Material and some slide content from:
- Mehdi Amoui Kalareh
- Derek Rayside
- Steve Easterbrook
- David Budgen

Design Qualities

Reid Holmes

Software design

- Designers control risk by limiting unknown factors
  - e.g., to reduce risk they can:
    - Compose existing components in a novel way
    - Compose new components in a known way
- All software quality measures are relative
  - e.g., how would you measure the quality of a chair?
    - construction quality? aesthetic quality? fitness for purpose?

Software design

- Predominant quality concern: fitness for purpose
  - Does it do what it should?
  - Does it do it as the users require?
  - Is it reliable / safe / fast / secure enough?
  - Is it affordable? Will it be finished on time?
  - Can it be adapted in the future?
- Questions are intermingled between software and its domain

Measuring quality

The Quality Concepts
(abstract notions of quality properties)

Measurable Quantities
(define some metrics)

Counts taken from Design Representations
(realization of the metrics)

examples...

reliability	complexity	usability
mean time to failure?	information flow between modules?	time taken to learn how to use?
run it and count crashes per hour???	count procedure calls???	minutes taken for some user task???

Quality attributes

- Simplicity
  - “There are two ways of constructing a software design. One way is to make it so simple that there are no obvious deficiencies. And the other is to make it so complicated that there are no obvious deficiencies.” -- Hoare [1981]
  - Meets goals without extraneous embellishment
- Measured by its converse --> complexity

Coupling

Given two units *(e.g. methods, classes, modules, …)*, **A and B:**

Form	*Features*	*Desirability*
Data coupling	A & B communicate by simple data only	High (use parameter passing & only pass necessary info)
Stamp coupling	A & B use a common type of data	Okay (but should they be grouped in a data abstraction?)
Control coupling (activating)	A transfers control to B by procedure call	Necessary
Control coupling (switching)	A passes a flag to B to tell it how to behave	Undesirable (why should A interfere like this?)
Common environment coupling	A & B make use of a shared data area (global variables)	Undesirable (if you change the shared data, you have to change both A and B)
Content coupling	A changes B's data, or passes control to the middle of B	Extremely Foolish (almost impossible to debug!)

Cohesion

Source: van Vliet 1999, pp299-300 (after Yourdon & Constantine)

How well do the contents of a procedure *(module, package,…)* go together?

Form	*Features*	*Desirability*
Data cohesion	all part of a well defined data abstraction	Very High
Functional cohesion	all part of a single problem solving task	High
Sequential cohesion	outputs of one part form inputs to the next	Okay
Communicational cohesion	operations that use the same input or output data	Moderate
Procedural cohesion	a set of operations that must be executed in a particular order	Low
Temporal cohesion	elements must be active around the same time (e.g. at startup)	Low
Logical cohesion	elements perform logically similar operations (e.g. printing things)	No way!!
Coincidental cohesion	elements have no conceptual link other than repeated code	No way!!

Spotting incoherency

- An operation's description is full of 'and' clauses:
  - e.g.,
  - Results in temporal cohesion, logical cohesion
- An operation's description has many 'if..then..else'
  - e.g.,
  - Results in control coupling, coincidental cohesion, logical cohesion

Cognitive dimensions

- Premature commitment
  - decision made with insufficient data that constrains future choices

    - encouraging out-of-order decision making can help, locked-in process == undesirable
- Hidden dependencies
  - some deps may be obvious (e.g., static), but others may be latent (e.g., temporal)
- Secondary notation
  - non-obvious relationships may be meaningful or provide context (why patterns good)
- Viscosity
  - resistence to change

Design Principles

- Some high-level advice exists in the form of principles that can help guide design decisions.
- SOLID represents common subset of these:
  - **S**ingle Responsibility
  - **O**pen/Close
  - **L**iskov Substitution Principle
  - **I**nversion of Control
  - **D**ependency Inversion

Design principles

- Single Responsibility
  - Classes should have only one major task
  - Insulates classes from one another
- Open/Close
  - Classes should be open for extension but closed to modification
  - If a class needs to be extended, try to do it through subclassing to minimize impact on existing clients

Design principles

- Liskov substitution principle
  - Subtypes should behave as their parent types
  - aka a program should still behave correctly should two subtypes of a common be interchanged
- Interface segregation
  - Only place key methods in interfaces
  - Clients should not need to support methods that are irrelevant to their behaviour
  - This can lead to a larger number of smaller interfaces in practice

Design principles

- Dependency inversion
  - Also known as the ‘hollywood principle’ or ‘inversion of control’
  - High-level methods should not depend on lower-level modules
  - Minimizes direct coupling between concrete classes

Lower-level principles

- Encapsulate what varies
  - This is a key concern to increase reusability and reduce the impact regression bugs
- Program to interfaces, not implementations
  - Reduces coupling between classes
- Favour composition over inheritance
  - Enables runtime behaviour changes and makes code easier to evolve in the future
- Strive for loose coupling

Why design patterns?

