Design Patterns

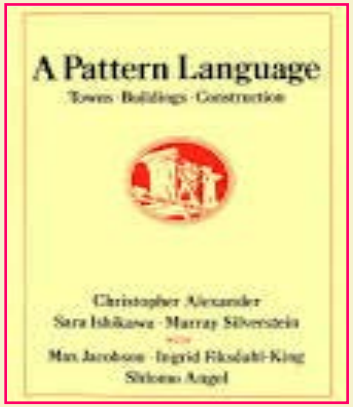
- Patterns are used as “building blocks” in software design.

Design Patterns

  - Help designers to make important design decisions correctly.

- If you can master a few important patterns:
  - **You can easily spot them in your next application design problem and use pattern solutions.**

Christopher Alexander, A Pattern Language, 1977



- **Each pattern:**
  - Describes a problem which occurs over and over again.
  - Describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice.

Patterns in Engineering

- How do other engineering fields find and use patterns?
  - **Mature engineering disciplines have handbooks describing successful solutions to known problems**
  - **Automobile designers don't design cars from scratch.**
  - **Instead, they reuse standard designs with successful track records.**
- Should software engineers make use of patterns?
  - **Developing software from scratch is expensive**
  - **Patterns support reuse of past knowledge mainly in the form of software architecture and design.**

Patterns in Software Design

- **Architectural Patterns:** MVC, Layers etc.
- **GoF Design Patterns:** Singleton, Observer etc
- **GUI Design Patterns:** Window per task, Disabled irrelevant things, Explorable interface etc
- **Database Patterns:** decoupling patterns, resource patterns, cache patterns etc.
- **Concurrency Patterns:** Double buffering, Lock object, Producer-consumer,Asynchronous processing etc.
- **J2EE Patterns:** Data Access Object, Transfer Objects etc.
- **GRASP (General Responsibility Assignment Patterns):** Low coupling/high cohesion, Controller, Law of Demeter (don't talk to strangers), Expert, Creator etc.
- **Anti-patterns (**Bad solutions deceptively appear good): God class,Singletonitis, Basebean etc.

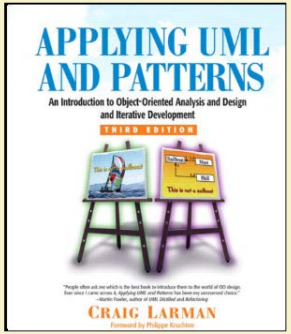
History of Design patterns

- The concept of a "pattern“:
  - First expressed in Christopher Alexander's work A Pattern Language in 1977
  - A solution to a common design problem in a certain context.
- In 1990:
  - Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compiled a catalog of design patterns
- Now assimilated into programming languages:
  - Example: Decorator, Composite, Iterator pattern
    Defines an interface that declares methods for sequentially accessing the objects in a collection.




"Gang of four" (GoF) and GRASP Patterns

- Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
  - **Design Patterns book catalogs 23 different patterns**
- **Larman**
  - **GRASP (General Responsibility Assignment Software Patterns) pattern**



- Several other types of patterns are also popular.

Elements of Design Patterns

- Design patterns usually have 4 essential elements:
  - **Pattern name:** Designers' vocabulary
  - **Problem:** intent, context, and when to apply
  - **Solution:** UML model, skeletal code
  - **Consequences:** results and tradeoffs

Design Patterns Aim To…

- **Codify good design**
  - Distill and generalize experience
  - Aids novices and experts alike
- **Give important design solutions explicit names**
  - Common vocabulary
- **Save design iterations**
  - Improve documentation
  - Improve understandability
  - Facilitate restructuring/refactoring

Types of Patterns

- Architectural patterns
- Design patterns
- Code patterns (Idioms)

Architectural Patterns

- **Architectural designs concern the overall structure of software systems.**
  - Architectural designs cannot directly be programmed.
  - Form a basis for more detailed design.
- **Architectural patterns:**
  - **Relevant to providing high-level solutions to large problems.**

Design Patterns

- A design pattern:
  - **Suggests classes in a design solution.**
  - **Also, defines the interactions required among the classes.**
- Design pattern solutions are described in terms of:
  - Classes, their instances, their roles and collaborations, skeletal code.

Idioms

- Idioms are a low-level patterns:
  - Programming language-specific.
  - **Describe how to implement a solution to a particular problem using a given programming language.**

Idioms

- Idioms in English language:
  - **A group of words that has meaning different from a simple juxtaposition of the meanings of the individual words.**
  - Example: “Raining cats and dogs”
- A C idiom:

```
for(i=0;i<1000;i++){
                }
```

Patterns versus Idioms

- A pattern:
  - **Describes a recurring problem**
  - **Describes a core solution**
  - **Used for generating many distinct designs**
- An Idiom though describes solution to a recurring problem, is rather restricted:
  - **Provides only a specific solution, with fewer variations.**
  - **Applies only in a narrow context**
    - **e.g., C++ or Java language**

Antipattern

- If a pattern represents a best practice:
  - **An antipattern represents lessons learned from a bad design.**
- Antipatterns help to recognise deceptive solutions:
  - **They appear attractive at first, but turn out to be a liability later…**
- Not enough to use patterns in a solution –must consciously avoid antipatterns.

Patterns versus Algorithms

- **Are patterns and algorithms identical concepts?**
  - After all, both target to provide reusable solutions to problems!
- Algorithms primarily focus on solving problems with reduced space and/or time requirements:
  - **Patterns focus on understandability and maintainability of design and easier development.**

Pros of Design Patterns

- Help capture and disseminate expert knowledge.
  - **Promotes reuse and helps avoid mistakes.**
- Provide a common vocabulary:
  - Help improve communication among the developers.
  - **"The hardest part of programming is coming up with good variable and function names."**

Pros of Design Patterns

- Reduces the number of design iterations:
  - **Helps improve the design quality and designer productivity.**

- Patterns exempify good soutions to common design problems by making use of:
  - **Abstraction,**
  - **Encapsulation**
  - **SRP, OCP, LSP, ISP, DIP**
  - **Separation of concerns**
  - **Coupling and cohesion**
  - **Divide and conquer**

Pros of Design Patterns

Cons of Design Patterns

- Design patterns do not directly lead to code reuse.
- To help select the right design pattern at the right point during a design exercise.
  - **At present no systematic methodology exists.**

Why learn Design Patterns?

- **Your own designs will improve**
  - borrowing well-tested ideas
  - pattern descriptions contain some analysis of tradeoffs
- **You will be able to describe complex design ideas to others**
  - assuming that they also know the same patterns
- **You can use patterns to "refactor" existing code**
  - Improve the structure of existing code without adding new functionality
- **You can understand why some aspects of Java language are that way.**

Thought for the day…

- **Why are there no design patterns for procedural design? ….**

Why No Patterns for Procedural Development?

- Object-Oriented programming languages (and paradigms) facilitate development of meaningful design patterns.
- Procedural languages need to support:
  - **Inheritance**
  - **Abstract classes and Interfaces**
  - **Polymorphism**
  - **Encapsulation**

Pattern Documentation

- A pattern documented by describing the situation in which to use the pattern solution.
  - **Explain the problem and its context.**
  - **Describe situations where it works and does not work.**
  - **Include a list of conditions that must be met before it makes sense to apply the pattern.**

Pattern Documentation Artifacts

- Overview of the solution…
- Describe solution in terms of:
  - Classes
  - Their relationships,
  - Responsibilities and
  - Collaborations
- **Example code.**

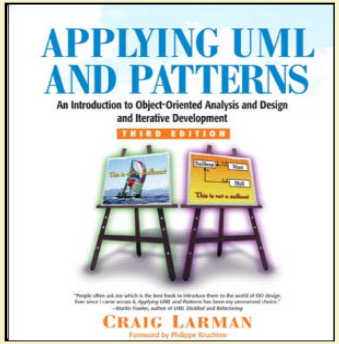
Design Patterns are NOT…

- NOT designs that can be plugged in and reused as is
  - For example, code for linked lists, hash tables
- NOT complex domain-specific designs:
  - For an entire application or subsystem.
- Patterns are actually:
  - **“Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”**

GRASP Patterns

GRASP Patterns

- GRASP: **Generalized Responsibility Assignment Software Patterns:**
  - Larman, "Applying UML and Patterns"
- **GRASP patterns can more accurately be described as best practices:**
  - If used judiciously, will lead to maintainable, reusable, understandable, and easy to develop software



GRASP Patterns

- GRASP patterns essentially describe how to assign responsibilities to classes:
  - **Warning**: **Some Grasp patterns tend to be vague and need to be seen as guidelines rather than a concrete solution.**
- What is a responsibility?
  - **A contract or obligation of a class**
  - Responsibilities can include behaviour, data storage, object creation, etc.
  - Usually fall into two categories:
    - **Doing**
    - **Knowing**

