Software Design Patterns

1 Elements of a Design Pattern

- **Pattern Name** - A handle which aims to encapsulate a design pattern, its solutions and any consequences arising from its use
- **Problem** - When the pattern can be applied, and may include any conditions which must be met before it makes sense to apply that pattern
- **Solution** - Provides an abstract description of the elements that make up the design as well as their relationships and responsibilities
- **Consequences** - The results and trade-offs from applying the pattern

2 The design pattern space

Broadly classify by purpose using three groups

- Creational patterns are concerned with the process of object creation
- Structural patterns are concerned with the composition of classes or objects
- Behavioural patterns characterise the ways in which classes or objects interact and distribute responsibility

Can also classify by scope

3 Some characteristics

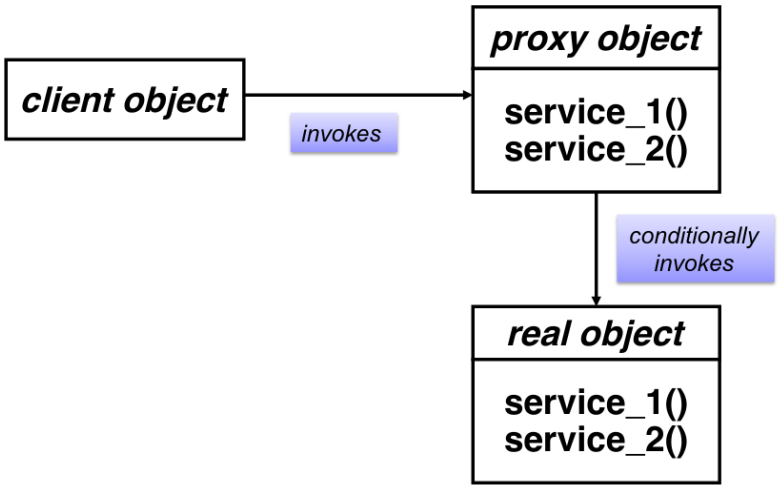
- Patterns tend to emphasise aggregation and interfaces over inheritance
- They aim to identify the parts of a system that are likely to change and to encapsulate those
- Patterns often aim for loose coupling, making for greater flexibility1

4 Caveat to using patterns

- They are only of use to experts
- Inappropriate use of patterns cause problems for system maintenance
- There is not a wide approval of patterns, people like to do things differently

5 Proxy

A proxy provides a surrogate or placeholder for another object to control access to it.



A proxy is applicable wherever there is a need for a more versatile (and indirect) reference to an object than just a pointer. Examples of where this might be appropriate include:

- **Remote proxy** - Provides a local representative for an object in a different address space
- **Virtual proxy** - creates ”expensive” objects on demand, with an example being the ”ImageProxy” described in the example of the word processor
- **Protection proxy** - Controls access to the original object, and is useful where users (objects) may have different access rights, depending on their role
- **Synchronisation Proxy** - Controls safe access to a subject from multiple threads

6 Design Anti-Patterns

- Design anti-patterns describe obvious, but wrong solutions to recurring problems
- Focus tends to be a little different - anti-patterns are apt to be described in terms of the reasons why wrong solutions are adopted, rather than their form

7 Code Smells

- These can be considered as a form of an anti-pattern
- A code smell is a design weakness that may form a future source of errors and that may contribute to the technical debt incurred when producing a design. Hence they act as a driver for refactoring
- Problem is to determine appropriate ”boundaries”

8 Documenting a design pattern

Major headings:

- Intent - what the pattern does
- Also Known As - Identifying any synonyms
- Motivation - why such a pattern is useful
- Applicability - Roles for the pattern
- Structure - Typically a class diagram
- Participants - Describing the classes in outline
- Collaborations - How the participants work together
- Consequences - benefits and liabilities of its use
- Implementation - pitfalls, hints, language-specific issues
- Sample code - usually C++
- Known uses - Examples of successful application
- Related patterns - similar ones and how they differ

9 Issues with design patterns

+ Attracted a strong body of support from a dedicated community of pattern enthusiasts
+ Provides a mechanism for reuse of design ideas, at least, for experts
+ Fairly standardised set of pattern names and description forms
? How many patterns can be realistically indexed and still be accessible to the designer
? Is the idea one that is readily transferable to other architectural styles beyond O-O
? Are they most likely to be effective when used for peer to peer exchange of knowledge

10 Note

Patterns need to be used with care:

- Patterns may create an unnecessary overhead in terms of adding classes and code - so there has to be a justification for their use
- Code creating using patterns may be hard to understand
- Avoid using the singleton pattern (ensure a class has only one instance, and provide a global point of access to it) - can be mis-used to provide persistent data in a system

Hence patterns may be more use in identifying the form that a design solution might take, then in providing details of how it is to be implemented