(low-level)

Design & Implementation

Software Engineering 2

(3103313-1)

Amirkabir University of Technology

Fall 1399-1400

The Design

- Architectural Design (Software Architecture)
  - Subsystems (Components)
  - Interfaces
  - Deployment and Component Diagram
- Structural Models
  - Design Classes, Units, Modules, …
- Dynamic Models
  - Sequence of services, interactions, state-based behaviour, …
- User Interface

Design Principles/Concepts

- Abstraction
- Modularisation (Low Coupling, High Cohesion)
- Information hiding
- Hierarchical structures
- Limit Complexity

- Refactoring
- Patterns
- Idioms (language-specific)

PDFConverter

An Example Scenario

- Create an application that converts Word documents to PDFs.

PDFConverter

An Example Scenario

- Create an application that converts Word documents to PDFs.

```
6  public class PDFConverter {
7
8      /**
9       * This method accepts as input the document to be converted and
10      * returns the converted one.
11      * @param fileBytes
12      * @throws Exception
13      */
14     public byte[] convertToPDF(byte[] fileBytes) throws Exception {
15         // We're sure that the input is always a WORD. So we just use
16         //aspose.words framework and do the conversion.
17
18         InputStream input = new ByteArrayInputStream(fileBytes);
19         com.aspose.words.Document wordDocument = new com.aspose.words.Document(input);
20         ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
21         wordDocument.save(pdfDocument, SaveFormat.PDF);
22         return pdfDocument.toByteArray();
23     }
24 }
```

PDFConverter

An Example Scenario

- Requirements Change, as Always; support Excel documents as well.

```
6  public class PDFConverter {
7
8      /**
9       * This method accepts as input the document to be converted and
10      * returns the converted one.
11      * @param fileBytes
12      * @throws Exception
13      */
14     public byte[] convertToPDF(byte[] fileBytes) throws Exception {
15         // We're sure that the input is always a WORD. So we just use
16         //aspose.words framework and do the conversion.
17
18         InputStream input = new ByteArrayInputStream(fileBytes);
19         com.aspose.words.Document wordDocument = new com.aspose.words.Document(input);
20         ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
21         wordDocument.save(pdfDocument, SaveFormat.PDF);
22         return pdfDocument.toByteArray();
23     }
24 }
```

PDFConverter

An Example Scenario

- Requirements Change, as Always; support Excel documents as well.

```
1  public class PDFConverter {
2      // we didn't mess with the existing functionality, by default
3      // the class will still convert WORD to PDF, unless the client sets
4      // this field to EXCEL.
5      public String documentType = "WORD";
6
7      /**
8       * This method accepts as input the document to be converted and
9       * returns the converted one.
10      * @param fileBytes
11      * @throws Exception
12      */
13     public byte[] convertToPDF(byte[] fileBytes) throws Exception {
14         if (documentType.equalsIgnoreCase("WORD")) {
15             InputStream input = new ByteArrayInputStream(fileBytes);
16             com.aspose.words.Document wordDocument = new com.aspose.words.Document(input);
17             ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
18             wordDocument.save(pdfDocument, SaveFormat.PDF);
19             return pdfDocument.toByteArray();
20         } else {
21             InputStream input = new ByteArrayInputStream(fileBytes);
22             Workbook workbook = new Workbook(input);
23             PdfSaveOptions saveOptions = new PdfSaveOptions();
24             saveOptions.setCompliance(PdfCompliance.PDF_A_1_B);
25             ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
26             workbook.save(pdfDocument, saveOptions);
27             return pdfDocument.toByteArray();
28         }
29     }
30 }
```

PDFConverter

An Example Scenario

- Requirements Change, as Always; support Excel documents as well.

```
1  public class PDFConverter {
2      // we didn't mess with the existing functionality, by
3      // the class will still convert WORD to PDF, unless
4      // this field to EXCEL.
5      public String documentType = "WORD";
6
7      /**
8       * This method accepts as input the document to
9       * returns the converted one.
10      * @param fileBytes
11      * @throws Exception
12      */
13     public byte[] convertToPDF(byte[] fileBytes) th
14         if (documentType.equalsIgnoreCase("WORD")) {
15             InputStream input = new ByteArrayInputStream(fileBy
16             com.aspose.words.Document wordDocument = new com.aspose.wo
17             ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
18             wordDocument.save(pdfDocument, SaveFormat.PDF);
19             return pdfDocument.toByteArray();
20         } else {
21             InputStream input = new ByteArrayInputStream(fileBytes);
22             Workbook workbook = new Workbook(input);
23             PdfSaveOptions saveOptions = new PdfSaveOptions();
24             saveOptions.setCompliance(PdfCompliance.PDF_A_1_B);
25             ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
26             workbook.save(pdfDocument, saveOptions);
27             return pdfDocument.toByteArray();
28         }
29     }
30 }
```