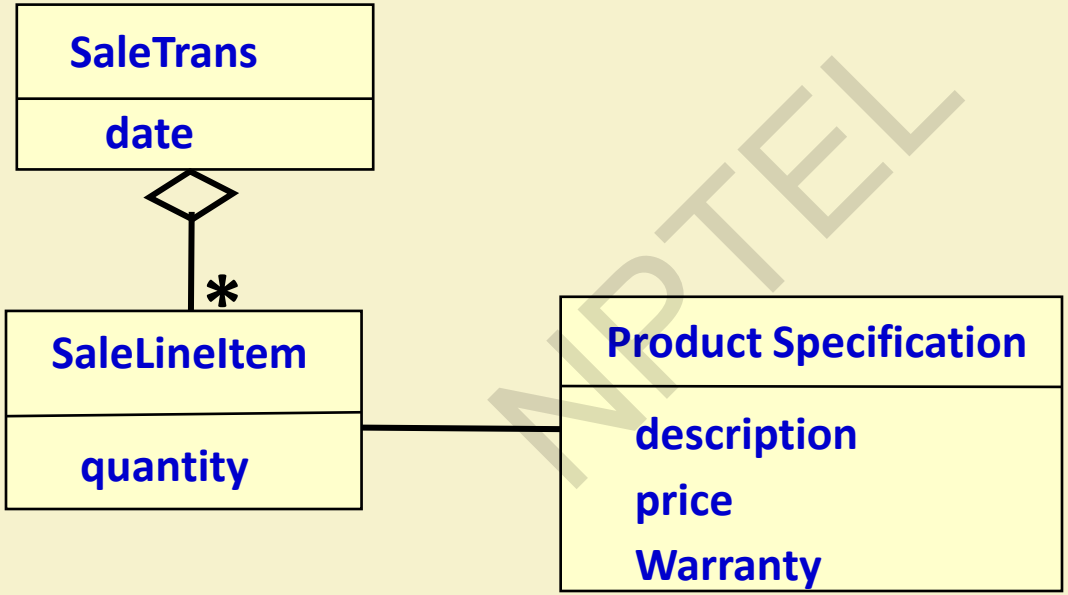
GRASP Patterns

- **Creator**
  - Who creates an object?
- **Information Expert**
  - Which class should be responsible?
- **Low Coupling**
  - Support low dependency and increased reuse
- **Controller**
  - Who handles a system event?
- **High Cohesion**
  - How to keep complexity manageable?
- **Polymorphism**
  - How to handle behavior that varies by type?
- **Pure Fabrication**
  - How to handle a situation, when you do not want to violate High Cohesion and Low Coupling?
- **Indirection**
  - How to avoid direct coupling?
- **Law of Demeter (Don't talk to strangers)**
  - How to avoid knowing about unassociated objects?

Expert Pattern

- **Problem**: Which class should be responsible for doing a certain thing?
- **Solution**:
  - **Assign responsibility to the expert (class that has all/most information necessary to fulfil the required responsibility).**

Which class is information expert for computing total price of a sales transaction?

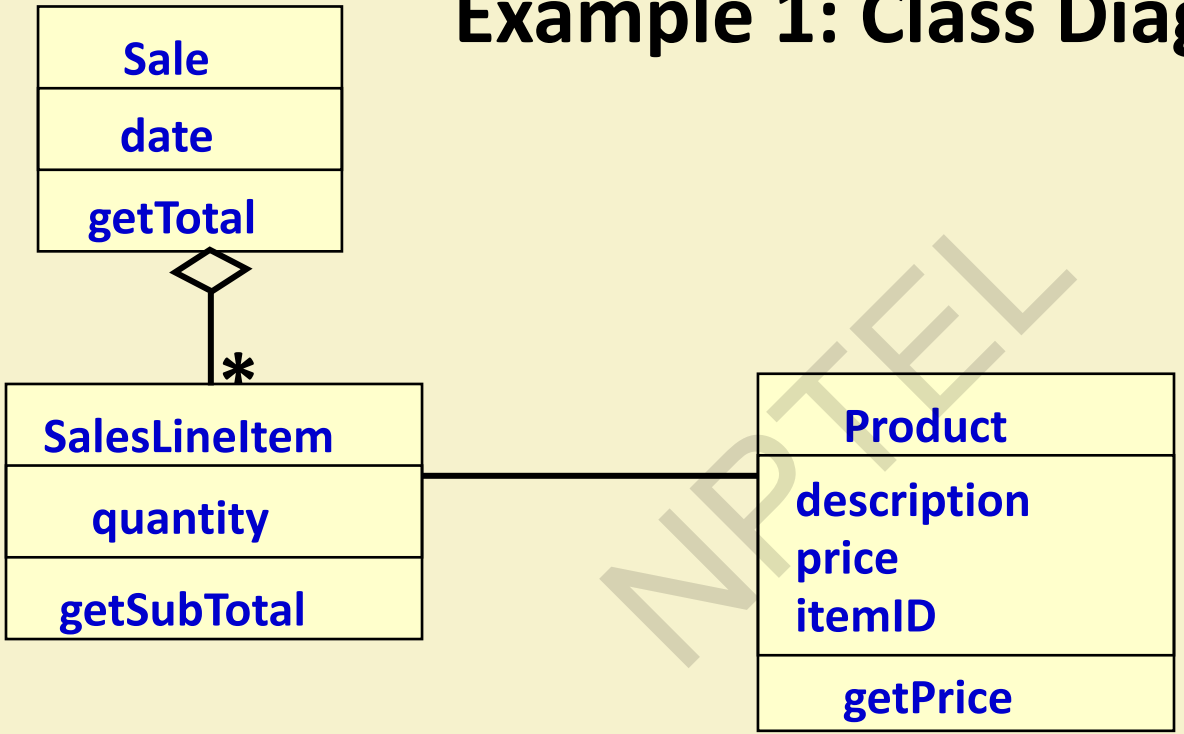


Example 1: Expert

Class Diagram

Collaboration Diagram: Compute Total Price

Example 1: Class Diagram

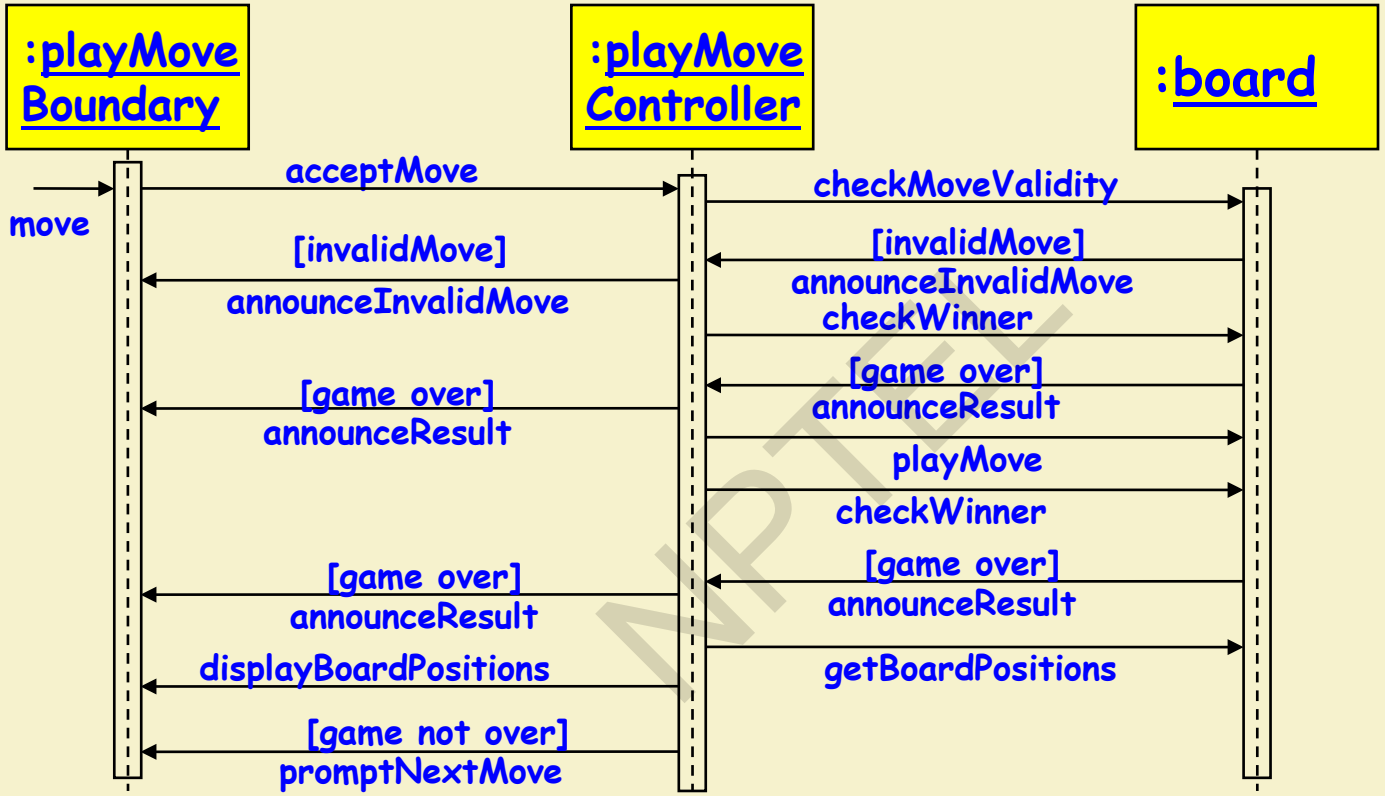


Example 2:Tic-Tac Toe

Initial domain model

Refined domain model

- Which class should check game result after a move?
- Which class should play the computer's move?



