Material and some slide content from:
- Software Architecture: Foundations, Theory, and Practice
- Krzysztof Czarnecki
- John Vlissides
- GoF Design Patterns Book
- Atif Khan

Design Introduction

Reid Holmes

Object-oriented design

References:

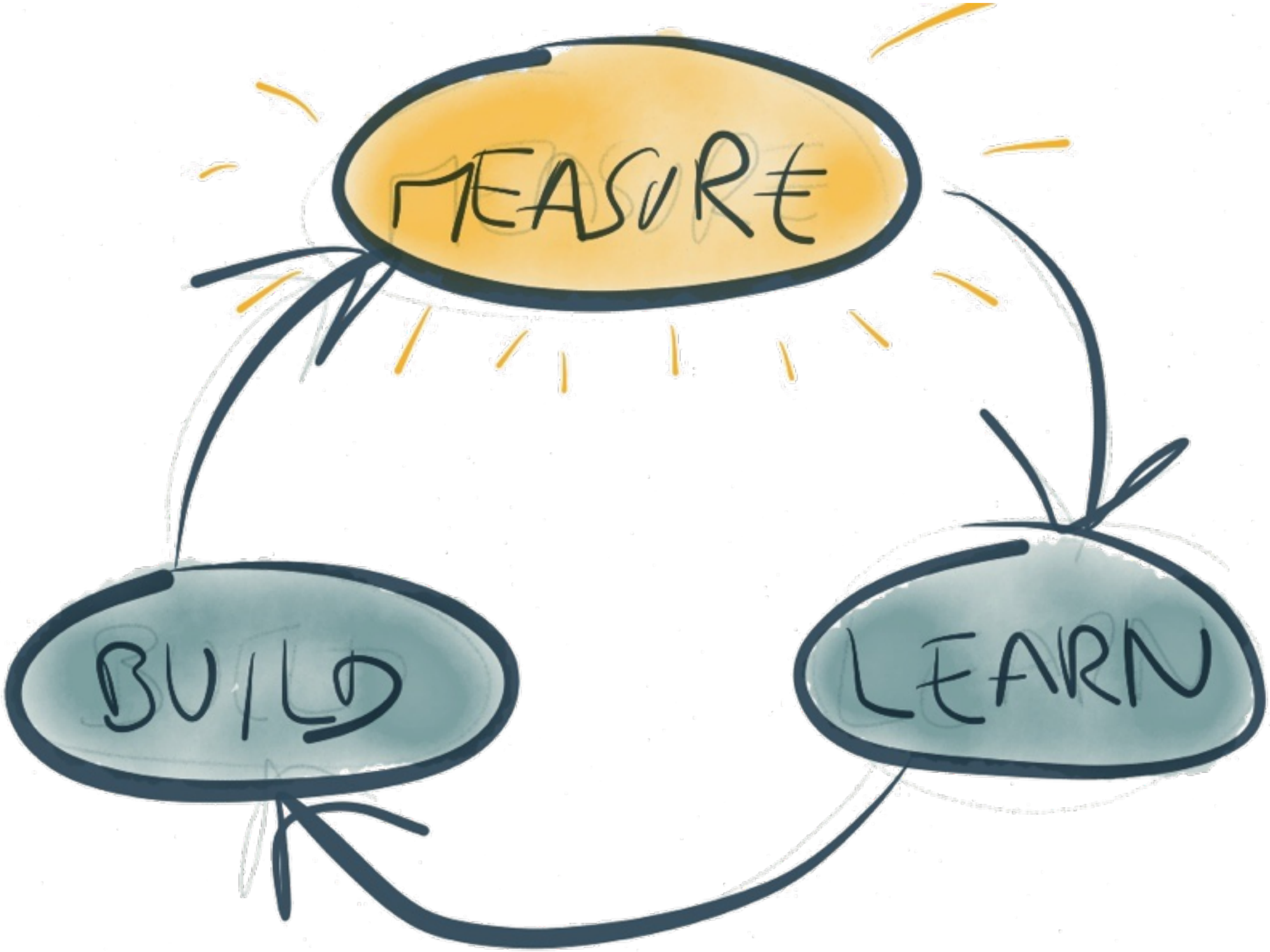
Design Patterns: Elements of Reusable Object-oriented Software by Gamma, Helm, Johnson, and Vlissides

Design of Design. by Fred P. Brooks Jr.

Analysis vs. design

- Analysis
  - Domain-level modelling of “real world” objects.
    - Addresses functional challenges.
    - Captures what a system does.
    - Provides implementation guidance.
- Design
  - Modelling of implementation objects.
    - Addresses implementation challenges.
    - Captures how the system realizes its OOA.
    - Assigning responsibilities to objects.

Design process



Christopher Jones

"If, as is likely, the act of tracing out the intermediate steps exposes some unforeseen difficulties... the designers are thrown back to square one."

