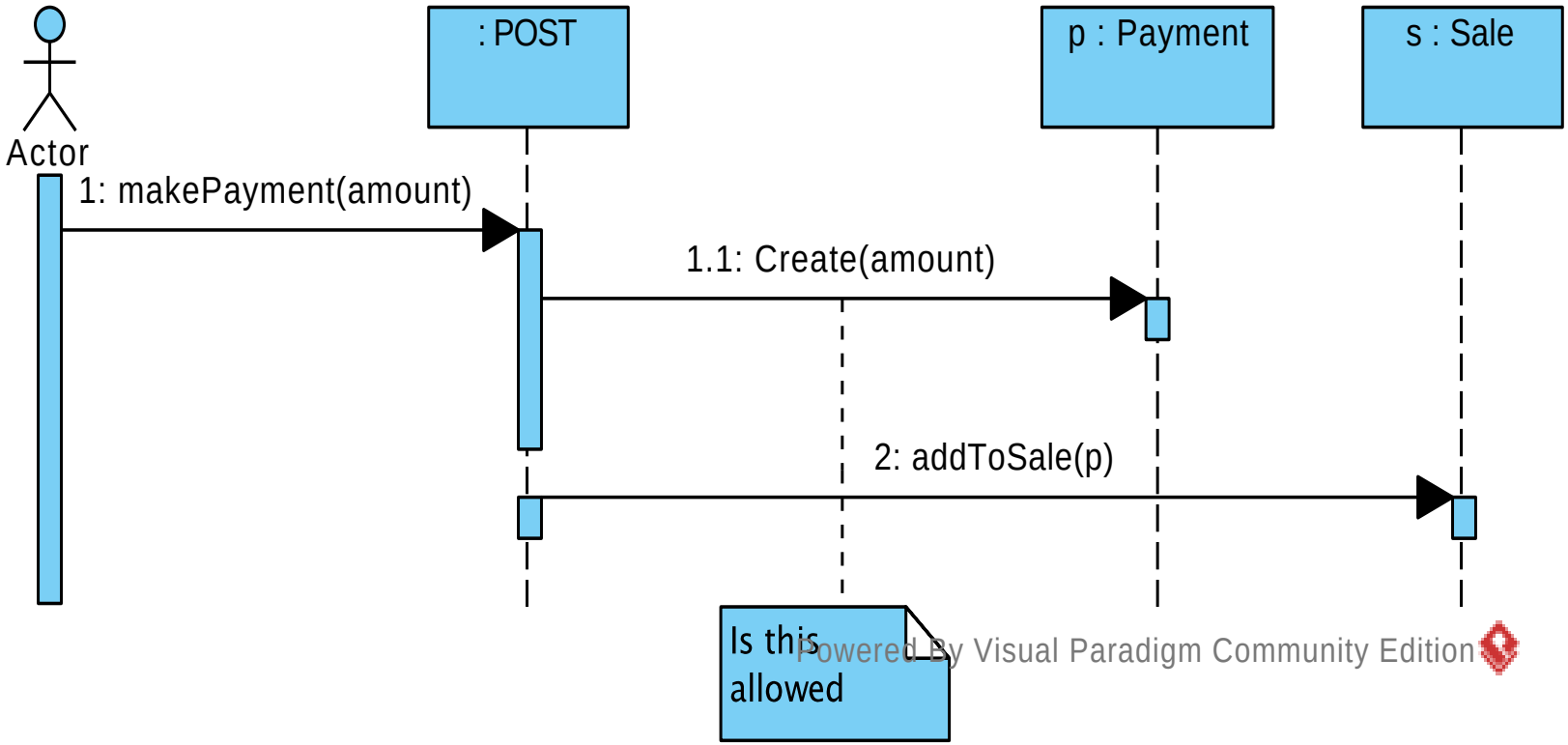
Object-Oriented Design
- Design Patterns: Low Coupling, High Cohesion & Controller

Zhiming Liu
zhimingliu88@swu.edu.cn
Centre for Research and Innovation of Software Engineering
School of Computer and Information Science
Southwest University, Chongqing, China

An Alternative Design?