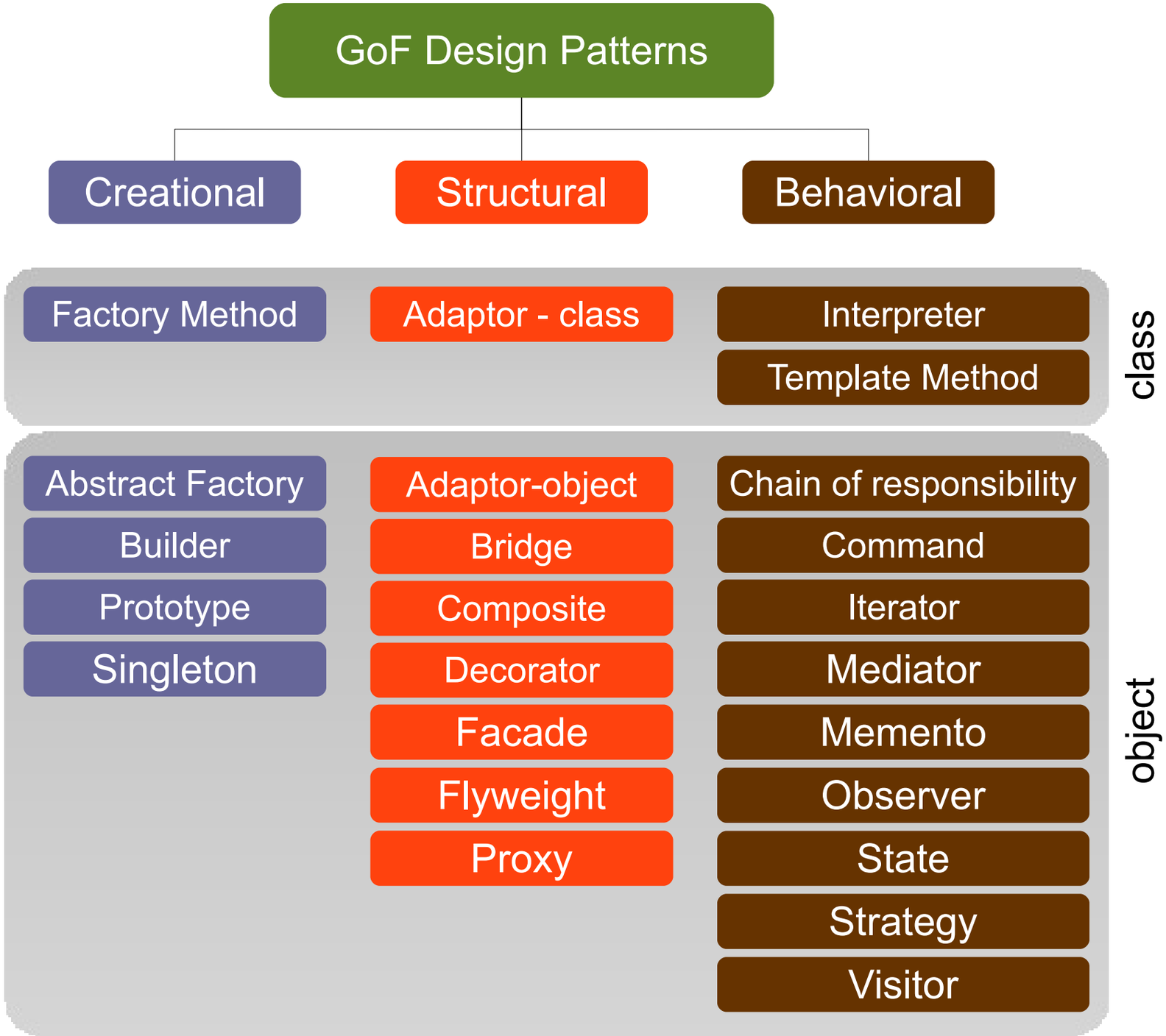
Ease communication by using a shared vocabulary

Enhance flexibility for future change

Leverage existing design knowledge

Increase reusability of developed code

GoF design patterns



Design patterns

- Design patterns are:
  - Common solutions to a recurring design problems.
  - Abstract recurring structures.
  - Comprises of class and/or object:
    - Dependencies
    - Structures
    - Interactions
    - Conventions
  - Names the design structure explicitly.
  - Distills design experience.

Design patterns

- Design patterns have four main parts:
  1. Name
  2. Problem
  3. Solution
  4. Consequences / trade-offs
- Are language-independent.
- Are “micro-architectures”
- Cannot be mechanically applied
  - Must be translated to a context by the developer.

Henri Poincaré

"Science is built up of facts as a house is built of stones, but an accumulation of facts is no more a science than a heap of stones is a house."