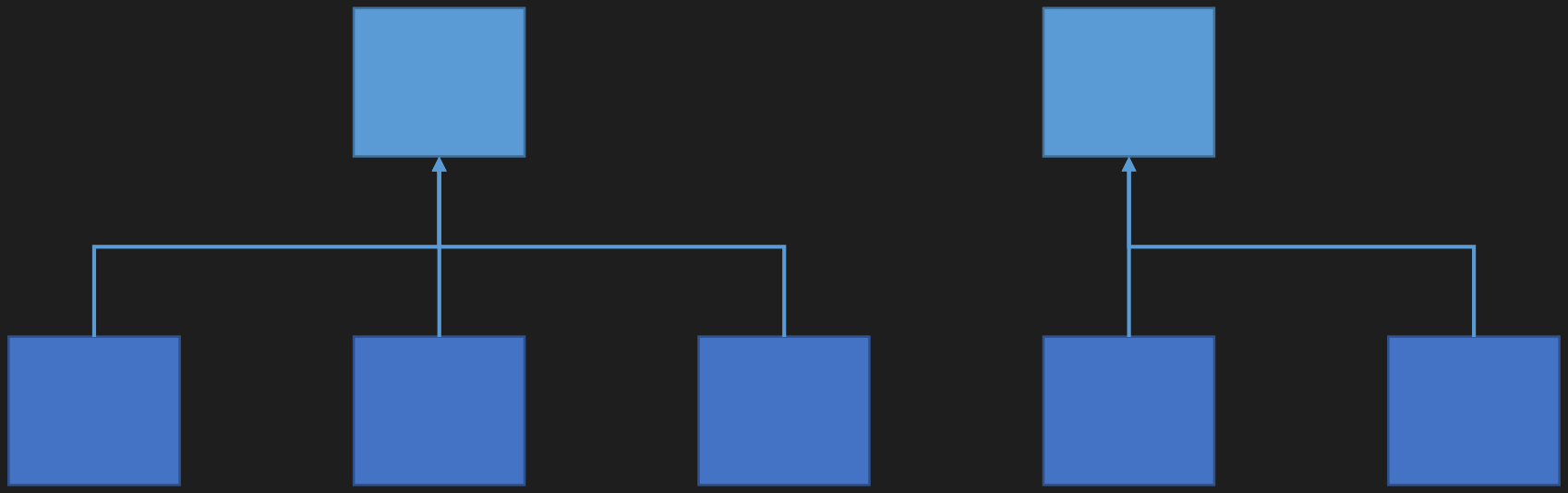
```
public class FileStore : IStoreWriter, IStoreReader
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    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


Rule
of three

Open Closed Principle


Open for extensibility
Closed for modification

Bug fixing OK


Inheritance

Favour Composition over inheritance


Strategy


Composite


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> messageFact
    {
        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);
```


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);
}
```

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

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

}
```

```
        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; }
    }
}
```

```
        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; }
    }
}
```

```
        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