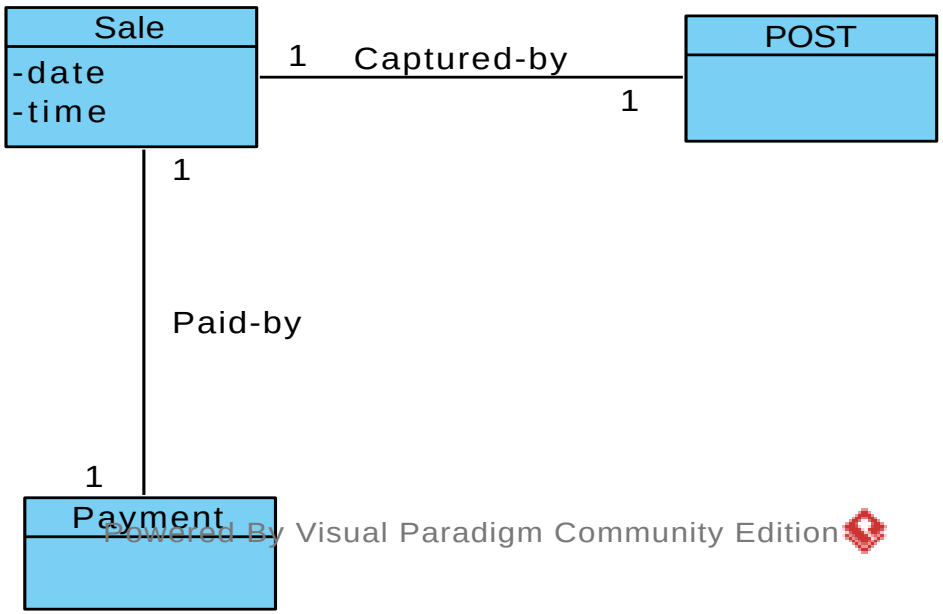
Why not right?

Why not right?

No association between *POST* and *Payment*

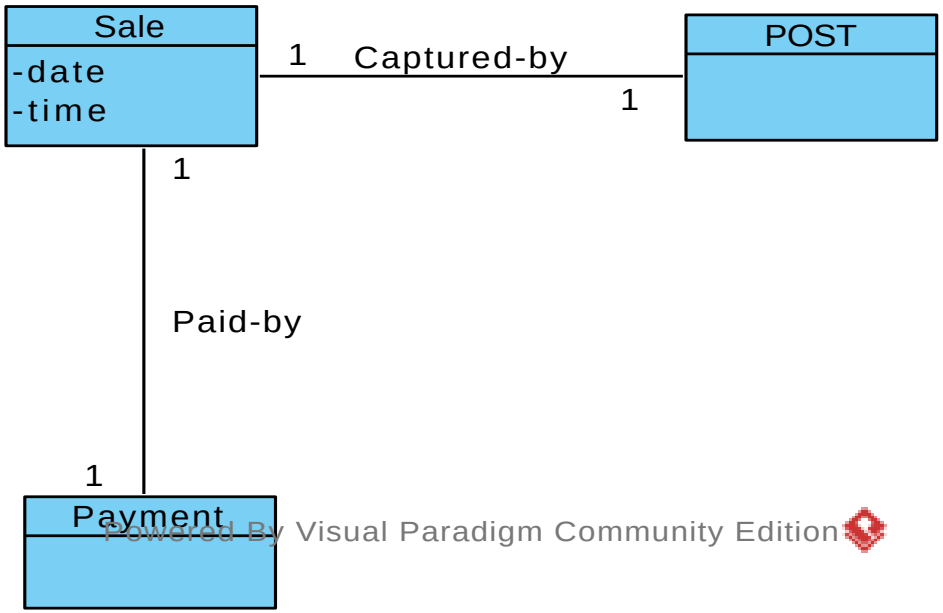
Why not right?

No association between *POST* and *Payment*



Why not right?

No association between *POST* and *Payment*



Need to add an association between POST and Payment, but .....

Coupling

- Coupling is a measure of how strongly one class is connected to or relies upon other classes
- A class with low coupling is not dependent on too many other classes.
- High coupling is not desirable

Low Coupling Pattern

Pattern Name: Low Coupling

Solution: **Assign a responsibility so that coupling remains low**

Problem: How to support low dependency an increased reuse?