Example: Sequence Diagram for the play move use case

Expert Pattern: An Analysis

- **Expert improves cohesion.**
  - Cohesion: The degree to which the information and responsibilities of a class are related to each other
- **How cohesion is improved?**
  - The information needed for a responsibility is in the same class.

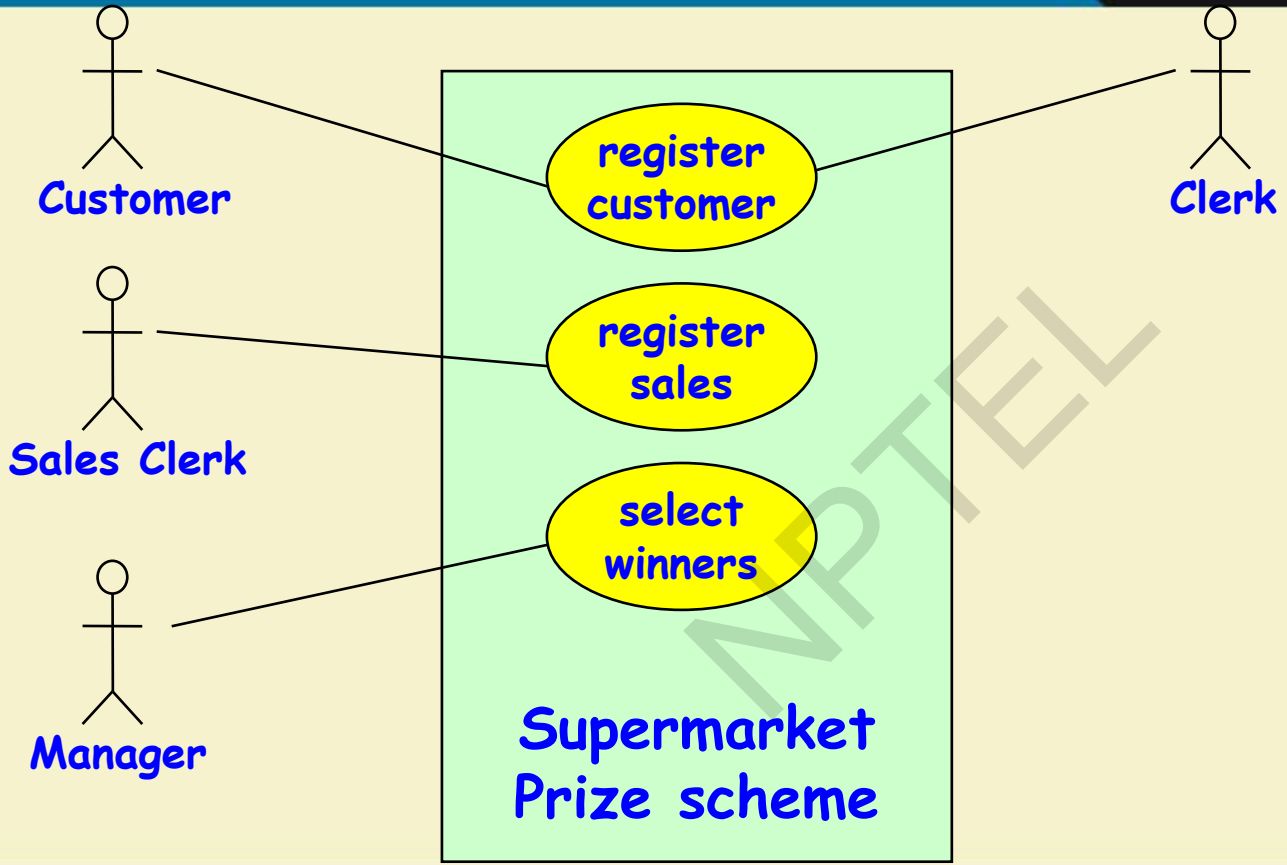
Creator Pattern: Background

- Every object must be created somewhere.
- Consider making a class responsible for creating an object if:
  - **It is an inventory of objects of that type.**
  - **It has the information needed to initialize the object.**
  - **It will be the primary client of the object.**
- Sometimes this pattern suggests a new class.
- In a Library software, who creates a Member?

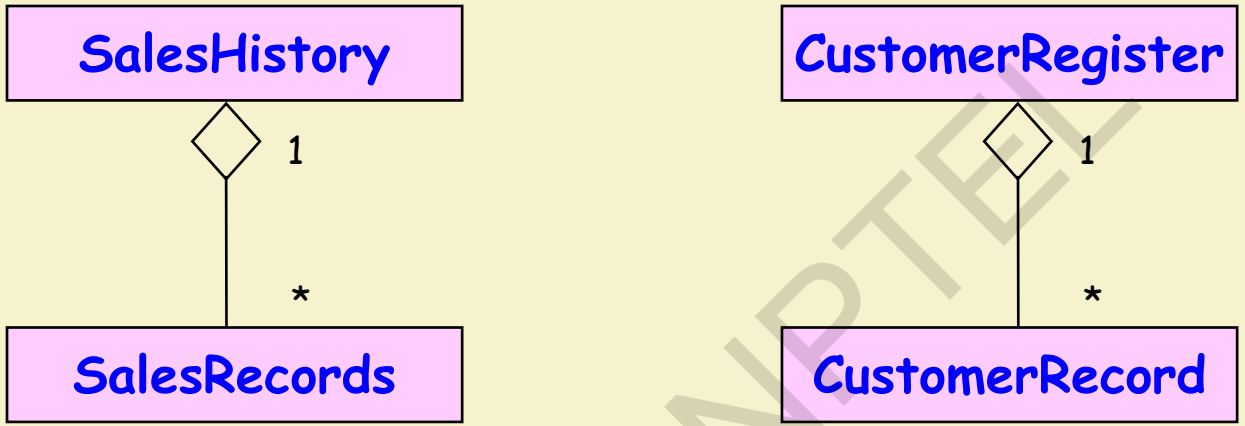
Creator Pattern

- **Problem:** Which class should be responsible for creating a new instance of some class?
- **Solution:** Assign a class C1 the responsibility to create class C2 object if
  - **C1 aggregates objects of type C2**
  - **C1 contains object of type C2**
  - **C1 closely uses C2**
  - **C1 has the initializing data for C2**

