Low Coupling

Low Coupling

- **Problem**
  - How can your design support low dependency and increased reuse?
- **Solution**
  - Assign a responsibility so that coupling remains low.

Low Coupling

- Coupling can be defined as the strength with which one class is connected to other classes.
- A class with low coupling is not dependent on too many other classes.
  - Not affected by changes in other components
  - Simple to understand in isolation
  - Convenient to reuse

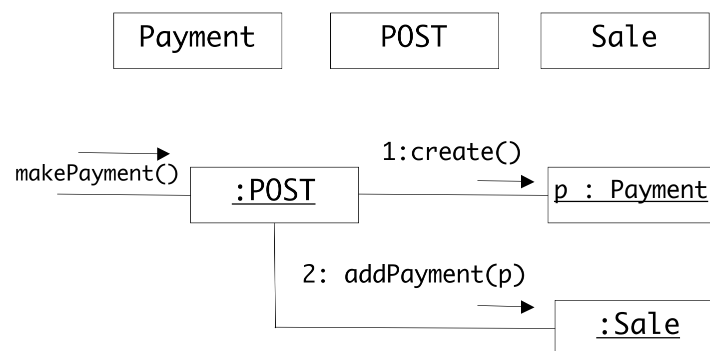
High Coupling

- Hard to understand the class in isolation.
- Hard to reuse
  - Requires additional presence of dependent classes.

Low Coupling

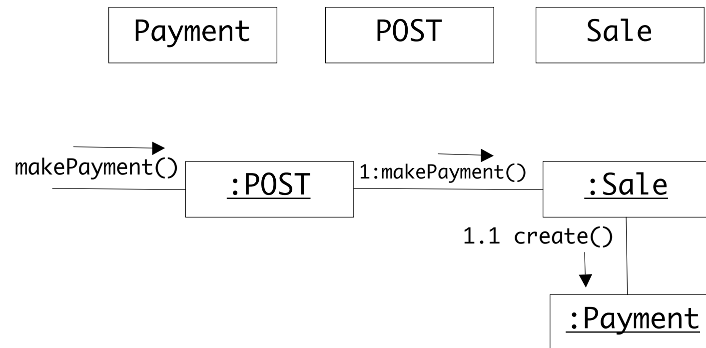
- This concept can be taken to extremes.
  - Almost no coupling!
  - Leads to poor design
  - Only a few bloated classes

Low Coupling Example



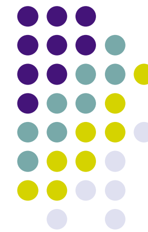
POST creates Payment

Low Coupling Example



Sale creates Payment

High Cohesion



High Cohesion

- **Problem**
  - How do you keep complexity manageable?
- **Solution**
  - Assign a responsibility so that cohesion remains high.

Cohesion

- Cohesion is a measure of how strongly related and focused are the responsibilities of a class.

Low Cohesion

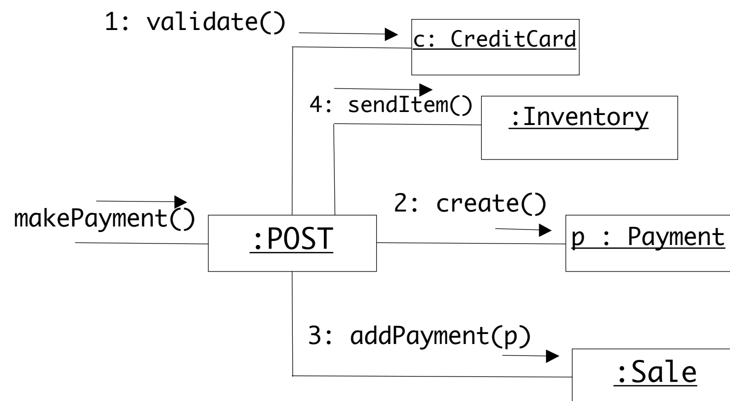
- Many unrelated things and too much work to do.
  - Hard to comprehend
  - Hard to reuse
  - Hard to maintain
  - Greatly affected by change

High Cohesion

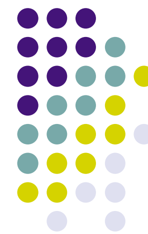
- This is an evaluative pattern.
  - It is applied to all decisions about design.
- There should be a fine grain of highly related functionality in a class.

High Cohesion Example

This design is fine for one action (making a payment) but what happens if POST is made responsible for more and more work?



Controller



Controller

- **Problem**
  - Who should be responsible for handling a system event?
- A system event is an external input event.
  - Associated with system operations.

Controller

- **Solution**
  - Assign responsibility for handling a system event message to a class that is a
  - *Façade Controller*
    - Represents the system or organization
  - *Role Controller*
    - Represents something in the real world that is active
  - *Use Case Controller*
    - Represents an artificial handler of all system events of a use case.

Controller

- A controller is a non-user interface object responsible for handling a system event.
- The same controller class should be used for all system events of one use case.
  - Maintain information about state of the use case, e.g. identify out-of-sequence operations.

Controller

- Controllers should not have too much responsibility; they should delegate to other objects.
  - They should just coordinate.

Façade Controller

- Suitable when there are only a few system events or it is not possible to redirect messages to alternate controllers.
  - E.g. message processing system

Use Case Controller

- Use when the alternatives lead to low cohesion or high coupling.
- Many system events across different processes.

Controller Pattern Problems

- **Bloated Controllers**
  - Handles too many events
  - Does too much work
  - Maintains too much information
  - Cures
    - Add more controllers
    - Design controller so that it delegates responsibilities to other objects

Controller Pattern Problems

- **Role Controllers**
  - Often lead to low cohesion
  - Should be used sparingly

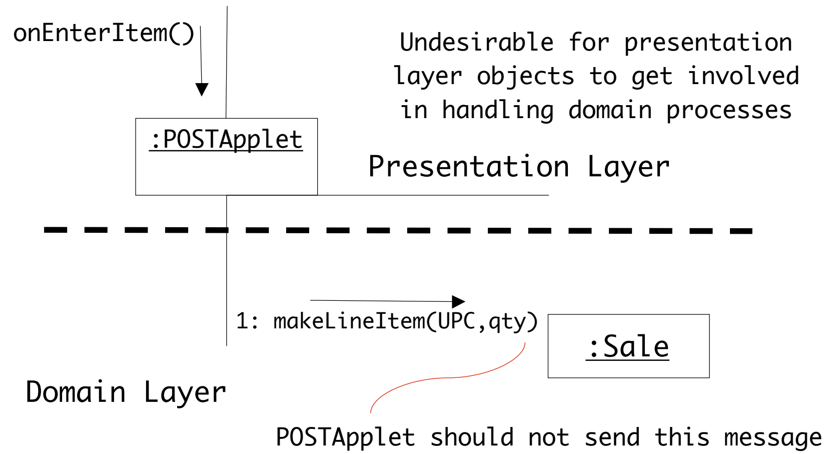
Corollary of Controller Pattern

- External interfacing objects (windows, applets, etc.) and the presentation layer should not have responsibility for fulfilling system events.
- System operations should be handled in the domain layer of objects rather than in the interface, presentation or application layers of the system.

Corollary of Controller Pattern

- Increased potential for reuse.
  - Interface as controller reduces opportunity to reuse domain process logic since it is bound to the interface.
- Ability to understand state of the system or use case.

Undesirable Coupling



Desirable Coupling