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

Design Introduction

Mei Nagappan

Software Design

- Wikipedia: Software design is the process of implementing software solutions to one or more sets of problems.
- Jack W. Reeves (What is software design?)
  - If the design documents truly represent a complete design, the manufacturing team can proceed to build the product…. After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings.

From the Engg. World

- There are common problems that one faces during development.
- There can be common solutions to them.
- These are not specific problems like sorting, which has an algorithm.
- Deals with how you create classes, methods, and objects (at least in the OO world).
- The solution is called design patterns.
- Note software design is an overloaded term.

Design patterns

- 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.
- Sounds a lot like Architecture.

Architecture vs Design

- Arch
  - Higher level of abstraction
  - How modules talk to each other
- Design
  - Lower level of abstraction (meaning more concrete solutions)
  - How is a particular module structured?
- All design is Arch but not other way around.

Why design patterns?

Leverage existing design knowledge

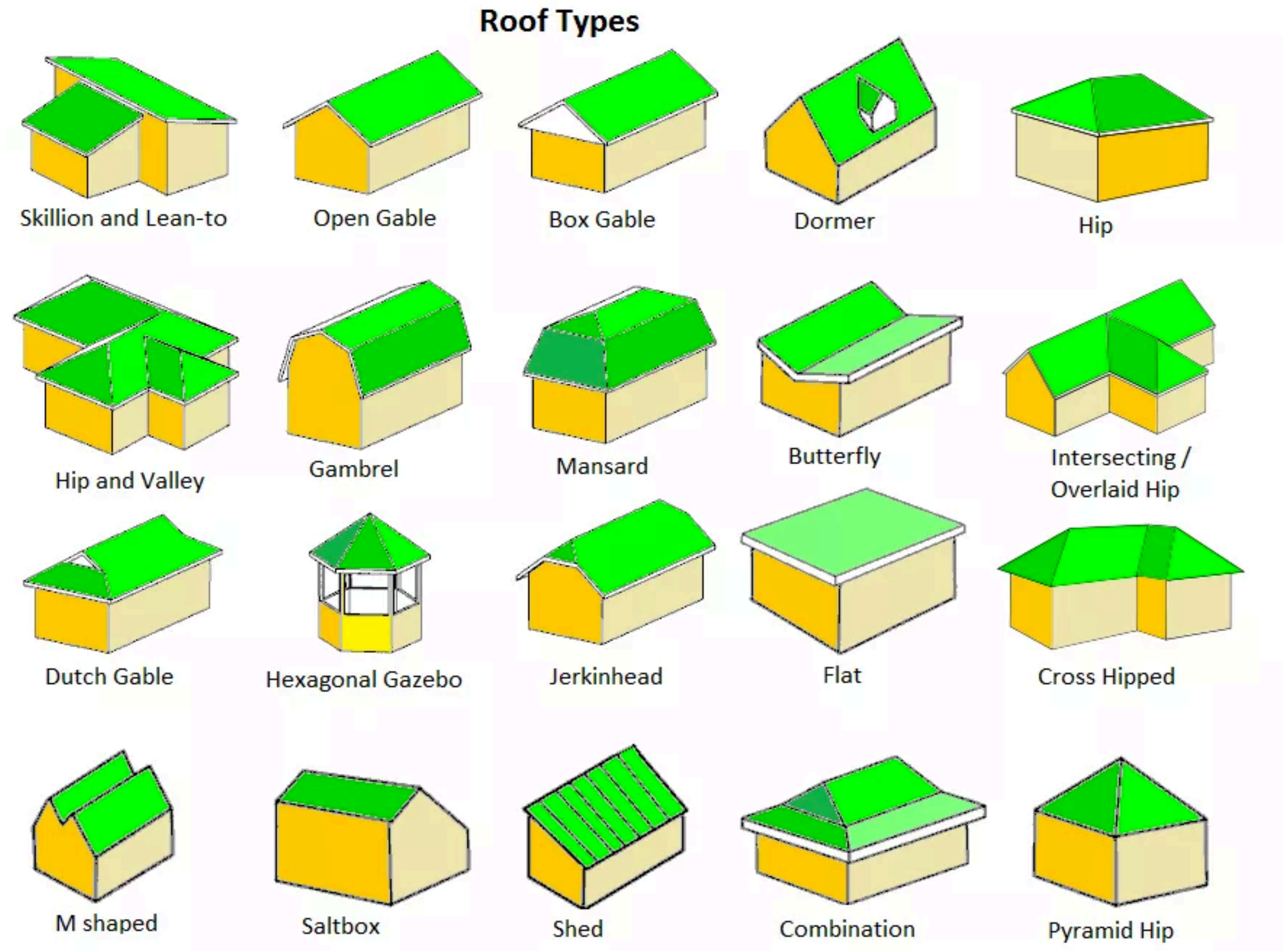
Increase reusability of developed code

Ease communication by using a shared vocabulary

Enhance flexibility for future change

Leverage existing design knowledge

- Other people have faced similar situations.
- Eg. Building a house.
  - Roofs in houses have been around for 1000s of years



https://i1.wp.com/www.roofcalc.net/wp-content/uploads/2014/06/Roof-Types-Diagram.png

Increase reusability

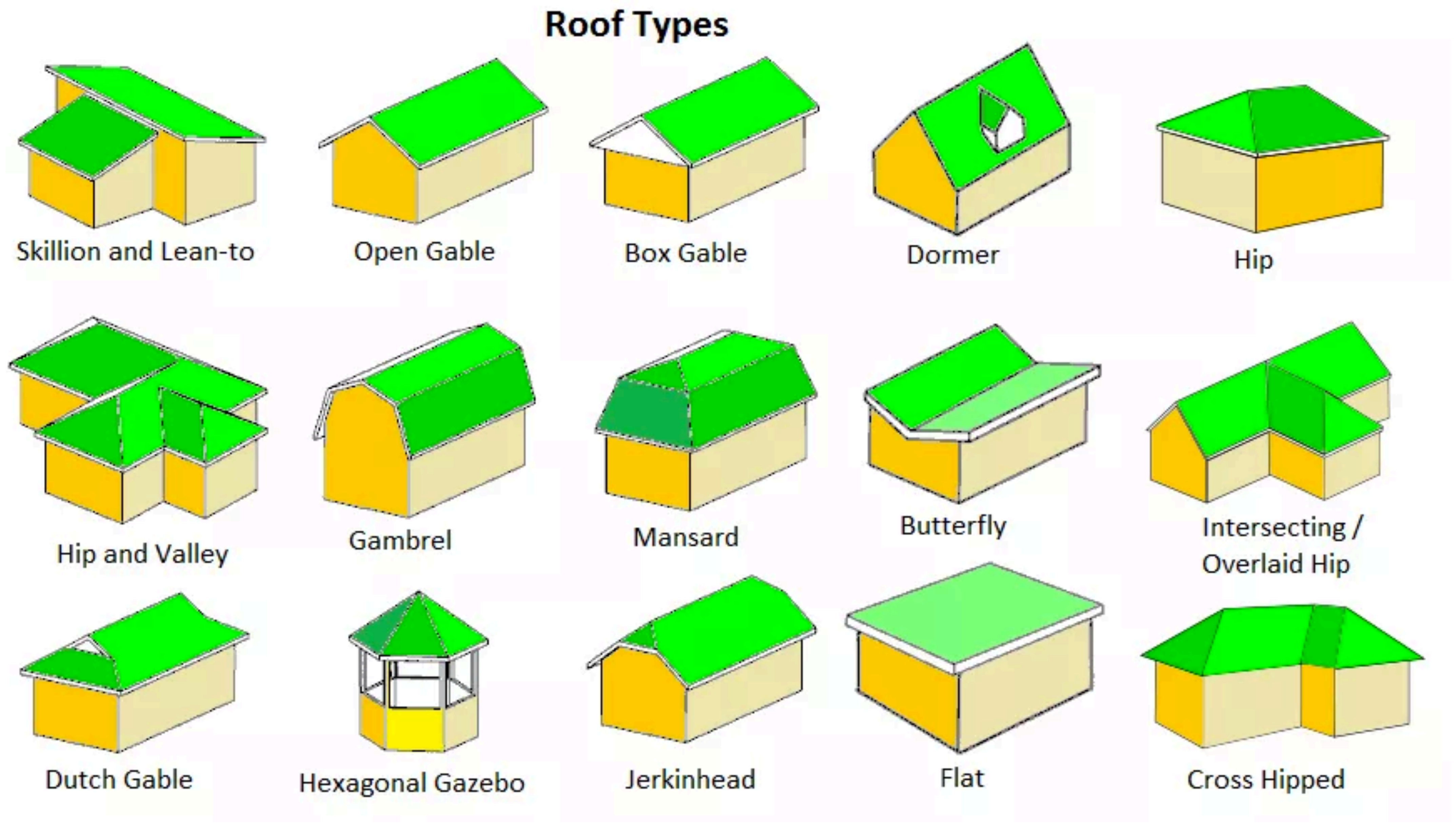
- If I have a problem with a known design solution, and I find an implementation with the same design then I can reuse the code
- Note: Cannot reuse all of it all the time, as is.
  - In the roof example, I can reuse parts of a roof from another house, but not all of it, exactly.