- Code repetition
- Rigidity
- Immobility
- Coupling between the high-level module and the frameworks

PDFConverter

An Example Scenario

- Doing it the right way ☺

PDFConverter

An Example Scenario

- Doing it the right way ☺

```java
/**
 * This interface represents an abstract algorithm for converting
 * any type of document to a PDF.
 * @author Hussein
 *
 */
public interface Converter {

    public byte[] convertToPDF(byte[] fileBytes) throws Exception;
}
```

PDFConverter

An Example Scenario

- Doing it the right way ☺

```
/**
 * This interface represents an abstract algorithm for converting
```

```
/**
 * This class holds the algorithm for converting Excel
 * documents to PDFs.
 * @author Hussein
 *
 */
public class ExcelPDFConverter implements Converter {

    public byte[] convertToPDF(byte[] fileBytes) throws Exception {
        InputStream input = new ByteArrayInputStream(fileBytes);
        Workbook workbook = new Workbook(input);
        PdfSaveOptions saveOptions = new PdfSaveOptions();
        saveOptions.setCompliance(PdfCompliance.PDF_A_1_B);
        ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
        workbook.save(pdfDocument, saveOptions);
        return pdfDocument.toByteArray();
    };
}
```

PDFConverter

An Example Scenario

- Doing it the right way ☺

```
/**
 * This interface represents an abstract algorithm for converting
```

```
/**
 * This class holds the algorithm for converting Excel
```

```
/**
 * This class holds the algorithm for converting Word
 * documents to PDFs.
 * @author Hussein
 *
 */
public class WordPDFConverter implements Converter {

    @Override
    public byte[] convertToPDF(byte[] fileBytes) throws Exception {
        InputStream input = new ByteArrayInputStream(fileBytes);
        com.aspose.words.Document wordDocument = new com.aspose.words.Document(input);
        ByteArrayOutputStream pdfDocument = new ByteArrayOutputStream();
        wordDocument.save(pdfDocument, SaveFormat.PDF);
        return pdfDocument.toByteArray();
    }
}
```

PDFConverter

An Example Scenario

- Doing it the right way ☺

```java
public class PDFConverter {

    /**
     * This method accepts the document to be converted as an input and
     * returns the converted one.
     * @param fileBytes
     * @throws Exception
     */
    public byte[] convertToPDF(Converter converter, byte[] fileBytes) throws Exception {
        return converter.convertToPDF(fileBytes);
    }
}
```

Principles

- KISS: **K**eep **I**t **S**imple, **S**tupid
- DRY: don't repeat yourself
- SOLID
  - **S**ingle responsibility
  - **O**pen/closed
  - **L**iskov substitution
  - **I**nterface segregation
  - **D**ependency inversion
- …

Interface segregation

- Application's interfaces should always be kept smaller and separate from one another; no client should be forced to depend on methods it does not use.

- Example
  - *Athlete* interface is an interface with some actions of an athlete.

```
public interface Athlete {

    void compete();

    void swim();

    void highJump();

    void longJump();

}
```

```java
public interface Athlete {

    void compete();

    void swim();

    void highJump();

    void longJump();

}
```

```java
public class JohnDoe implements Athlete {

    @Override
    public void compete() {
        System.out.println("John Doe started competing");
    }

    @Override
    public void swim() {
        System.out.println("John Doe started swimming");
    }

    @Override
    public void highJump() {
    }

    @Override
    public void longJump() {
    }
}
```

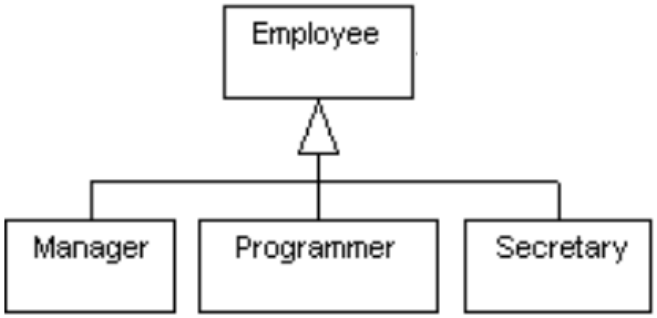
```
public interface Athlete {

void compete();
}

public interface SwimmingAthlete extends Athlete {

    void swim();

}

public interface JumpingAthlete extends Athlete {

    void highJump();

    void longJump();

}
```

```
public class JohnDoe implements SwimmingAthlete {

    @Override
    public void compete() {
        System.out.println("John Doe started competing");
    }

    @Override
    public void swim() {
        System.out.println("John Doe started swimming");
    }

}
```

