GRASP in Short

"GRASP" Patterns

General **R**esponsibility **A**ssignment **S**oftware **P**atterns:

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

"The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology."

- Larman, Craig. *Applying UML and Patterns - Third Edition*
- GRASP is a learning aid.

Information Expert

- Assign a responsibility to the class that has the information necessary to fulfill it. "Partial experts" collaborate.
- This is the most basic responsibility assignment principle.
- If you find yourself using many *setters* and *getters*, you may have violated this principle.
- Examples:
  - In the Sticks game, which class should have the responsibility of displaying the character for a stick?
  - In the Video Store, which class should have the responsibility for knowing if a video is overdue?
  - In the Sticks game, which class knows when the game is over?
  - Temperature converter.

Creator

- Every object must be created somewhere.
- Consider making a class responsible for creating an object if:
  - It has the information needed to initialize the object.
  - It will be the primary client of the object.
  - It is an inventory of objects of that type.
- Sometimes this pattern suggests a new class.
- In the Sticks game, who creates a Move?
- Refer to the ***creational design patterns***:
  - ***Factory, Builder, Prototype & Singleton***
- In C++ there needs to be a ***destructor*** as well.

High Cohesion

- Cohesion is a measure of the degree to which a class' responsibilities are ***semantically related***.
- High cohesion promotes:
  - Ease of understanding & maintenance
  - Encapsulation
  - Low coupling
- ***Separate concerns***.
- Counter Example:
  - An entire program could be written with one class and many methods. That's not OO.

Low Coupling

- Coupling is a measure of how strongly one class has knowledge of, or relies upon other classes.
- Low coupling is encouraged by using ***interfaces***, and the maximum degree of ***encapsulation***.
- Low coupling reduces the complexity of the graph of interacting objects.
- Counter Example: Spaghetti code.

Law of Demeter:

- Only talk to your immediate friends:

  `dog.legs.walk()` breaks the law;

  `dog.walk()` does not.

Controller

You often need an object to coordinate other objects.

- Objects have responsibilities which can include controlling and sequencing.
- Commonly used with transactions & program flow.
- Examples:
  - In the Sticks Game, the Referee
  - The ***Transaction Agent*** Design Pattern
  - The ***Mediator*** and ***Façade*** Design Patterns
  - The ***Model - View - Controller*** Design Pattern
  - The class Fractal in the Fractal Applet
    `http://www.gui.net/fractal.html`

Design Pattern: *Mediator*

***Intent*:** Encapsulate the interaction(s) between a set of classes.

***Examples*:**

- The Oven
- The Sticks Game Referee.

***Applicability*:**

- Whenever the communication between classes gets complex.
- Whenever two or more interfaces must vary independently.
- Whenever information must flow between two classes, but neither class wishes to accommodate the other.

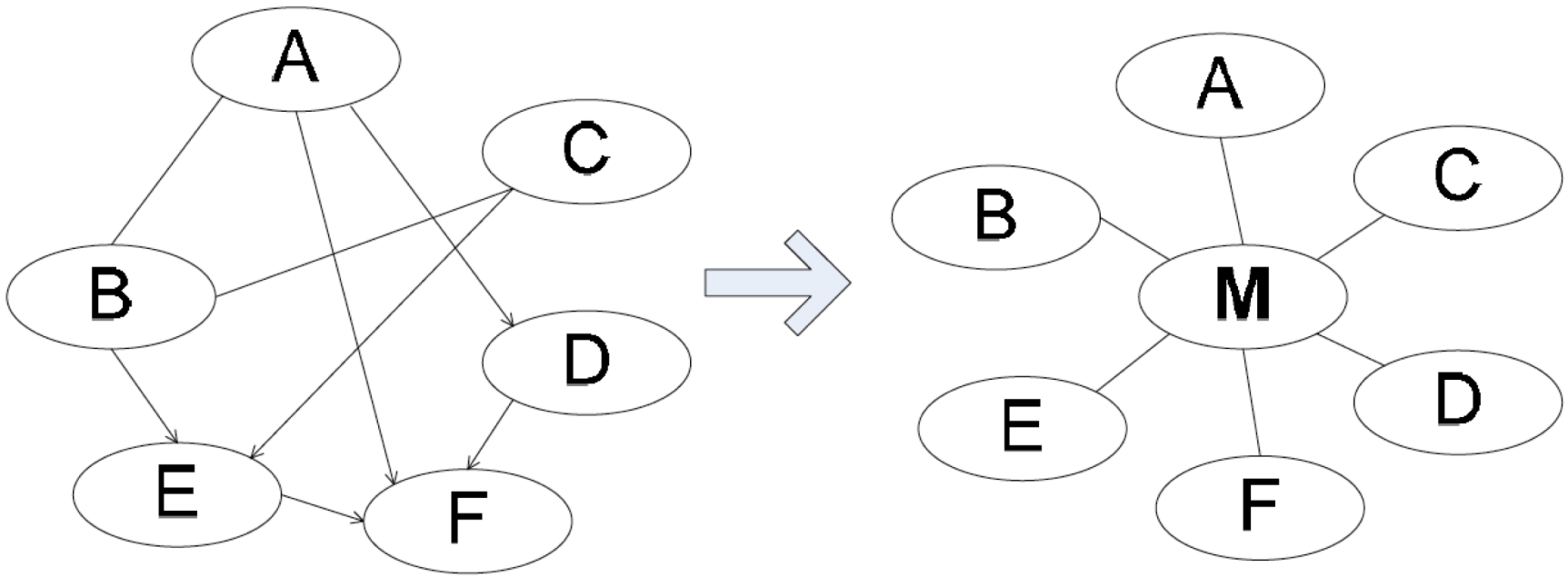
***Pros*:**

- Promotes loose coupling between objects.
- Simplifies complex collaboration diagrams.