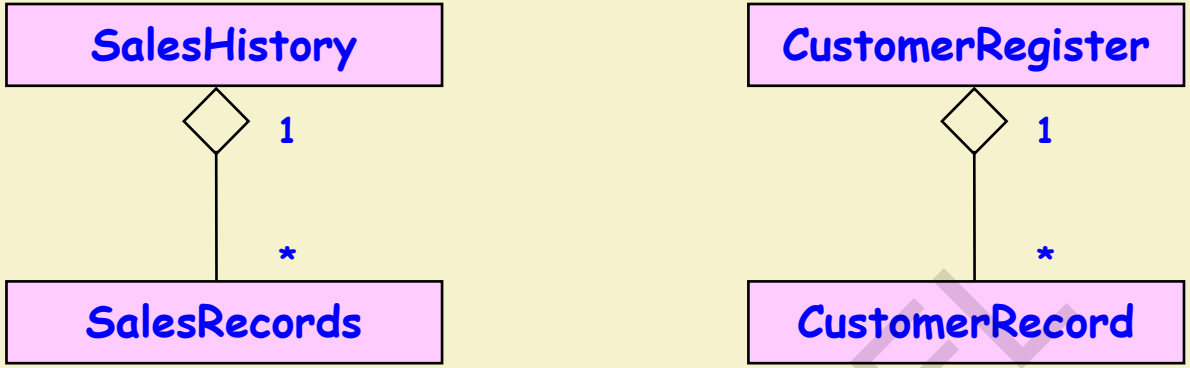
Example: Use Case Model



Solution: Initial Domain Model



Initial domain model



Solution: Refined Domain Model

RegisterCustomerBoundary

RegisterCustomerController

RegisterSalesBoundary

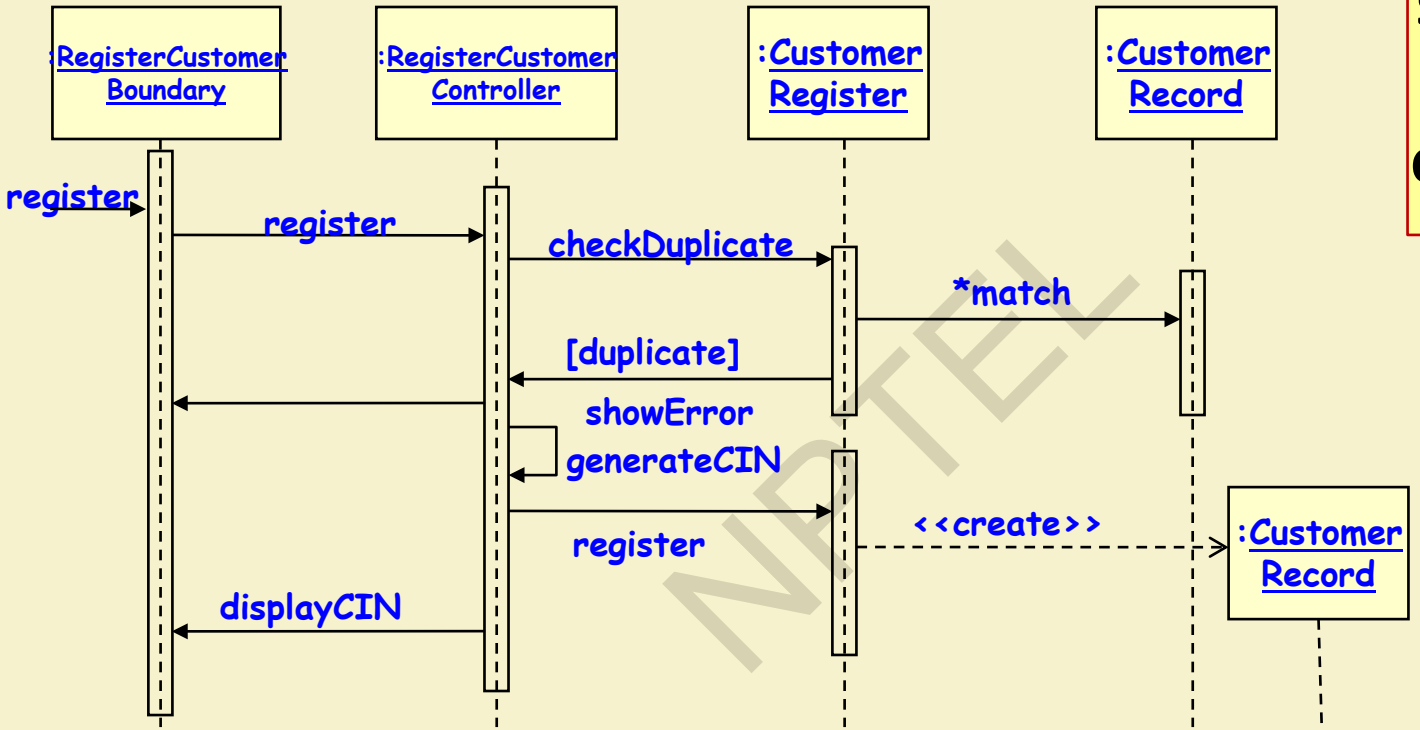
RegisterSalesController

SelectWinnersBoundary

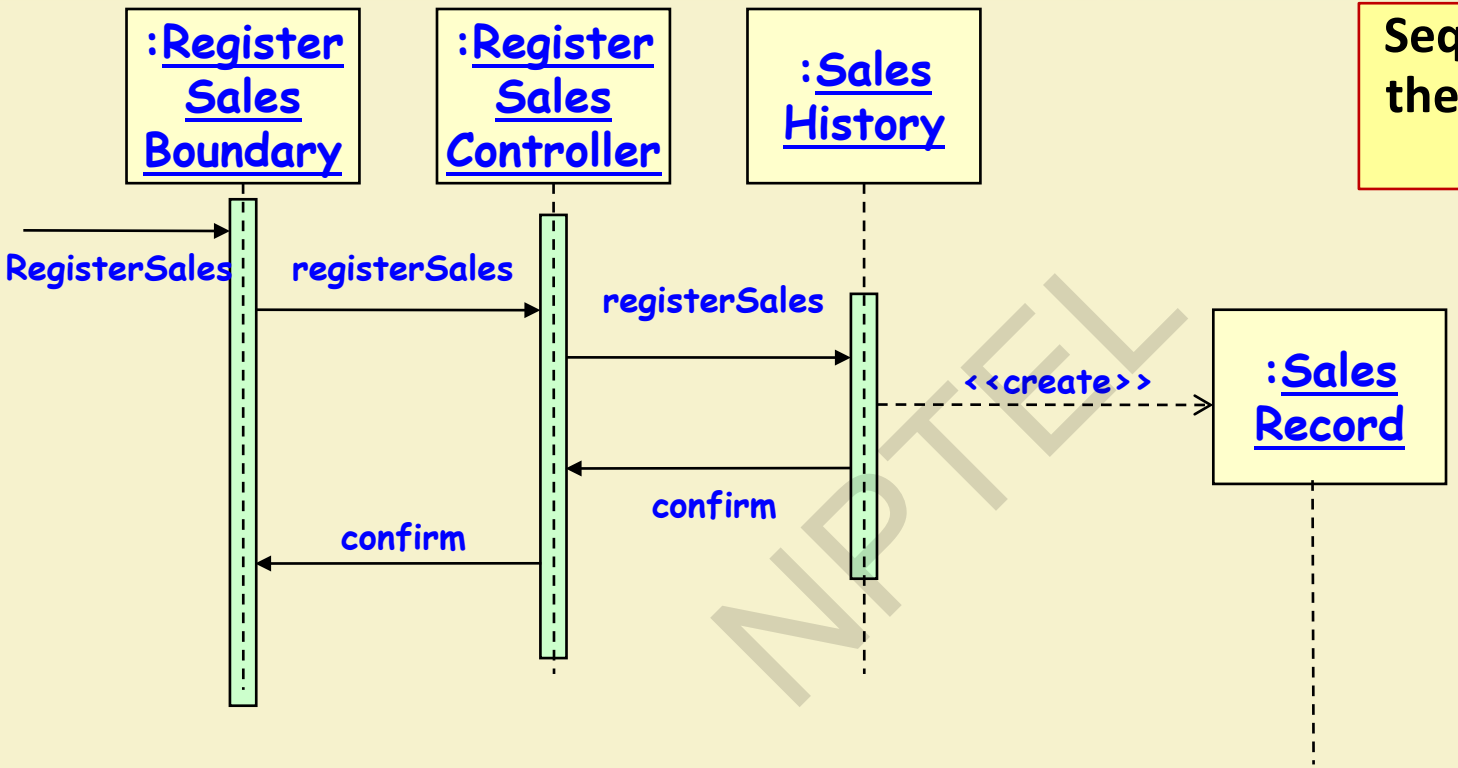
SelectWinnersControllers

Sequence Diagram for the Register

Customer Use Case

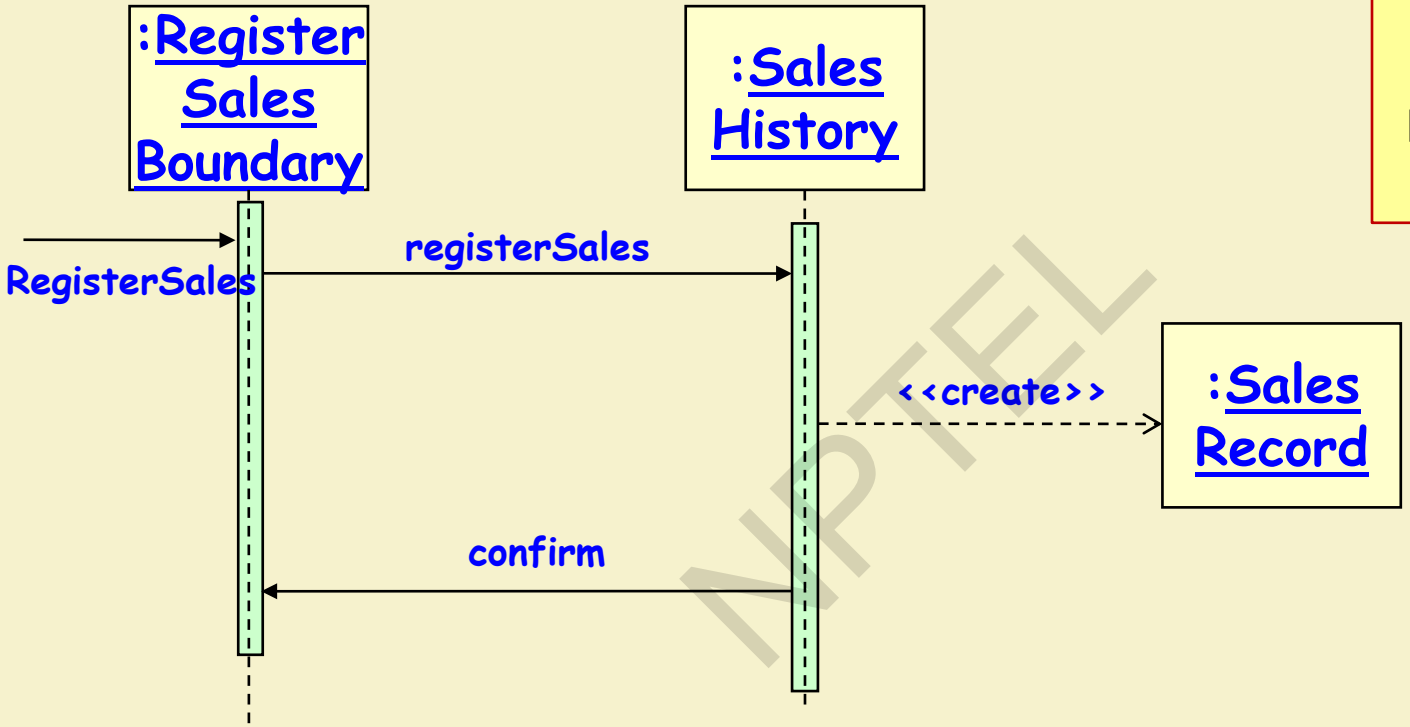


Sequence Diagram for the register customer use case



Sequence Diagram for the Register Sales Use Case

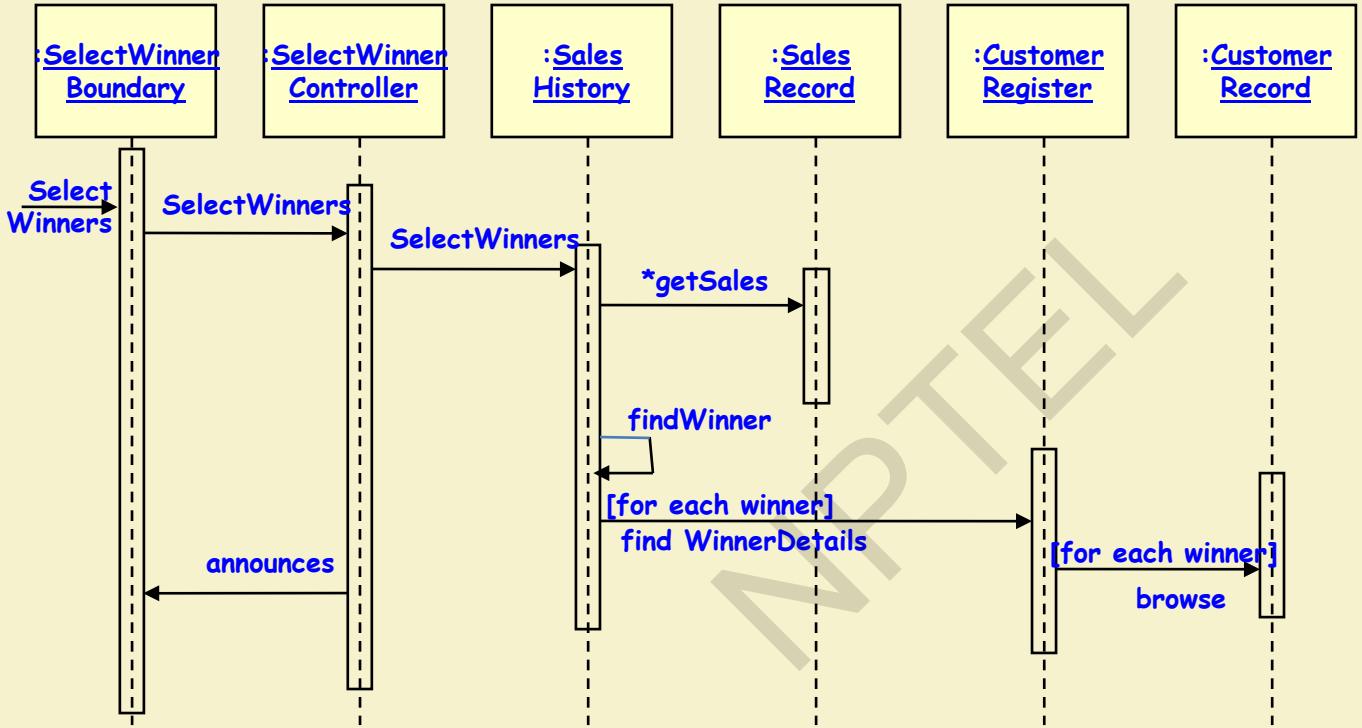
Sequence Diagram for the register sales use case



Refined Sequence Diagram for the register sales use case

Refined Sequence Diagram for the Register Sales Use Case

Sequence Diagram for the Select Winners Use Case



Sequence Diagram for the select winners use case

Contraindications (or Caveats) of Creator Pattern

- Object creation may at times involve significant complexity:
  - Recycling instances for performance reasons
  - Conditionally creating instances from a family of similar classes
- In these situations, other patterns are available…
  - We'll discuss Factory, abstract factory, object pool, and other patterns later…

Creator: An Analysis

- Whenever one class has the responsibility of creating instances of another class:
  - The two classes get coupled