Ease of communication using a shared vocabulary

- Different stakeholders even in the development sphere
  - Dev, Tester, Maintainer, Rel Engg.
- Many in each of the stakeholders as well
- When I ask another person to implement a design, they know exactly what to do
- When a different stakeholder looks at the code, they know what design was implemented, and therefore the rationale.
- Eg. As a home owner, when I request a gable roof, the builder knows exactly what I mean.

Enhance flexibility for change

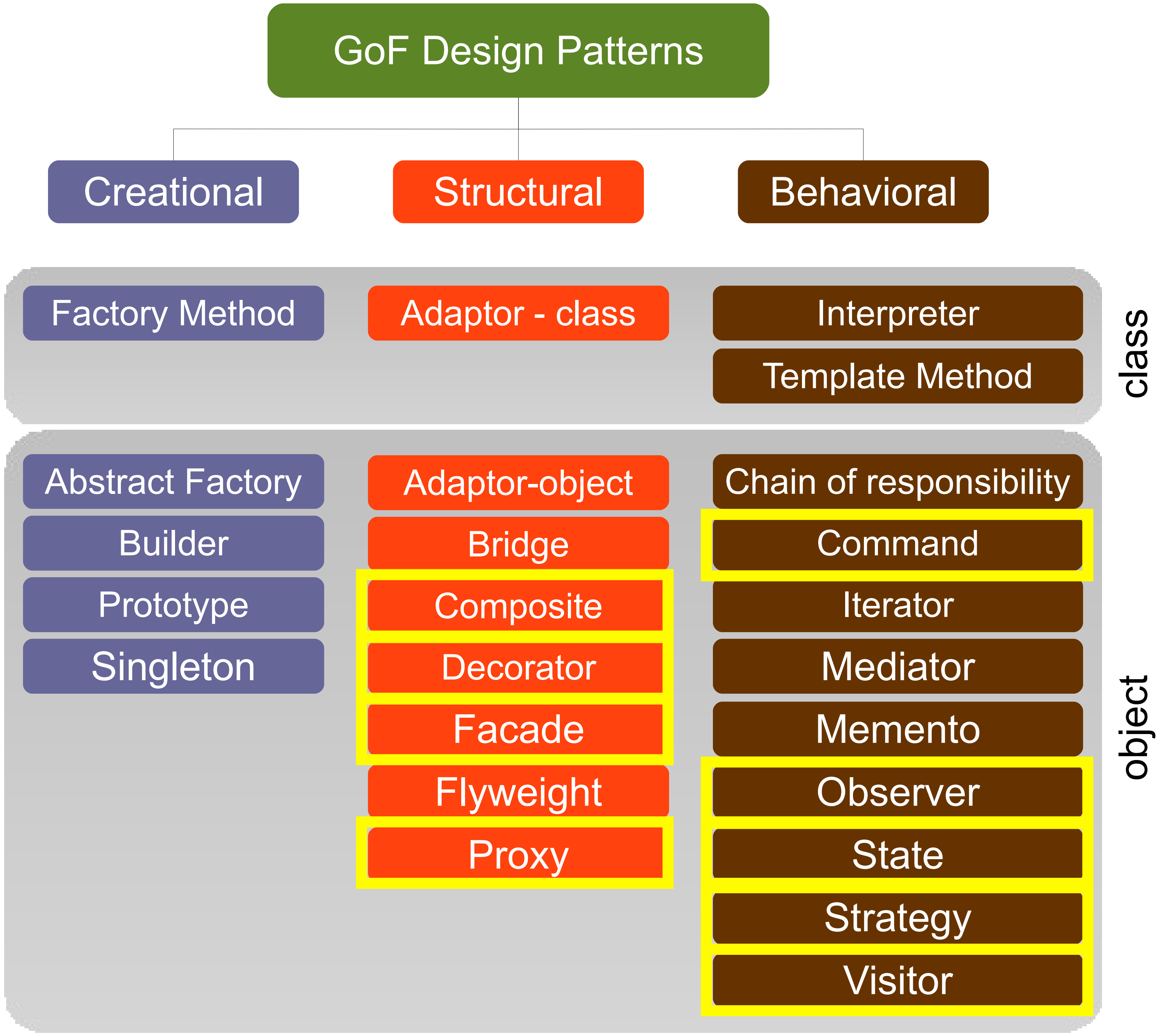
- When maintainer looks at the code, and design choices, they know what changes they can make without breaking the design.