Motivation

- OOD methods emphasize design notations.
  - But... these notations must be expressible in code.
- The importance of experience in OOD cannot be overemphasized.
- Design-level reuse is valuable.
  - Matches problems to design experience.
  - Avoid previously-encountered difficulties.

Concept

- OO systems exploit recurring design structures that promote:
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance
- Capturing, communicating, and applying this knowledge is problematic
- Must contend with similar constraints as architecture (e.g., complexity, conformity, changeability, invisibility)

Abstraction

- The removal of detail while retaining essential properties of its structure
- Plays a central role in the design process:
  - Enables the designer to focus on the key issues without being distracted by implementation
- It can be easy for developers to be distracted by implementation minutiae
- Different abstractions are appropriate for different applications and needs

Static vs. Dynamic

- Is it enough that the code compiles?
- Is it sufficient that it meets its specified structure?
- Behaviour matters. Static relationships are only a subset of a complete system
- Behaviour is inherently dynamic
  - The code alone may not be sufficient
  - Debugging only gives glimpses in time
- Increased abstraction == decreased control

Design principles

- System designs balance a variety of concerns
- Design principles provide a set of considerations to keep in mind when modelling various dependencies in a design
- There is no one set of principles designs should strive to support; here we describe a set of five high-level principles
- Dependency management heavily influences the evolvability, reusability, and brittleness of a system

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
  - These dependencies often manifest during object creation

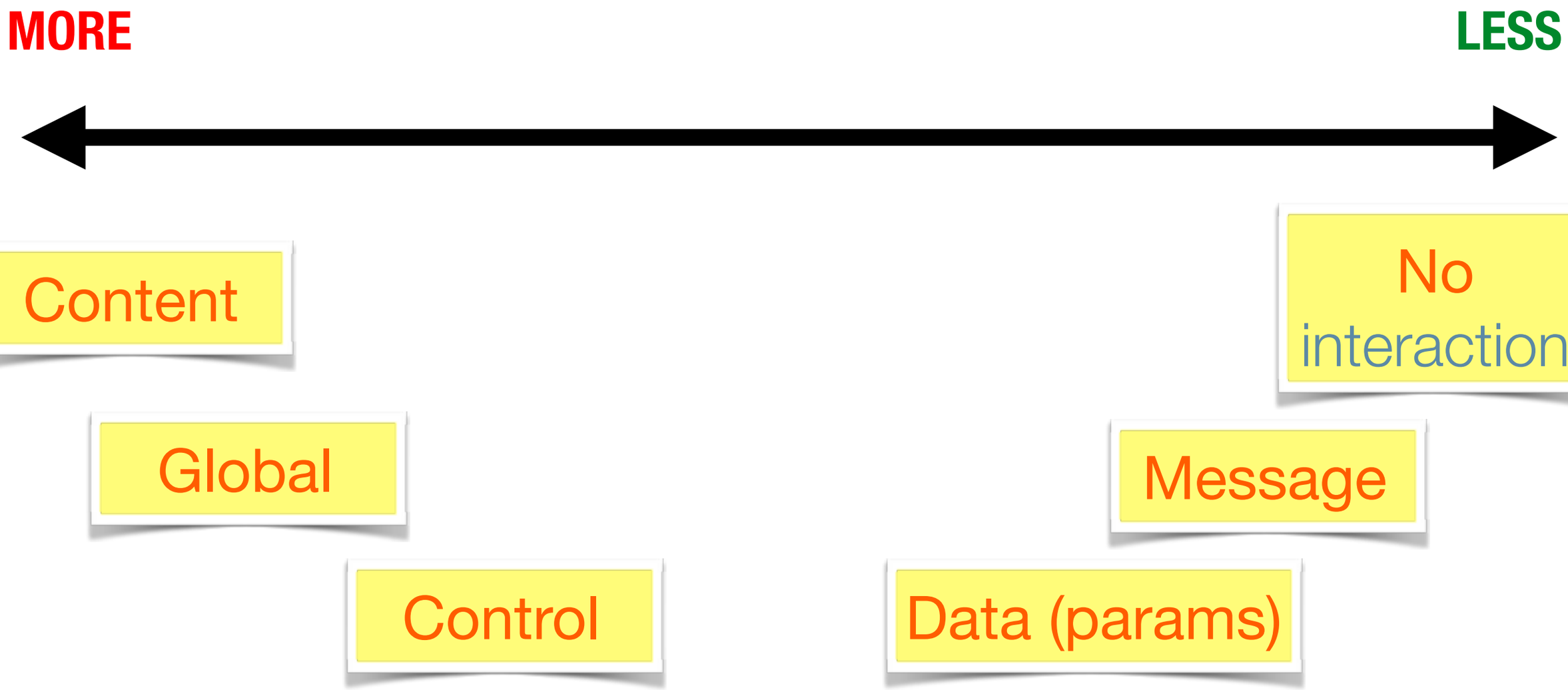
Lower-level principles

- 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

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


MORE
LESS
Content
Global
Control
Data (params)
Message
No interaction

Cohesion

WORSE **BETTER**

Coincidental

Functional

Logical

Sequential

Temporal

Communication

Spotting incoherency

- An operation's description is full of 'and' clauses:
  - e.g., 'initialize the data structure and initialize the screen and initialize the history and initialize the layout and show the spash screen'
  - Results in temporal cohesion, logical cohesion
- An operation's description has many 'if..then..else'
  - e.g., 'if x==0 do foo else if x == 1 then do bar, else if x == 2 baz else do bah.'
  - Results in control coupling, coincidental cohesion, logical cohesion