  - Coupling cannot be totally prevented, but we want to reduce overall coupling.
  - Whenever two classes are coupled, one class becomes dependent upon the other.

Creator Pattern

- **Problem:** Which class should be responsible for creating a new instance of some class?
- **Solution:** Assign a class C1 the responsibility to create class C2 object if
  - **C1 aggregates objects of type C2**
  - **C1 contains object of type C2**
  - **C1 closely uses C2**
  - **C1 has the initializing data for C2**

Creator: Final Analysis

- **Creator ensures that coupling due to object instantiation occurs only between closely related classes:**
  - An aggregate or container of a class is already coupled with that class
  - **Therefore, assigning the creation responsibility to the container or aggregate does not worsen the coupling in the design.**

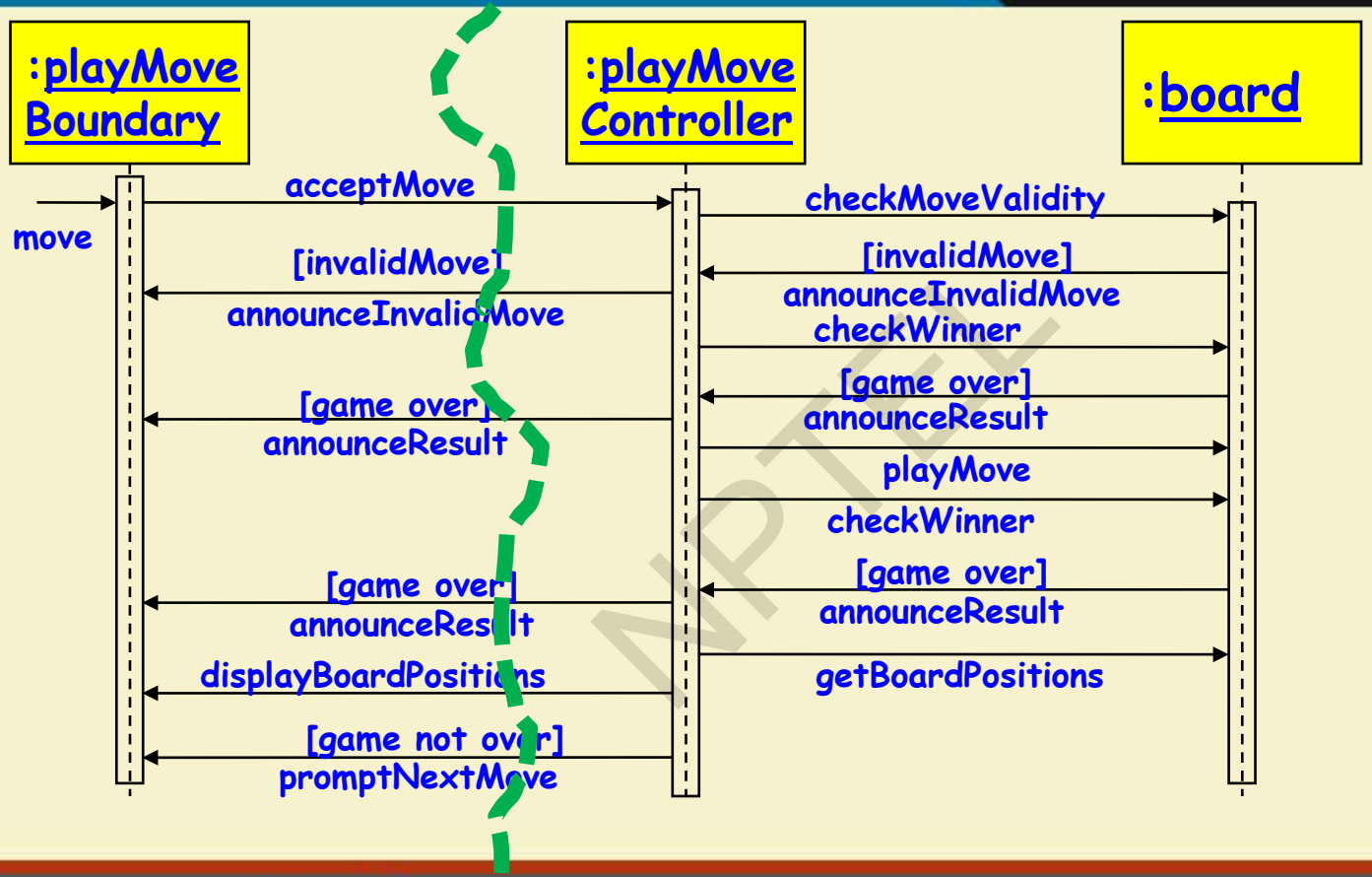
Controller Pattern

- **Problem**: Which class should be responsible for handling the actor requests?

- **Solution**: Have a separate controller object for each use case.

Controller

- Assign the responsibility for receiving and handling an actor message to a synthesized **controller object**
- Do not assign these responsibilities to View classes (windows, dialogs, etc.) or model classes:
  - A user interface object should never be a Controller.
  - Neither should be an entity class.