Composite Reuse Principle (CRP)

- Favor polymorphic composition

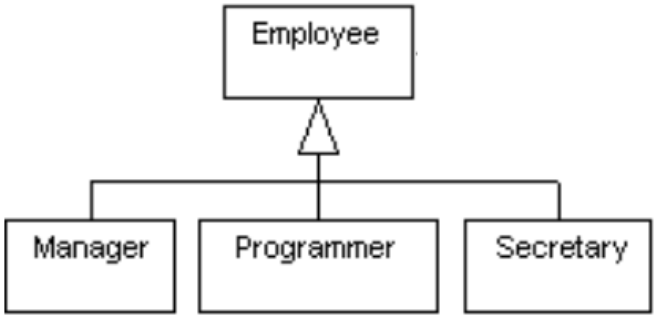
of objects over inheritance.

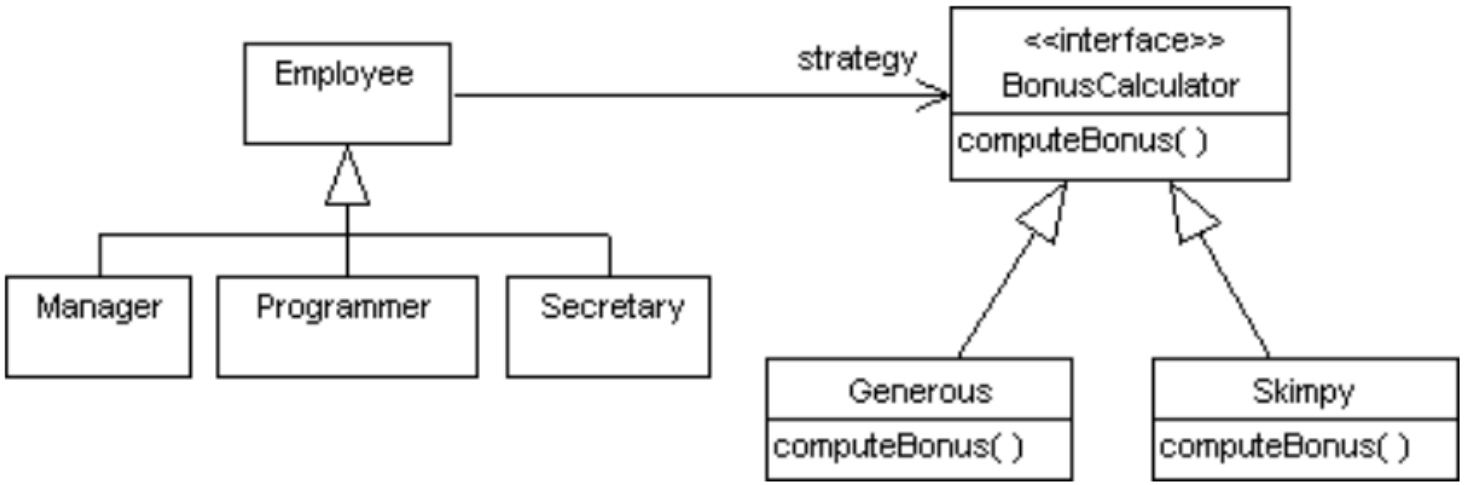


Composite Reuse Principle (CRP)

- Favor polymorphic composition

of objects over inheritance.