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.

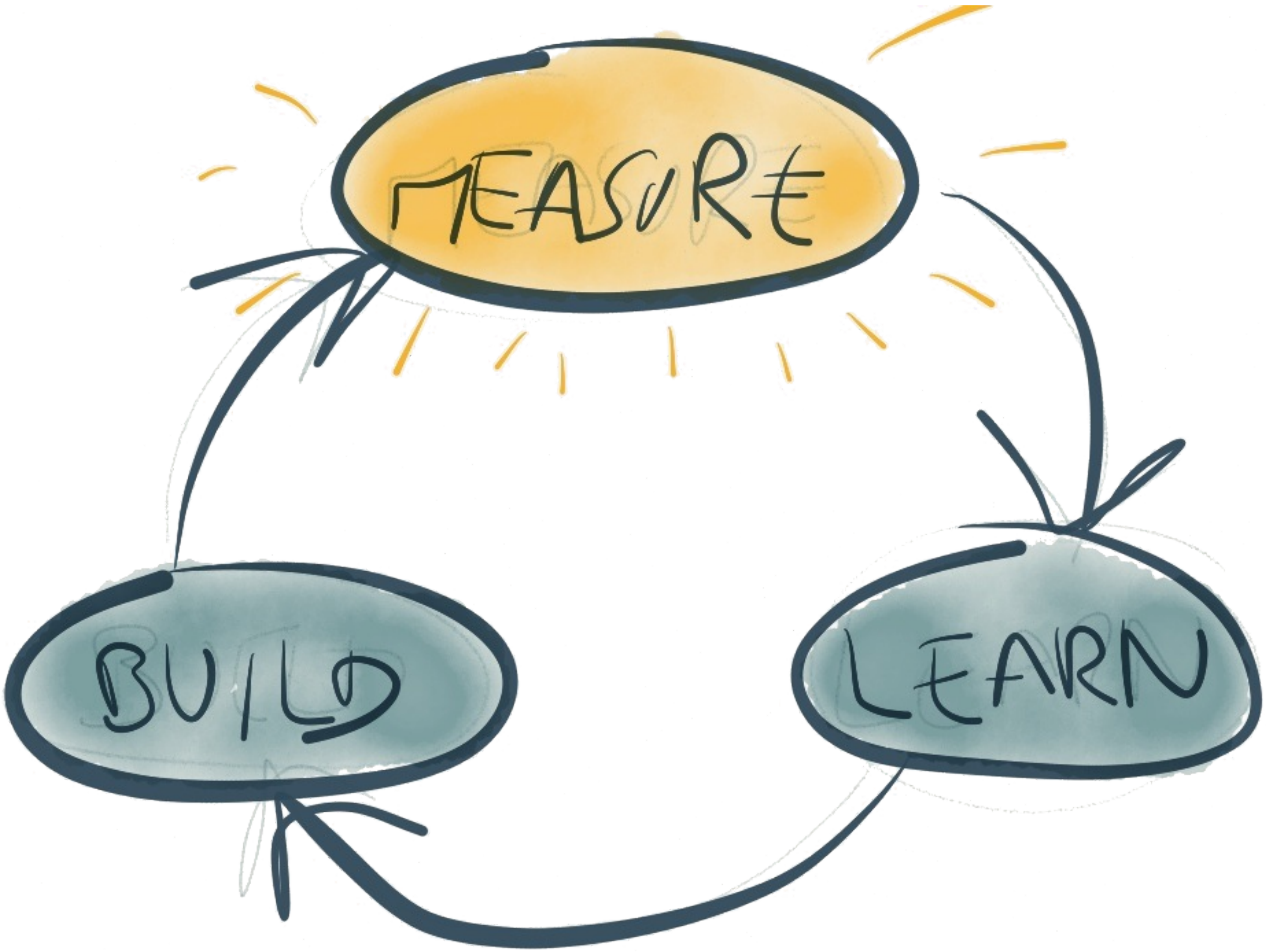
GoF design patterns



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.”

Design process



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.
- Good design can be marred by poor implementation, but good implementation cannot overcome bad design