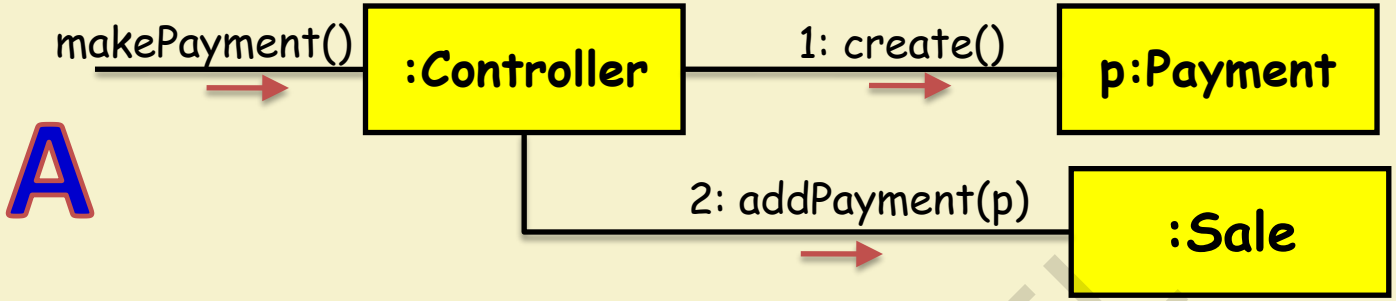
Sequence Diagram for the play move use case

Bloated Controllers

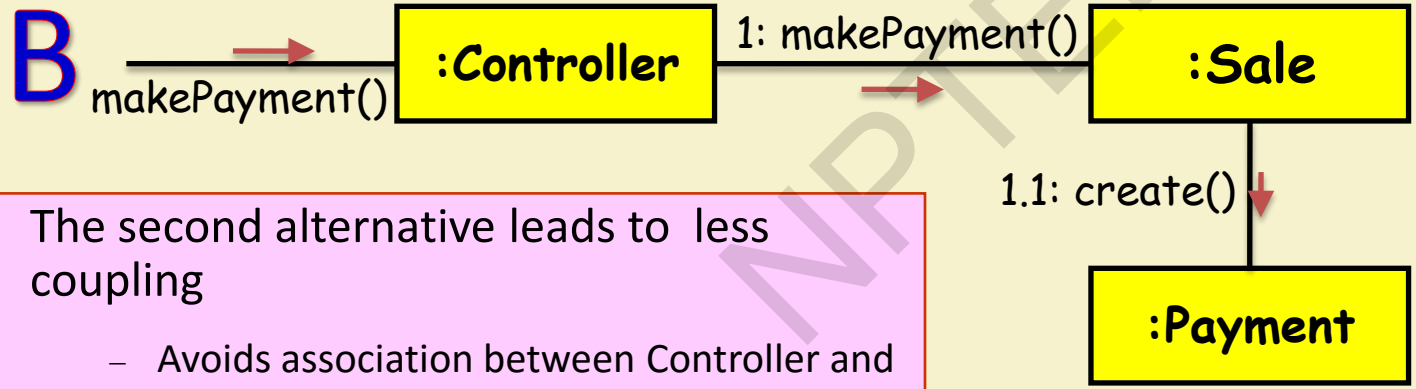
- **Signs of a bloated controller**
  - A single controller receives many system events
  - Controller itself performs tasks instead of delegating them
  - Controller has many attributes, maintains a lot of info about the system or domain
    - Better to distribute these to other objects
- **Cure: Refactor**
  - Add more controllers
  - Delegate responsibility

Low Coupling Pattern

- **Problem:** How to support low dependency, low change impact, and increased re-use?
- **Why is a class having high coupling bad?**
  - Forced to change a class because of changes to related classes
  - Harder to understand a class – cannot do in isolation
  - Harder to re-use --- requires presence of classes that it depends on
- **Solution: Assign responsibilities so that coupling remains low.**



Two alternative designs: "Who creates Payment?"



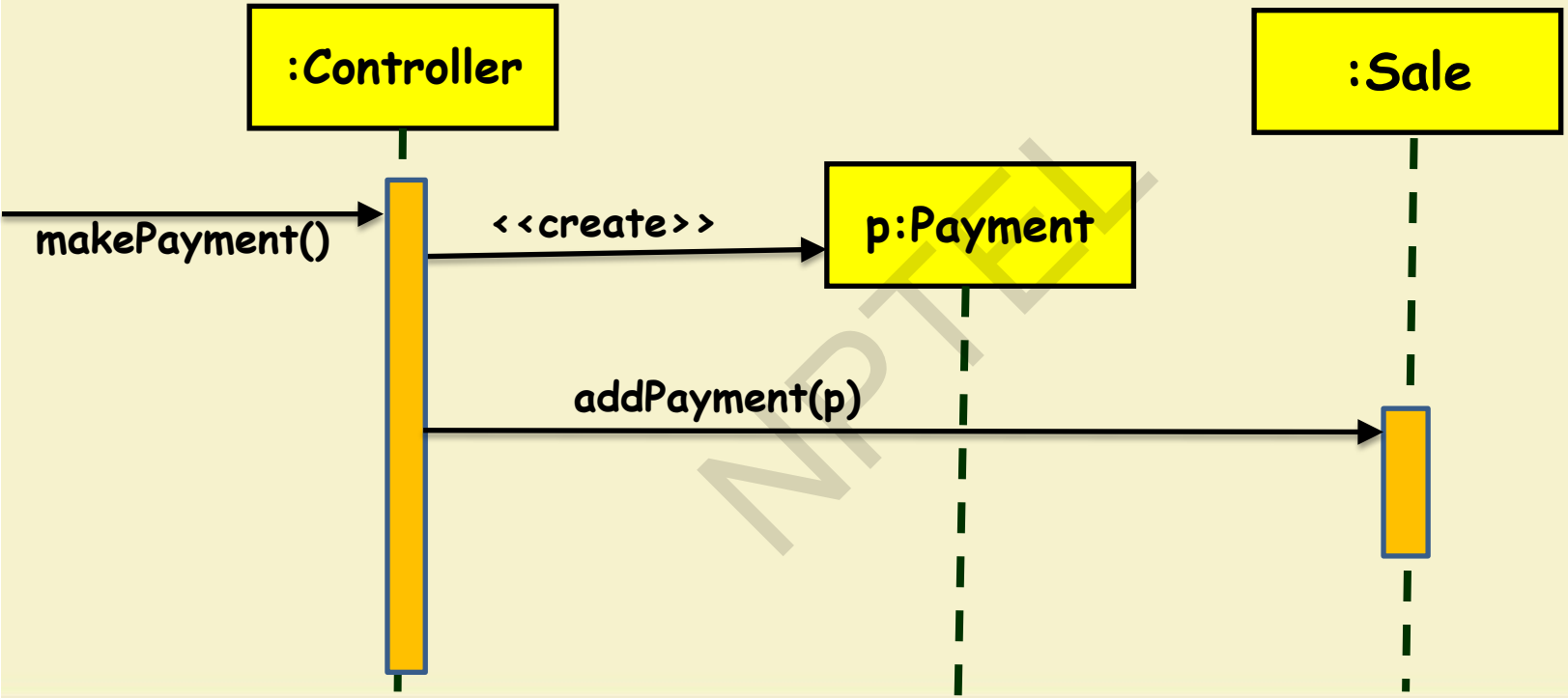
The second alternative leads to less coupling

- Avoids association between Controller and Payment
- Only Sale and Payment are related