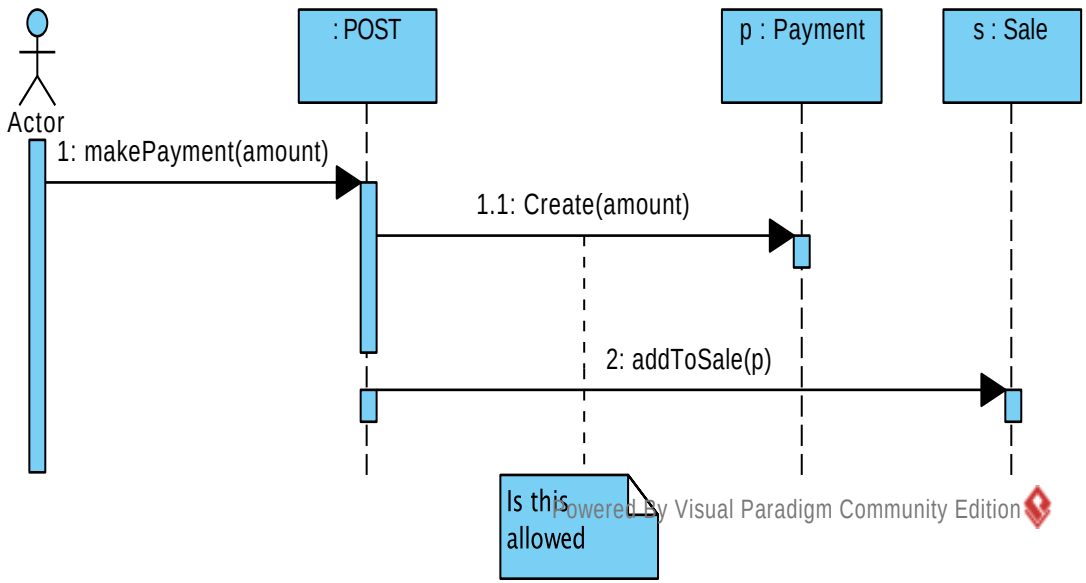
- Consider the classes *Payment*, *CashDesk* and *Sale* in POST
- Assume we need to create a *Payment* instance and associate it with the *Sale*.
- Who should be responsible for this, *CashDesk* or *Sale*?

If the *CashDesk*

In the slides, *CashDesk* is some times used for class POST in the course notes and the Conceptail Class Diagrams in the Requirements Analysis