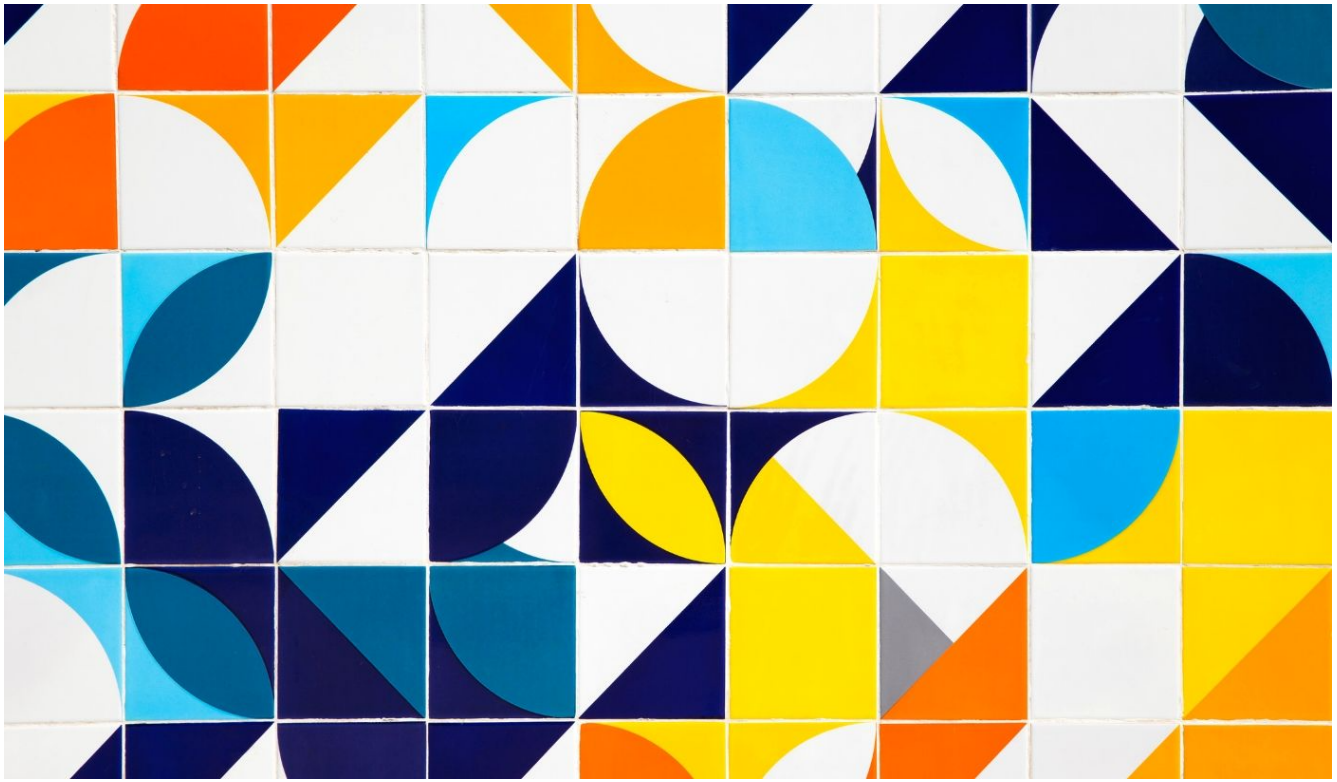


Principle of Least Knowledge (PLK)

- For an operation O on a class C, only operations on the following objects should be called:
  - itself,
  - its parameters,
  - objects it creates, or
  - its contained instance objects.

- Also known as the Law of Demeter

- Avoid calling any methods on an object where the reference to that object is obtained by calling a method on another object.



Design Patterns

Patterns

- A **pattern** is named and well-known problem/solution pair that
  - Can be applied in new contexts
  - With advice on how to apply it in novel situations
  - With a discussion of its trade-offs, implementations, variations, …
  - Names facilitate communication about software solutions.
- Some patterns may seem obvious: That's a good thing!
- **Design Patterns-** medium-level strategies that are used to solve design problems.

Well-known Pattern Families

OO Languages

- GRASP - General Responsibility Assignment Software Patterns (or Principles)
  - 9 Patterns

- GoF - Design Patterns: Elements of Reusable Object-Oriented Software
  - GoF:
    - Erich Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides
  - 23 patterns
    - Structural
    - Behavioral
    - Functional

Responsibility

Doing and Knowing

"… an obligation to perform a task or know information …"

- Doing responsibility of an object is seen as
  - doing something itself – create an object, process data, do some computation/calculation
  - initiate and coordinate actions with other objects

- Knowing responsibility of an object can be defined as
  - private and public object data
  - related objects references
  - things it can derive

GRASP

General Responsibility Assignment Software Patterns

- Guides in assigning responsibilities to collaborating objects.

- 9 GRASP patterns
  1. Information Expert
  2. Creator
  3. Controller
  4. Low Coupling
  5. High Cohesion
  6. Indirection
  7. Polymorphism
  8. Protected Variations
  9. Pure Fabrication