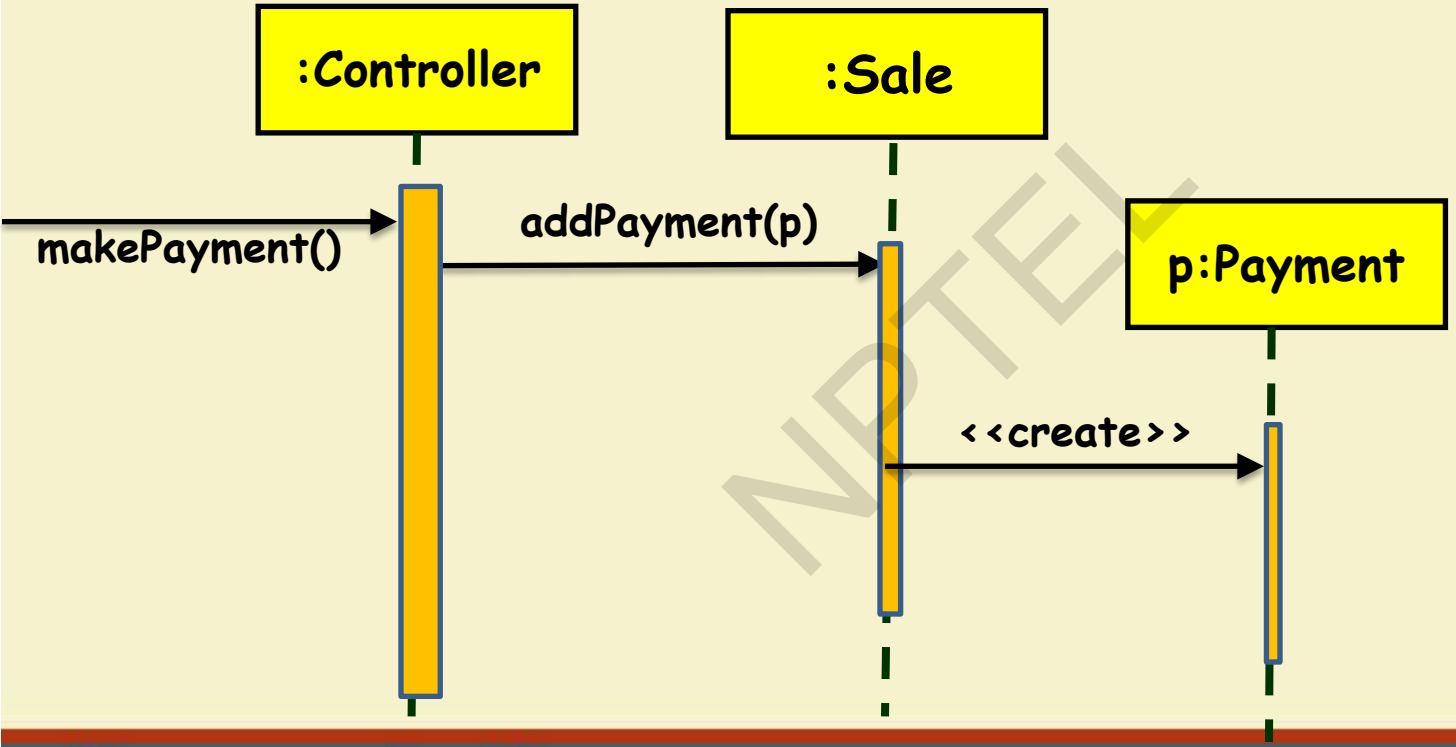
High Cohesion

- Problems with a class with low cohesion:
  - Hard to comprehend
  - Hard to reuse
  - Hard to maintain
  - Constantly affected by change

Analysis: Controller has Diverse Responsibilities



Better design: Higher cohesion



- **High Cohesion**
  - **Problem**: To keep complexity manageable. Classes implement a set of cohesive tasks.
  - **Solution**: Assign focused and related responsibilities to classes.
- **Low Coupling**
  - **Problem**: To support low dependency and increased reuse.
  - **Solution**: Assign responsibilities so that coupling remains low.

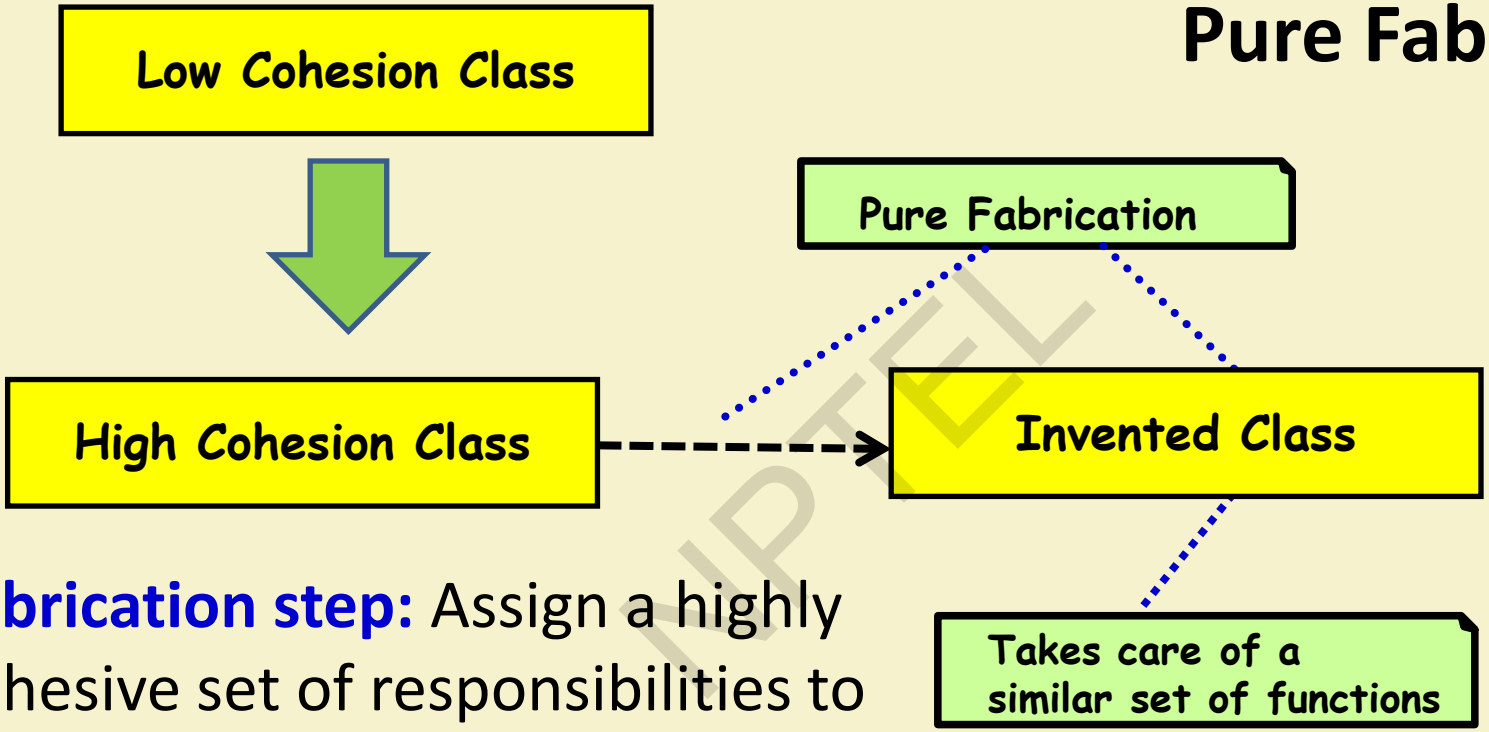
Pure Fabrication – Background

- Suppose a class has responsibilities unrelated to its main task.
  - **It is a Bad design** --- low cohesion and high coupling.
  - **How to improve the design?**

Pure Fabrication

- **Problem:**
  - How to improve a design when a class has very high coupling and low cohesion?
- **Solution:**
  - **Assign a highly cohesive set of responsibilities to an artificial class**
  - May not represent anything in the problem domain...
  - Created only to support high cohesion, low coupling, and reuse...

Pure Fabrication



Fabrication step: Assign a highly cohesive set of responsibilities to an artificial class.

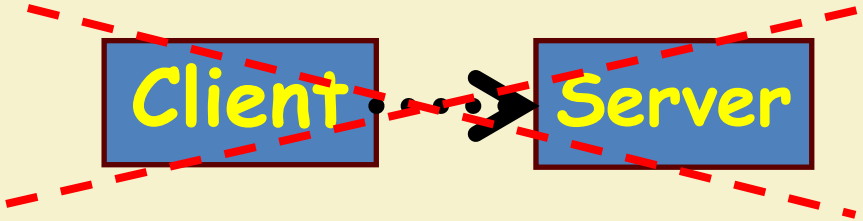
Indirection Pattern

Problem:

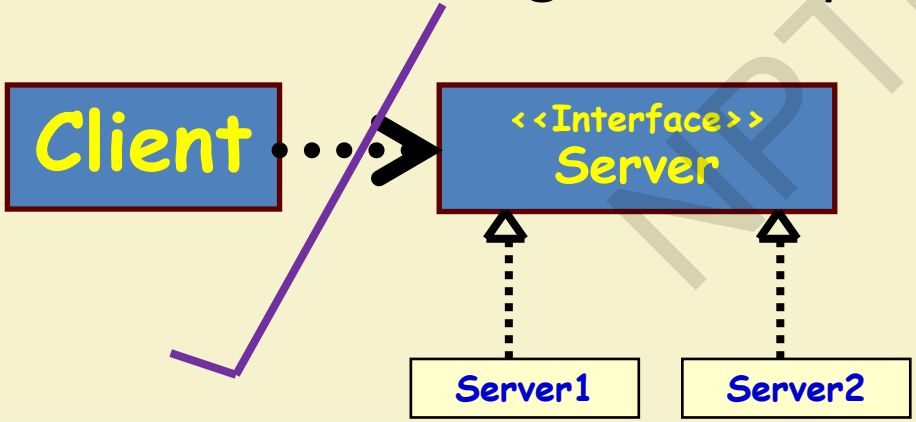
- **How to avoid direct coupling between classes?**
- How to decouple objects so that low coupling is achieved and changes become easy?

Solution:

- Depend on an interface class,
  - So that objects are not directly coupled.

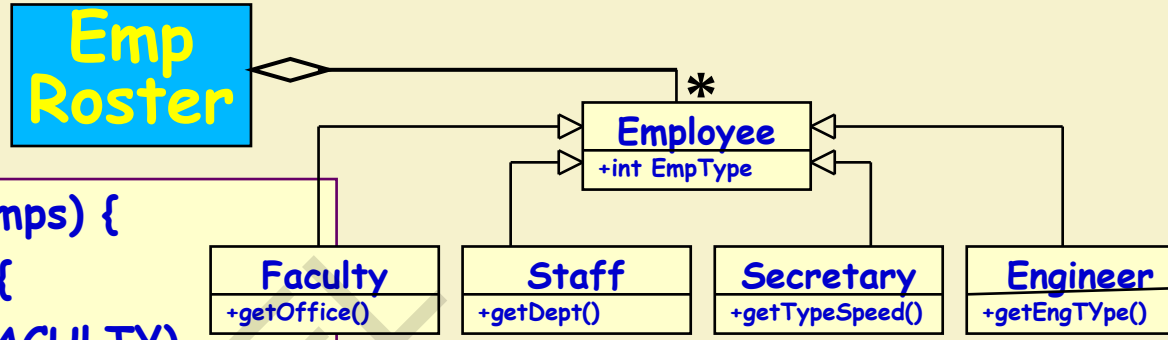


- **Indirection Pattern violation:**
  - Server can change and requires changes to the client.