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

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**nterface Segregation
  - **D**ependency Inversion

Single Responsibility

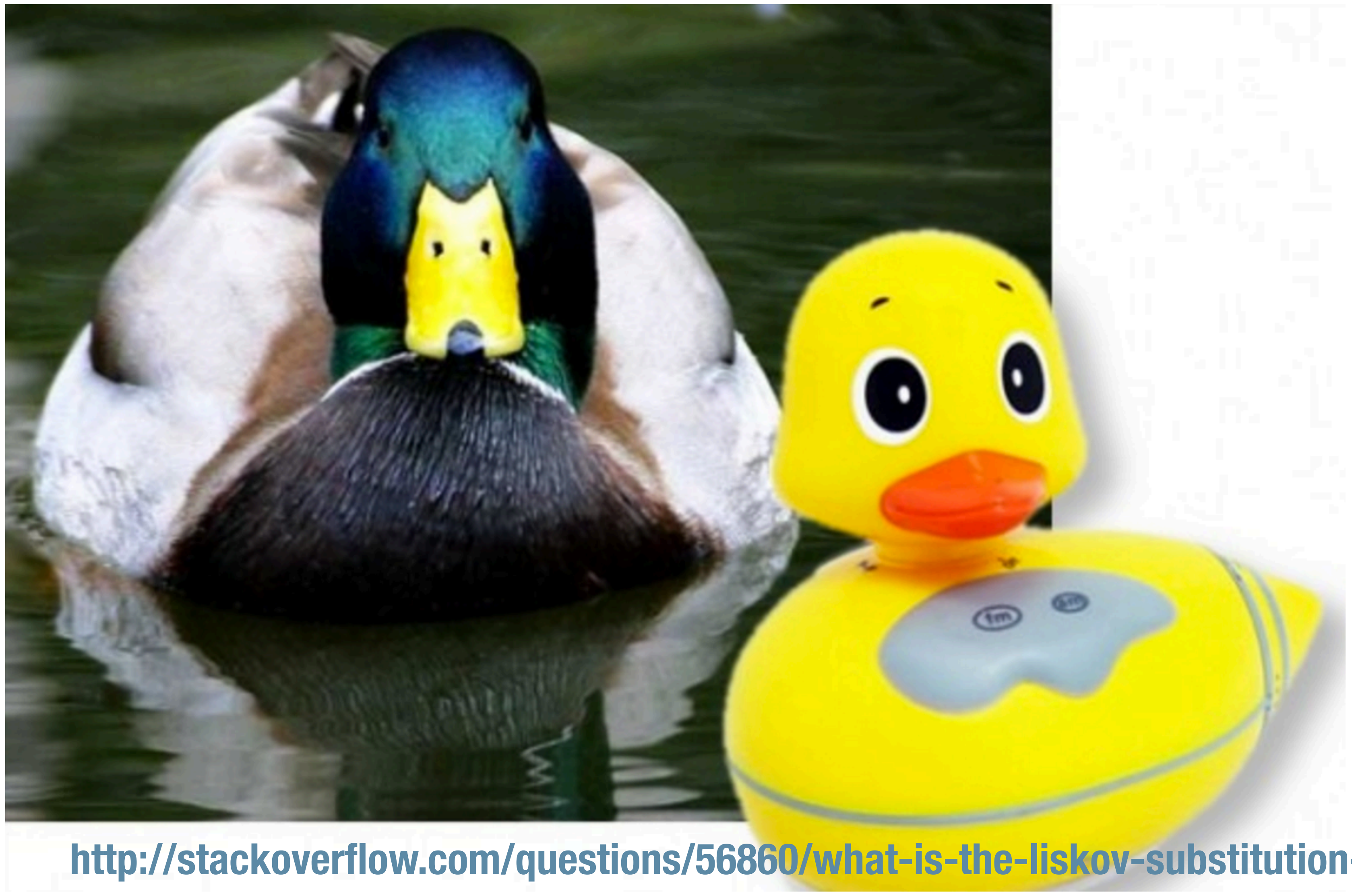
- Classes should have only one major task
- Insulates classes from one another