Cognitive dimensions

- Premature commitment
  - decision made with insufficient data that constrains future choices
- Hidden dependencies
  - some dependencies 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
  - resistance to change

Why design patterns?

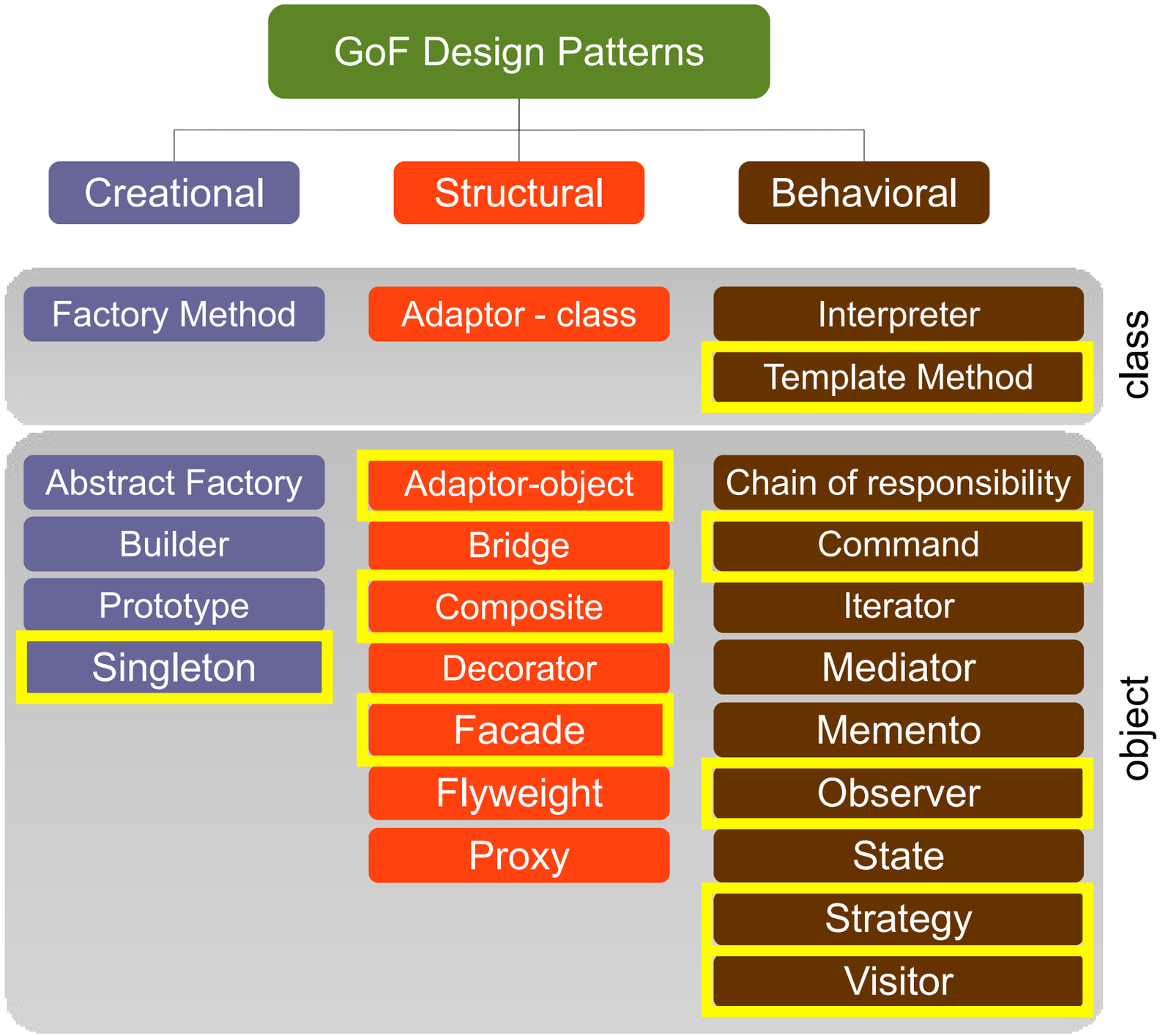
Ease communication by using a shared **vocabulary**

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."