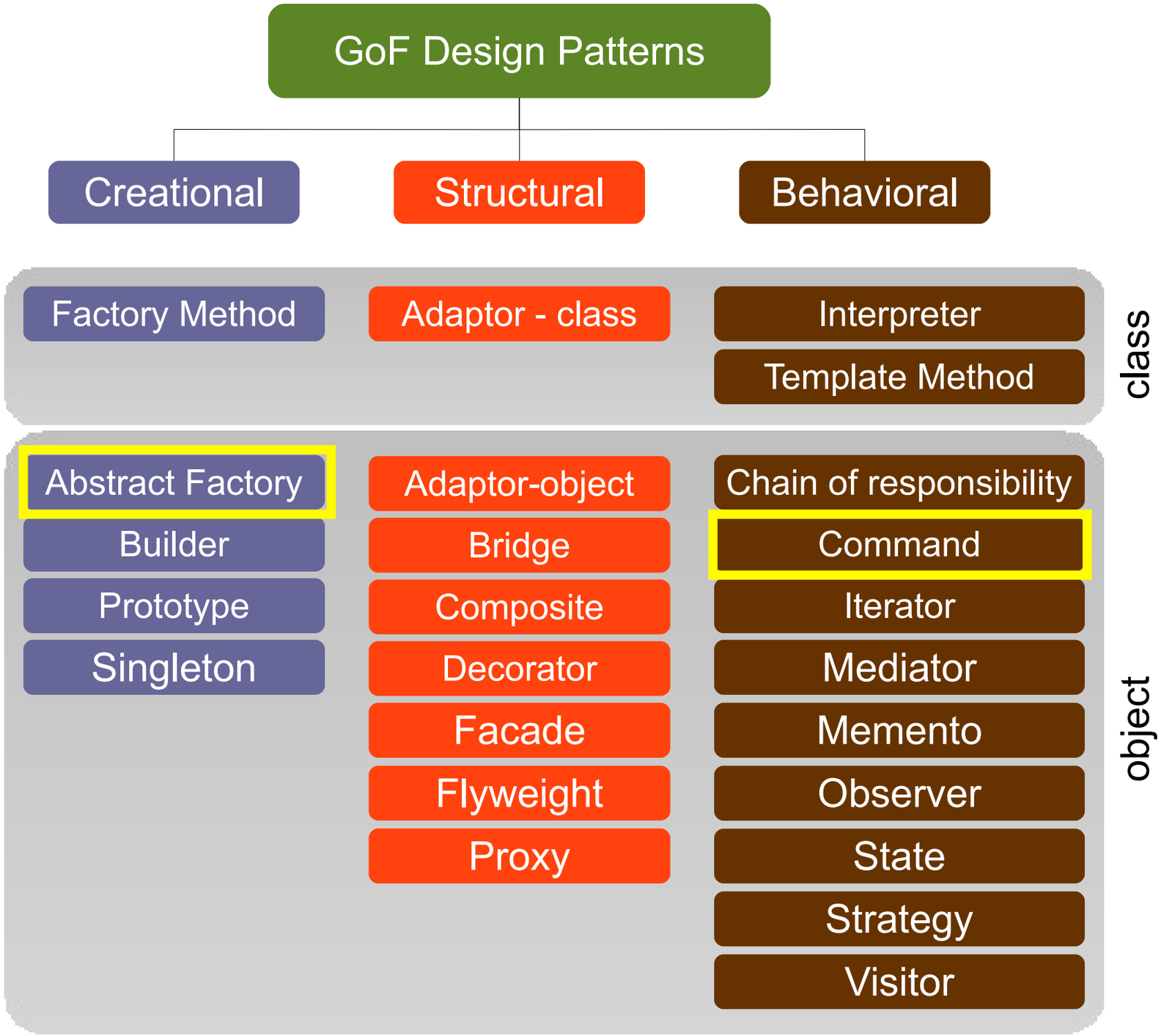
Material and some slide content from:
- GoF Design Patterns Book
- Head First Design Patterns

Design Patterns #3

Reid Holmes

Lecture 16 - Thursday November 15 2011.

GoF design patterns



Factory Method

- Intent: “Provide an interface for creating an object but let subclasses decide which class to instantiate”
- Implementation:
  - Create an abstract method (e.g. `createPizza()`)
  - Let subclasses implement method
  - In this way the subclasses control instantiation without the client knowing what is being created

Abstract factory

- Intent: “Provide an interface for creating families of related objects without specifying their concrete classes”
- Motivation: Consider a multi-platform UI toolkit. A WidgetFactory can provide an interface to make sure the right widget is instantiated for each platform.

- Applicability:
  - When a system should be independent of how its products are created and represented.
  - A system contains multiple families of products.
  - You want to reveal interfaces, not implementations.

Abstract factory

- Structure

- Participants:
  - Abstract/Concrete Factory
  - Abstract/Concrete Product
  - Client

Abstract factory

- Collaborations
  - Usually only one Abstract Factory (singleton).
  - Objects are created by concrete factories.
- Consequences:
  - Isolates concrete classes from clients.
    - (Clients only know about interfaces, not implementations)
  - Makes exchanging families easy.
    - (Concrete family reference appears only once)
  - Makes adding products hard.
    - (Abstract + all concrete factories must be updated.)

Abstract factory

- Implementation:
  - Create abstract factory interface.
  - Use factory method to create descriptive names.
  - Create concrete products/factories.
  - Associate client with one factory.
- Known uses: Frequently used in widget toolkits.
- Related to: Often implemented with Factory Method or Prototypes. Concrete factories are often Singletons.
- XXX: Elaborate on Complex(..) example and the utility of the Factory Method.

Dependency Inversion

Depend upon **abstractions**,
not **concrete** classes.

- Instantiations are references to concrete classes
- Factories allow high-level components to depend on abstractions
- Low-level components can then implement those abstractions and depend upon them
- Hints:

Facade

- Intent: “Provide a unified, higher-level, interface to a whole module making it easier to use.”
- Motivation: Composing classes into subsystems reduces complexity. Using a Facade minimizes the communication dependencies between subsystems.
- Applicability:
  - When you want a simple interface to a complex subsystem.
  - There are many dependencies between clients and a subsystem.
  - You want to layer your subsystems.

Facade

Facade

- Participants:
  - Facade
  - Subsystem classes
- Collaborations:
  - Clients interact subsystem via Facade.
- Consequences:
  - Shields clients from subsystem components.
  - Promotes weak coupling. (strong within subsystem, weak between them)
  - Doesn't prevent access to subsystem classes.

Facade

- Implementation:
  - 1) Analyze client / subsystem tangling.
  - 2) Create interface. Abstract factories can also be used to add further decoupling.
- Known uses: Varied.
- Related to: Abstract Factory can be used with Facade to create subsystem objects. Facades are frequently Singletons. Abstracts functionality similar to Mediator but does not concentrate on communication.

Activity

- 5 mins:
  - Right side: Develop a use for a observer or command pattern for your system.
  - Left side: Develop a usage of a decorator pattern or abstract factory for your system.
- 10 mins (5 / group):
  - Match up with team from other side of room. Explain your pattern and how it improves your system's design.