***Cons*:**

- The Mediator itself can become quite complex.

Mediator according to Graph Theory

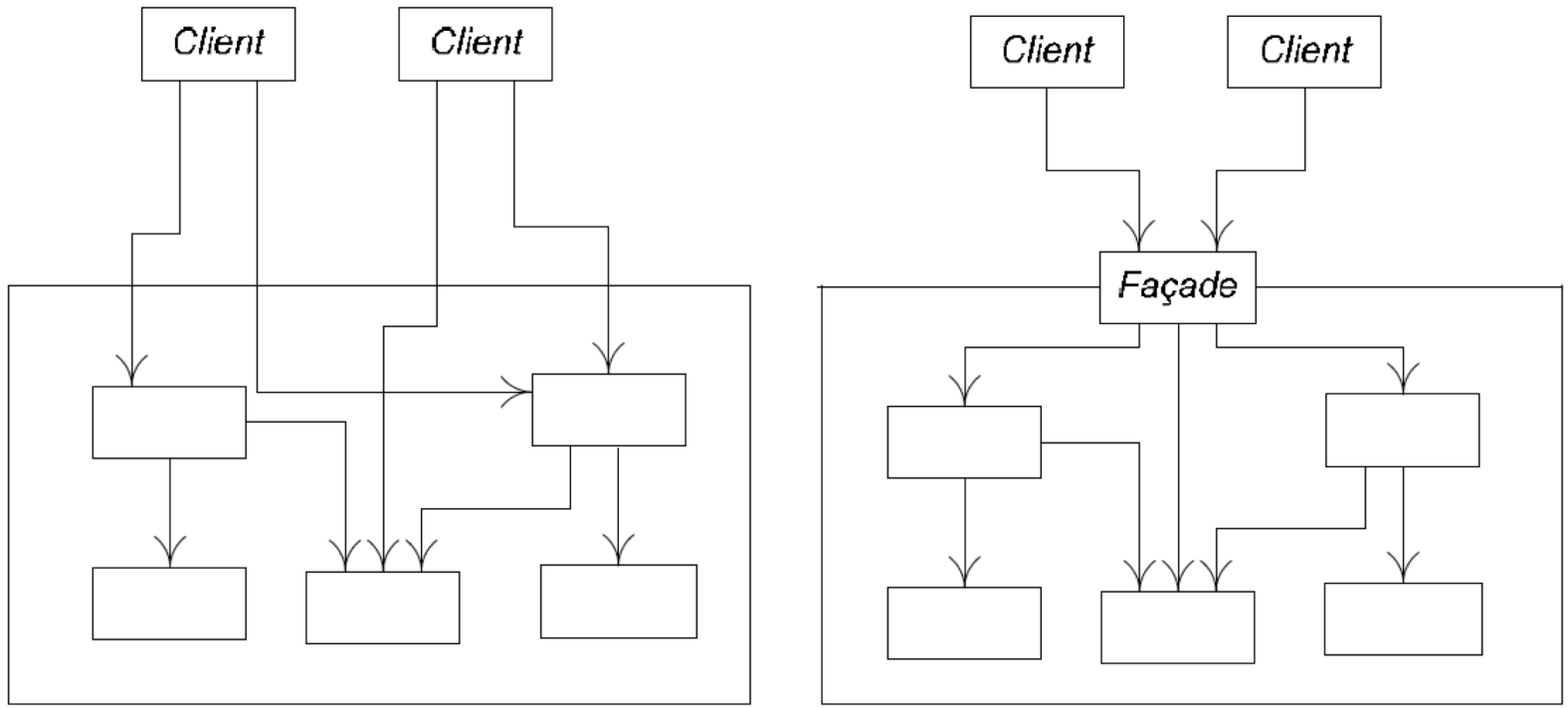


N.B.: This diagram is not UML

Design Pattern: *Façade*

- **Intent**: Provide a single interface to an architectural layer or component. The Façade class ***controls*** the other classes that make up the sub-system.
- Promotes ***low coupling***; the client knows only about the Façade.
- May compromise ***cohesion***; the Façade class itself can get large.
- Does little more than ***delegate*** to other classes inside the package.
- Often used in distributed applications to increase performance by reducing the number of calls across the network.
- Examples:
  - Modern interface to legacy system
  - Customer Service Representative

Façade Example



- The Façade defines an interface that makes the subsystem easier to use. Clients remain ignorant of the details of the subsystem's components.

Model-View-Controller

Poorly designed GUIs have classes with a large and incoherent set of responsibilities that include configuring the layout, listening for events, domain logic, application logic, technology specifics, … "spaghetti code" ... We should ***separate the concerns***.

- The **Model** (data plus domain / ***business logic***) knows nothing about the presentation of information to humans.
- The **View** is concerned only with the user interface, handling user events (deciphering the user's ***intent*** from the ***gesture)***, and delegating to the Controller.
- The **Controller** is "the glue" – the ***application logic***. Things like multi-threaded synchronization and transaction control are often done here.