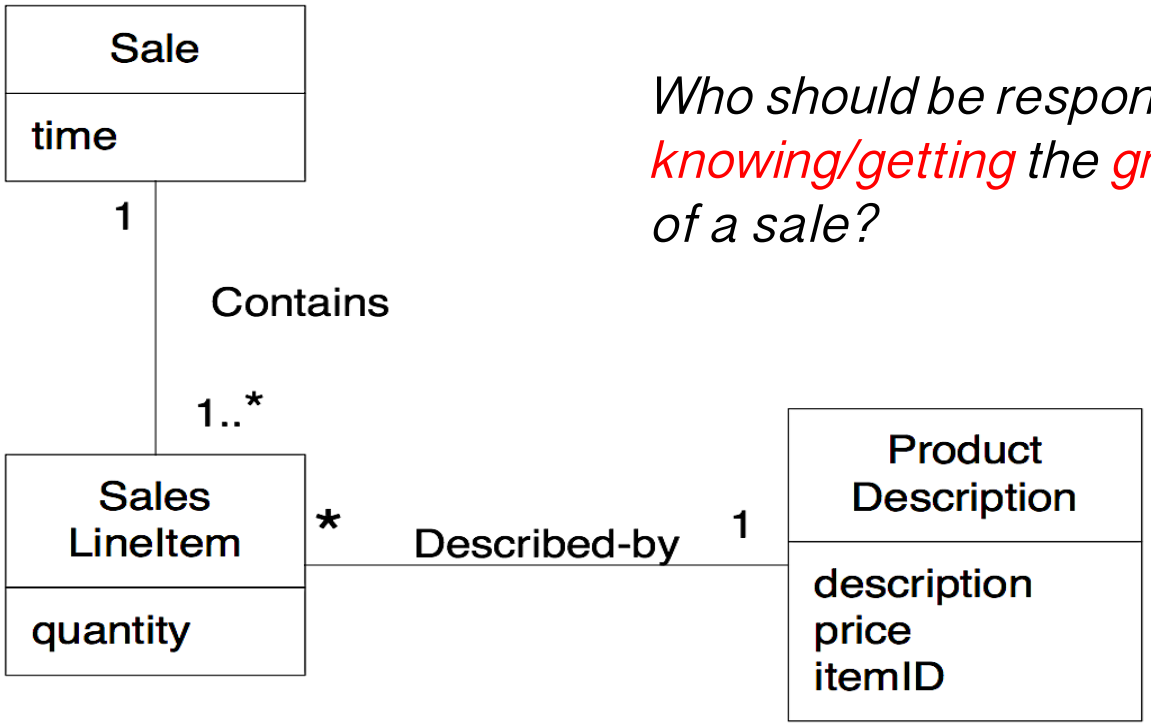
Reference: Larman, C., Applying *UML and Patterns: An introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd Ed. Prentice-Hall, 2004.

1. Information Expert

Problem: What is a basic principle by which to assign responsibilities to objects?
Solution: Assign a responsibility to the class that has the information needed to fulfill it.