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

Liskov substitution principle

- Subtypes should behave as their parent types
- aka a program should still behave correctly should two subtypes of a common type be interchanged



http://stackoverflow.com/questions/56860/what-is-the-liskov-substitution-principle

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

https://lostechies.com/derickbailey/files/2011/03/InterfaceSegregationPrinciple_60216468.jpg

Dependency inversion

- Also known as the ‘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

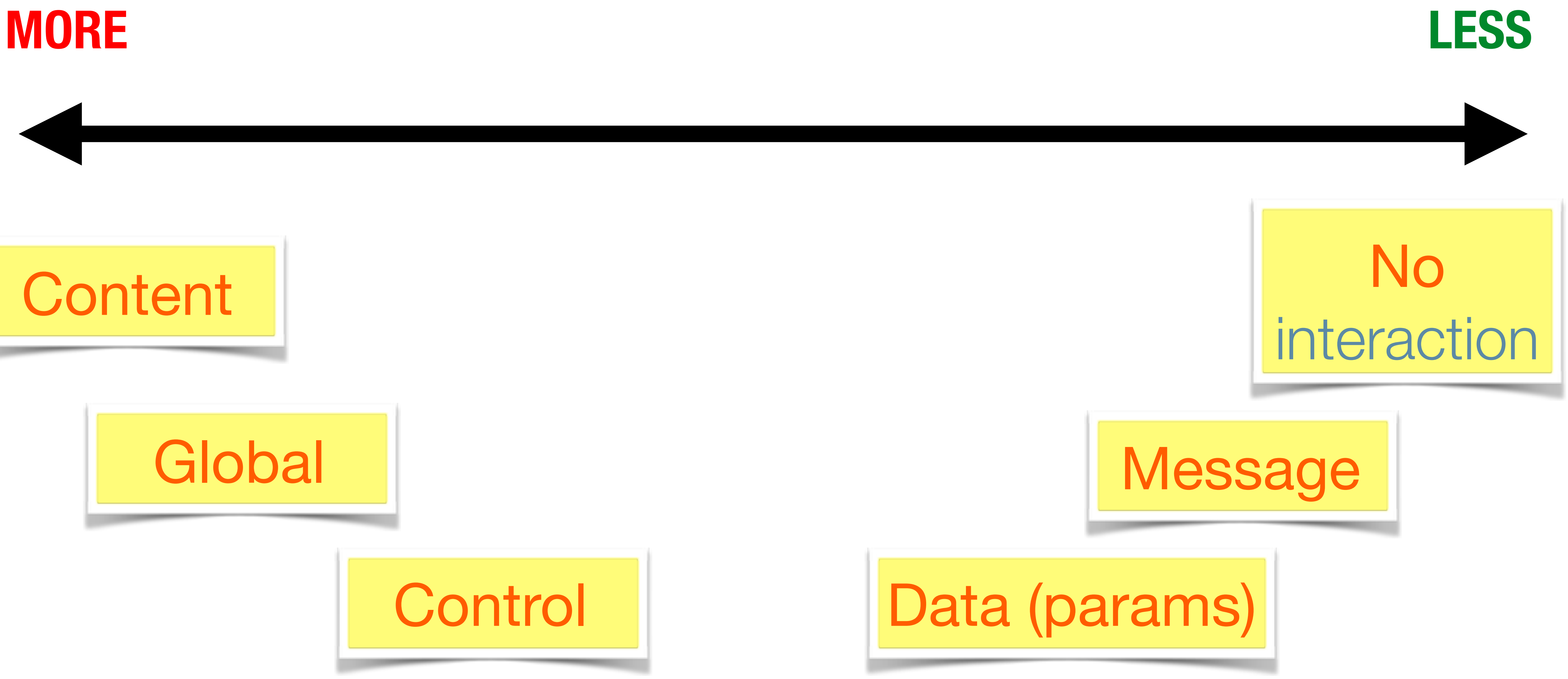
Lower-level principles

- Encapsulate what varies
  - This is a key concern to increase reusability and reduce the impact of 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