Indirection Pattern Compliant

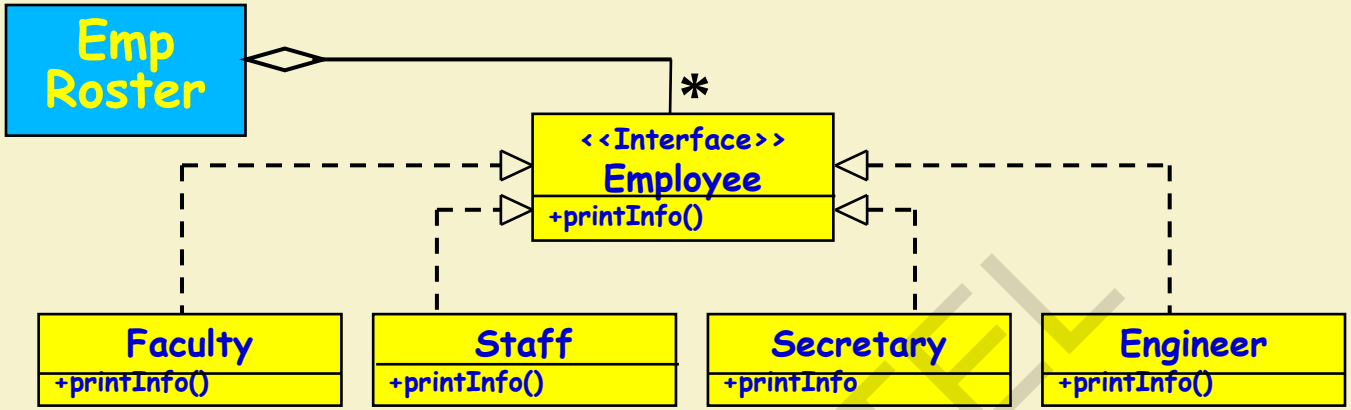
Example: Naïve Solution




```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        if (emps[i].empType == FACULTY)
            printfFaculty((Faculty)emps[i]);
        else if (emps[i].empType ==STAFF)
            printStaff((Staff)emps[i]);
        else if (emps[i].empType == SECRETARY)
            printSecretary((Secretary)emps[i]);
    }
}
```

What if we need to add Engineer??

Indirection Pattern Solution




```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        emps[i].printInfo();
    }
}
```

When Engineer is added, printEmpRoster() does not even need to recompile. PrintEmpRoster() is open to extension, closed for modification.

Indirection Advantages

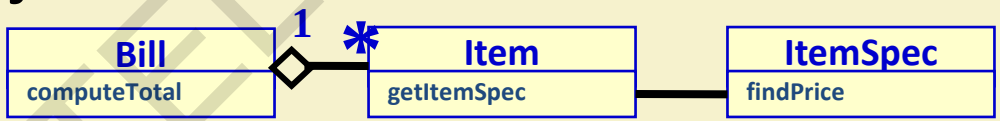
- Low coupling
- Promotes reusability

Law of Demeter

Problem:

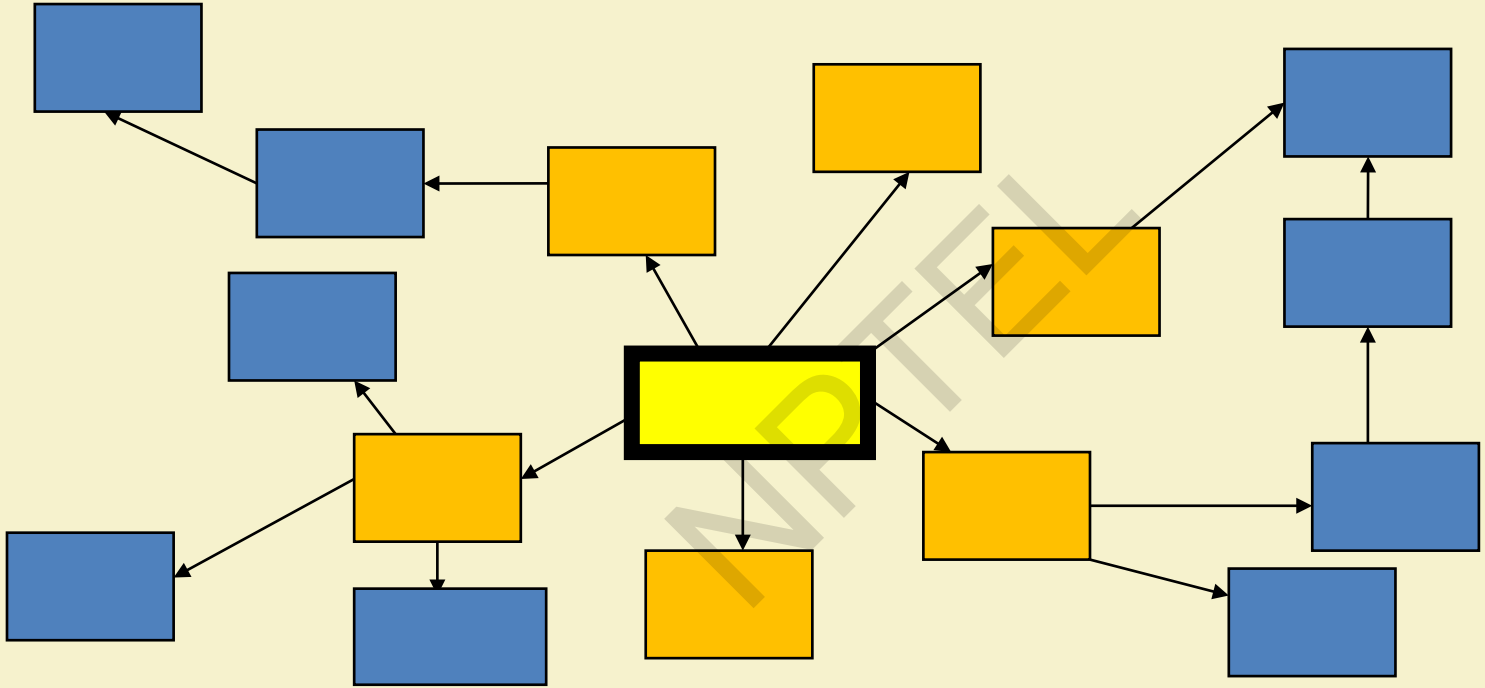
- How to avoid a class from interacting with indirectly associated objects?

Solution:



- If two classes have no other reason to be directly aware of each other:
  - Then the two classes should not directly interact.

Collaborators



Law of Demeter

- In a method, calls should only be made to the methods of following objects:
  - **This object (or self)**
  - **An object parameter of the method**
  - **An object attribute of self**
  - **An object created within the method**

A Hypothetical example

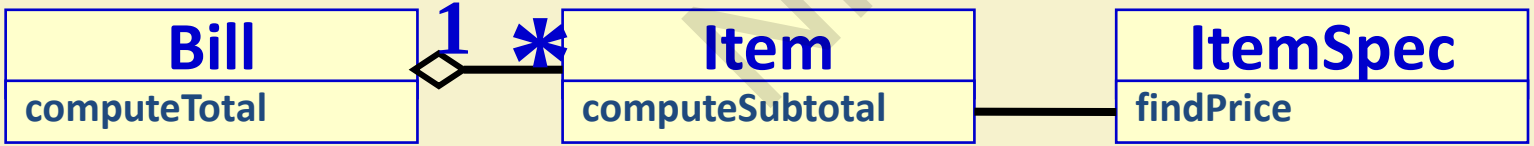
```
class A {
  private B b = new B();

  public void m() {
    this.b.c.foo();  // High
                     coupling: bad
  }
}
```

```
class B {
  C c;
}
class C {
  public void foo() {
  }
}
```

Law of Demeter: Example

Violation: item.getItemspec().findprice()



LoD Programming Style

- In a class PaperBoy,
  - **do not use** customer.getWallet().getCash(due);
  - **rather use** customer.getPayment(due); and in customer.getPayment (due), use wallet.getCash(due)
- **Benefit:**
  - Easier analysis
- **Tradeoff:**
  - More statements, but simpler statements
- Experiments show significantly improved maintainability.

Law of Demeter: Final Analysis

- Reduces coupling between classes
- Adds a small amount of overhead in the form of indirect method calls

"The basic effect of applying this Law is the creation of loosely coupled classes, whose implementation secrets are encapsulated. Such classes are fairly unencumbered, meaning that to understand the working of one class, you need not understand the details of many other classes." **Grady Booch**

Polymorphism

- **Problem:**
  - How to handle alternative operations based on subtypes?
- **Solution:**
  - When alternate behaviours are selected based on the type of an object,
  - **Use polymorphic method call to select the behaviour,**
  - **Rather than using if statement to test the type.**

Polymorphism Pattern Advantage

- Easily extendible as compared to using explicit selection logic