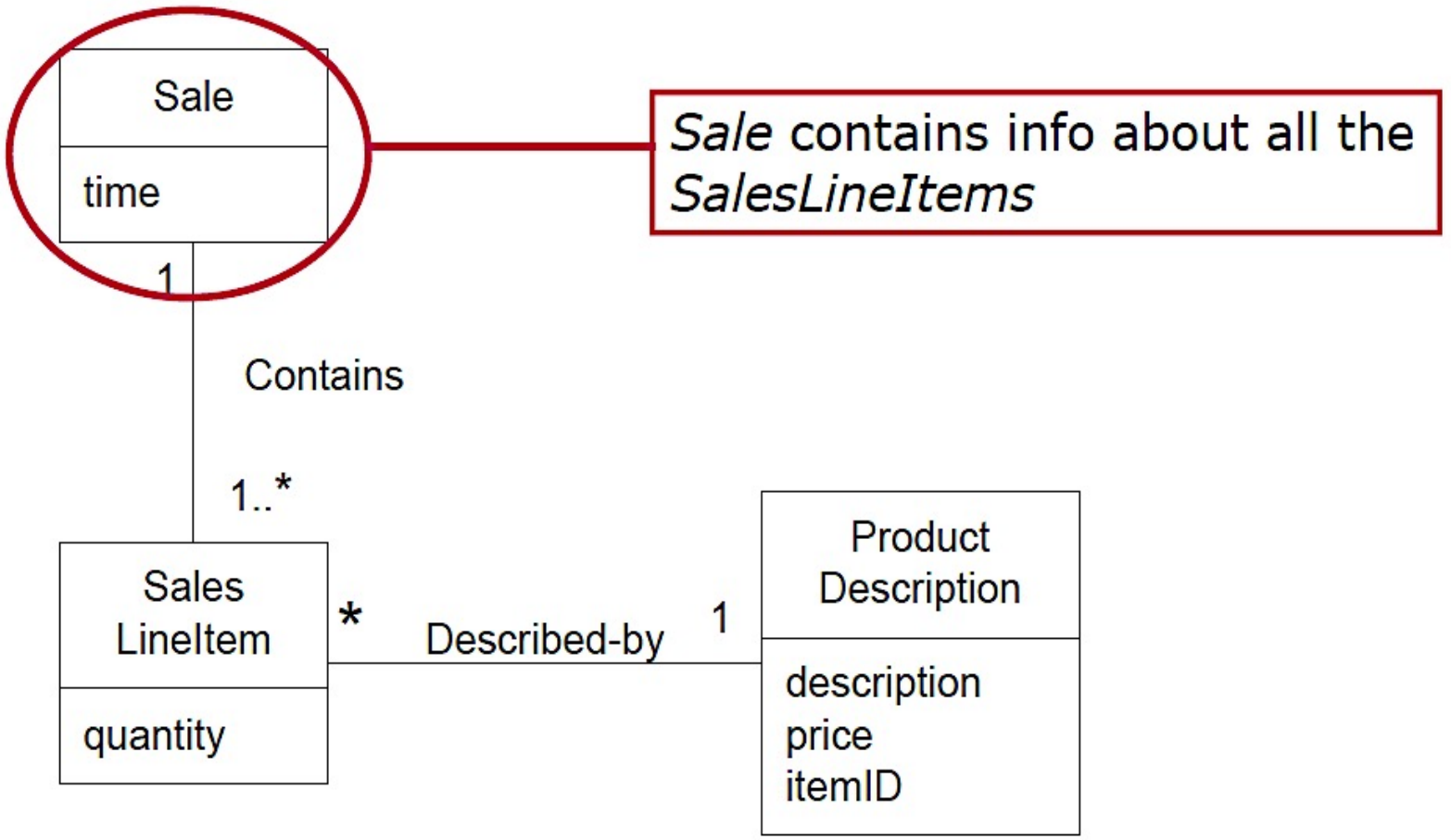
1. Information Expert

Problem: What is a basic principle by which to assign responsibilities to objects?
Solution: Assign a responsibility to the class that has the information needed to fulfill it.

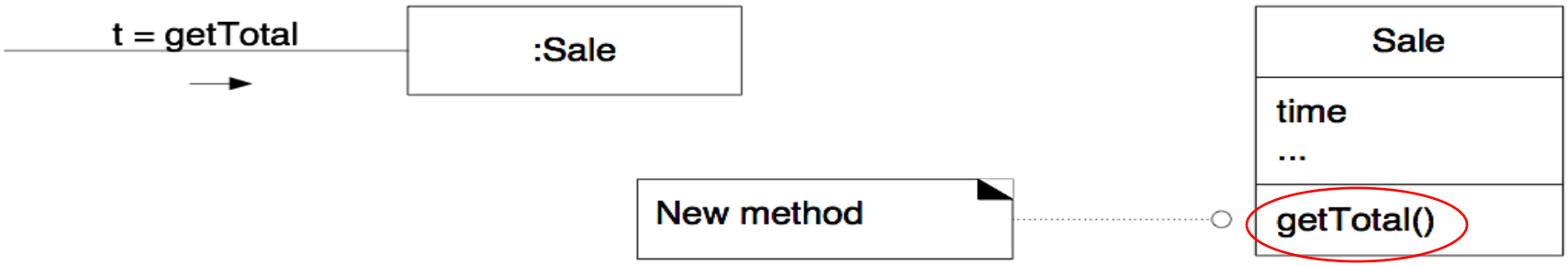


Who should be responsible for knowing/getting the grand total of a sale?

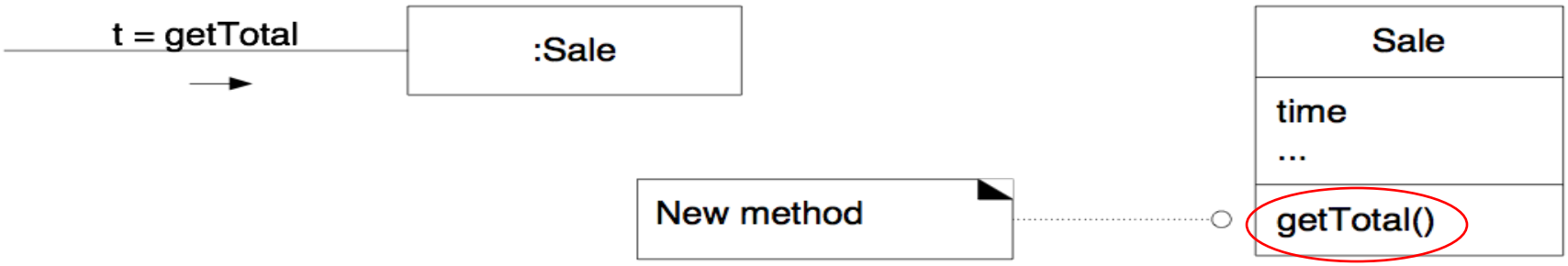
1. Who should be responsible for knowing/getting the grand total of a sale?