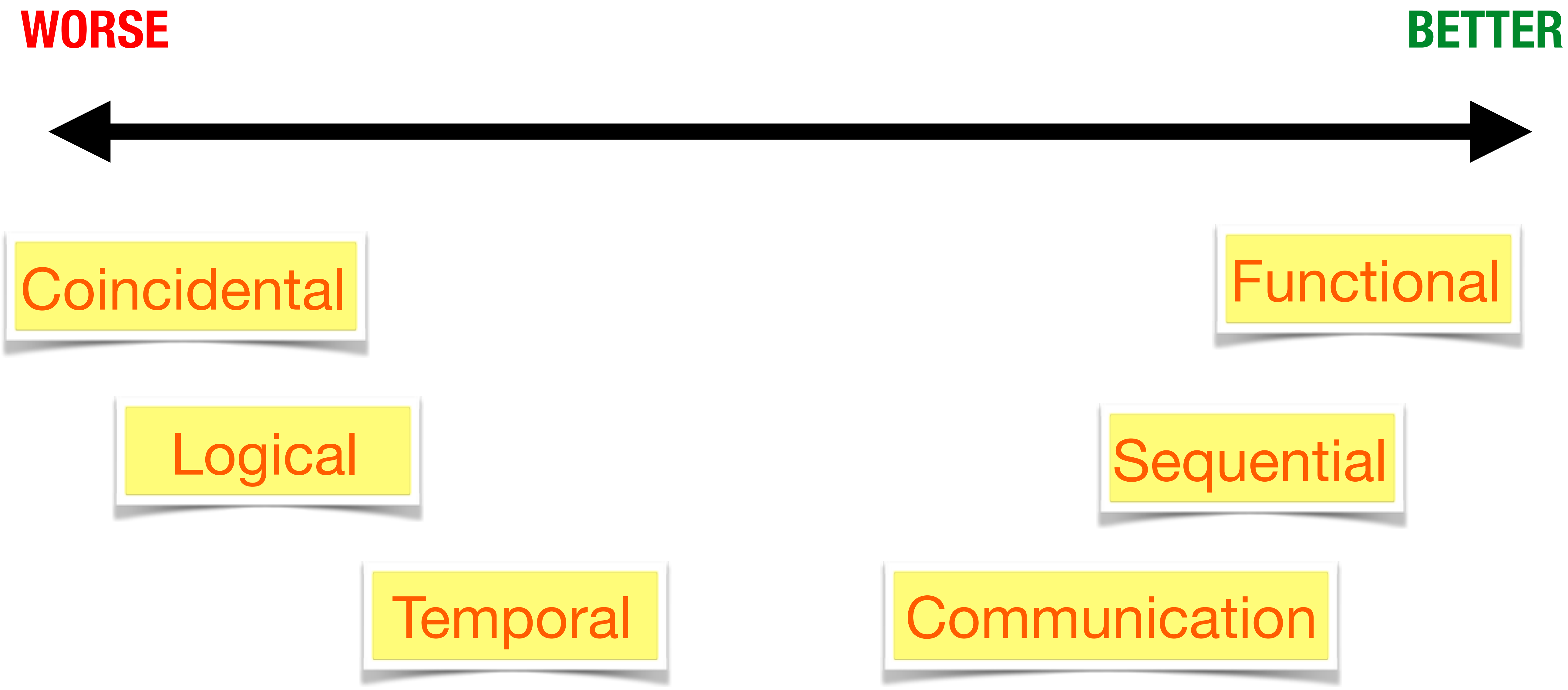
Quality attributes

- 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
Logical
Temporal
Communication
Sequential
Functional

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