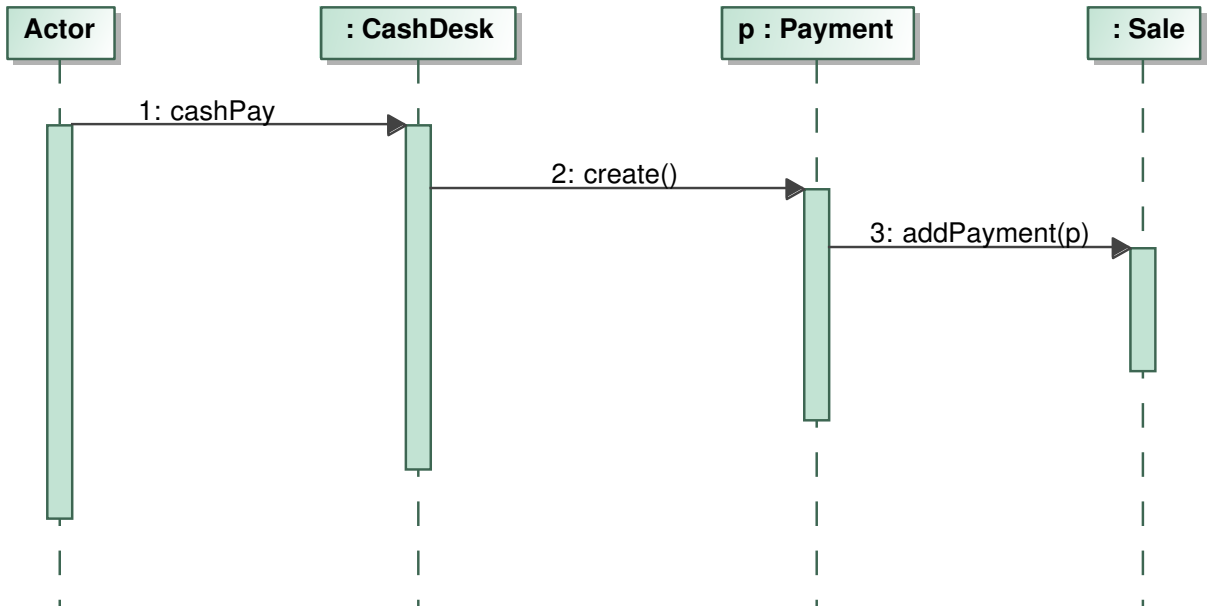
If the *CashDesk*

In the slides, *CashDesk* is some times used for class POST in the course notes and the Conceptail Class Diagrams in the Requirements Analysis



We need an extra link from *POST* to *Payment*
So *Sale* is a better choice!

How about?



CashDesk is the same as POST

Cohesion

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

- Classes with high cohesion have highly related functionalities, and does not do a tremendous amount of work
- They have a small number of methods with simple but highly related functionality
- Classes with low cohesion are not desirable

High Cohesion Pattern

Good OOD is to assign responsibilities to classes that

- Are naturally and strongly related to the responsibilities, and
- Every class has something to do but does not have too much to do.

Pattern Name: High Cohesion

Solution: Assign a responsibility so that cohesion remains high

Problem: How to keep complexity manageable?

High coupling also contributes to low cohesion!

Example

- *CashDesk* can take on part of the responsibility for carrying out the *cashPay*()
- But *CashDesk* is used as handling the interface methods, it would better if it does not act as the creator of *Payment*
- Thus, a design that delegates the payment creation responsibility to the *Sale* would more cohesive

Note: *CashDesk* here is *POST* in the Conceptual Class Diagram

Controller Pattern

Pattern Name : Controller

Solution: Assign the responsibility for handling an input to a class representing one of the following choices:

- Represents the "overall component" (*facade controller*).
- Represents the overall business (*facade controller*).
- Represents something in the real-world that is active that might be involved in the task (*role controller*).
- Represents an artificial handler of all input events of a use case, (*use-case controller*).

Problem: Who should be responsible for handling an external input event?

Design of POST

To design a project

1. Create a separate **object sequence diagram** for **each use case operation** whose **contracts** are defined:

   For each use case op(), make a diagram with it as message to the **controller object**.

Design of POST

To design a project

1. Create a separate **object sequence diagram** for **each use case operation** whose **contracts** are defined:

   For each use case op(), make a diagram with it as message to the **controller object**.

2. If the diagram gets complex, split it into smaller diagrams.

Design of POST

To design a project

1. Create a separate **object sequence diagram** for **each use case operation** whose **contracts** are defined:
   For each use case op(), make a diagram with it as message to the **controller object**.
2. If the diagram gets complex, split it into smaller diagrams.
3. For some functionality, mostly outputs, may need to create a separate object sequence diagram, e.g. toto()

For these, we need

Design of POST

To design a project

1. Create a separate **object sequence diagram** for **each use case operation** whose **contracts** are defined:
   For each use case op(), make a diagram with it as message to the **controller object**.
2. If the diagram gets complex, split it into smaller diagrams.
3. For some functionality, mostly outputs, may need to create a separate object sequence diagram, e.g. toto()

For these, we need

- Use the contracts and conceptual class diagram, and apply the GRASP

Design of POST

To design a project

1. Create a separate **object sequence diagram** for **each use case operation** whose **contracts** are defined:
   For each use case op(), make a diagram with it as message to the **controller object**.
2. If the diagram gets complex, split it into smaller diagrams.
3. For some functionality, mostly outputs, may need to create a separate object sequence diagram, e.g. toto()

For these, we need

- Use the contracts and conceptual class diagram, and apply the GRASP
- For each use case, we use a an **interface class** as the controller, following the Controller Pattern