1. Who should be responsible for knowing/getting the grand total of a sale?

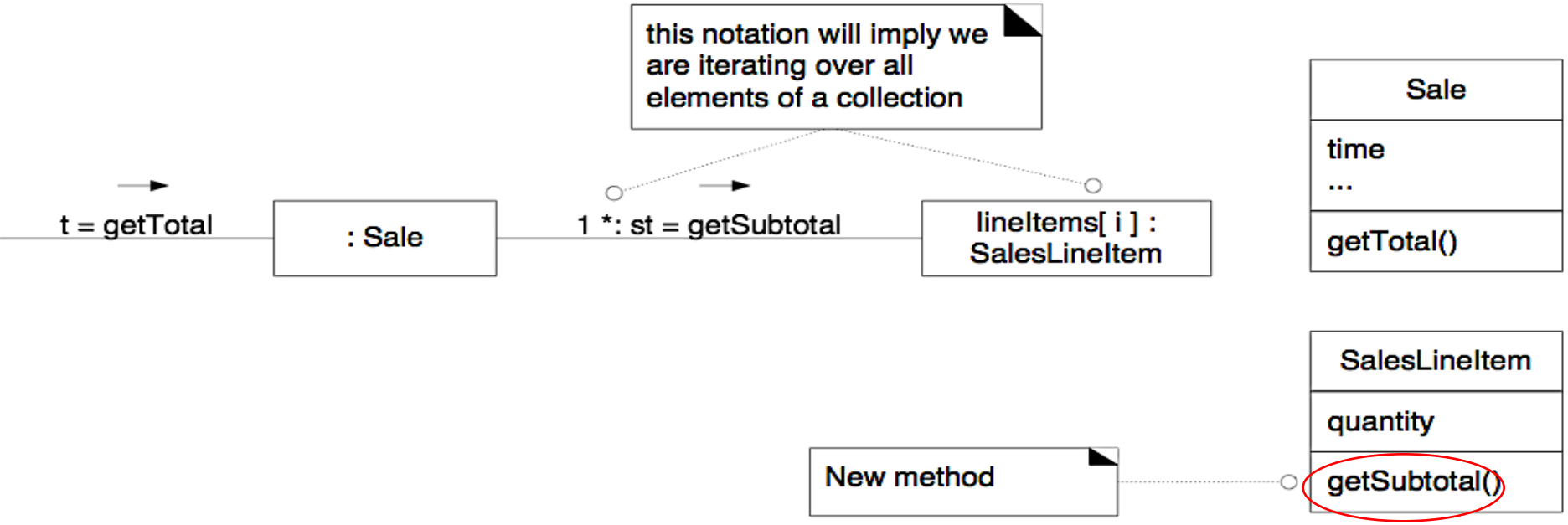


1. Who should be responsible for knowing/getting the grand total of a sale?

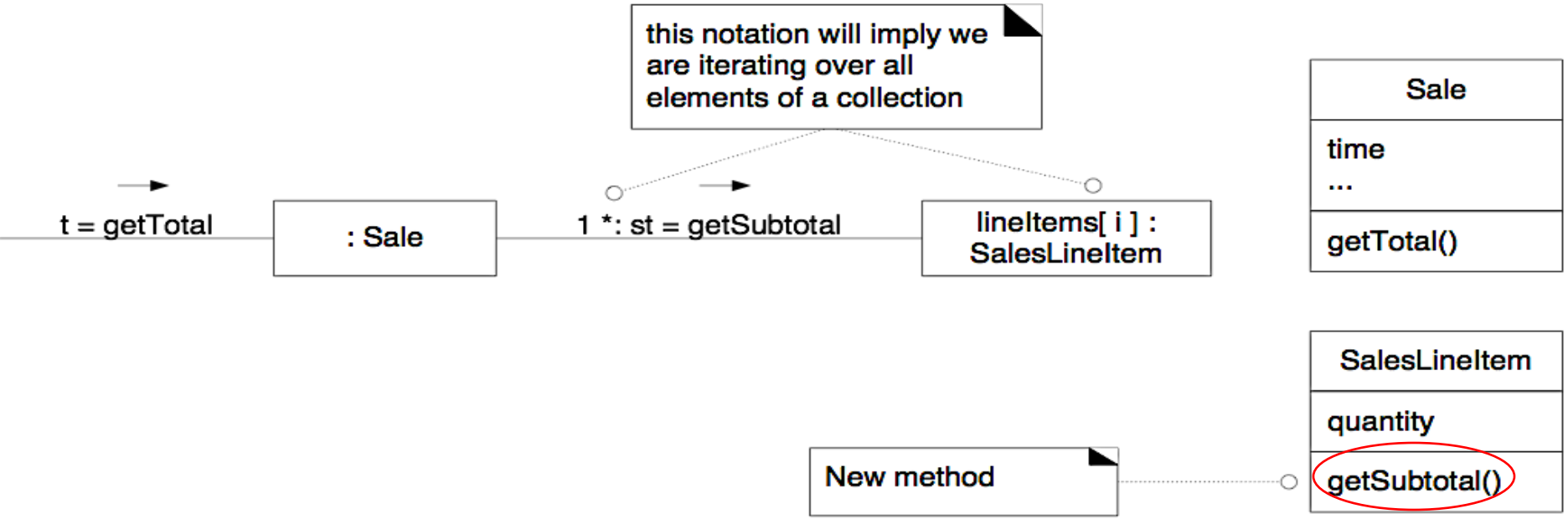


2. What information do we need to know to determine the LineItem subtotal?

2. What information do we need to know to determine the LineItem subtotal?

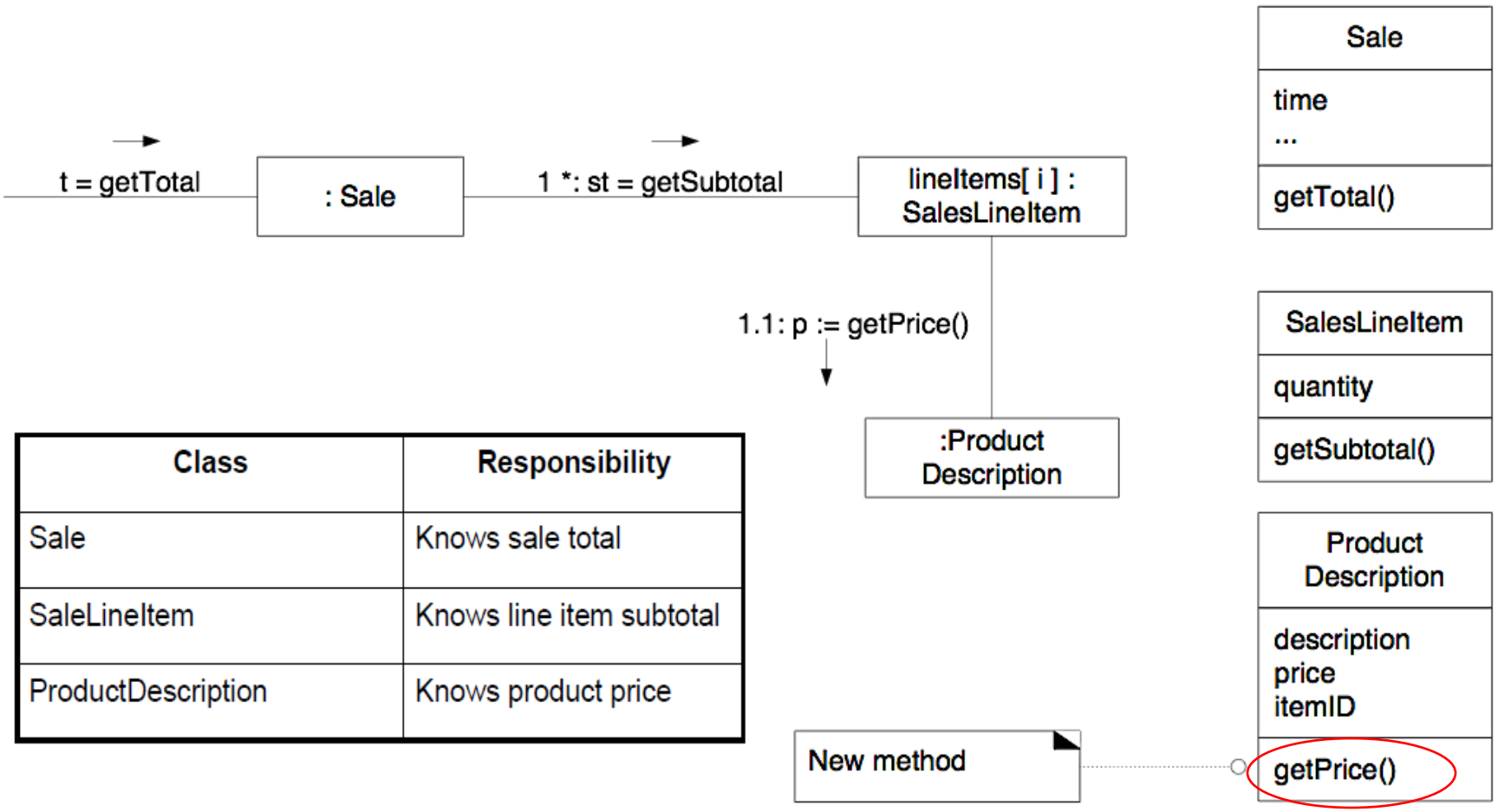


2. What information do we need to know to determine the LineItem subtotal?



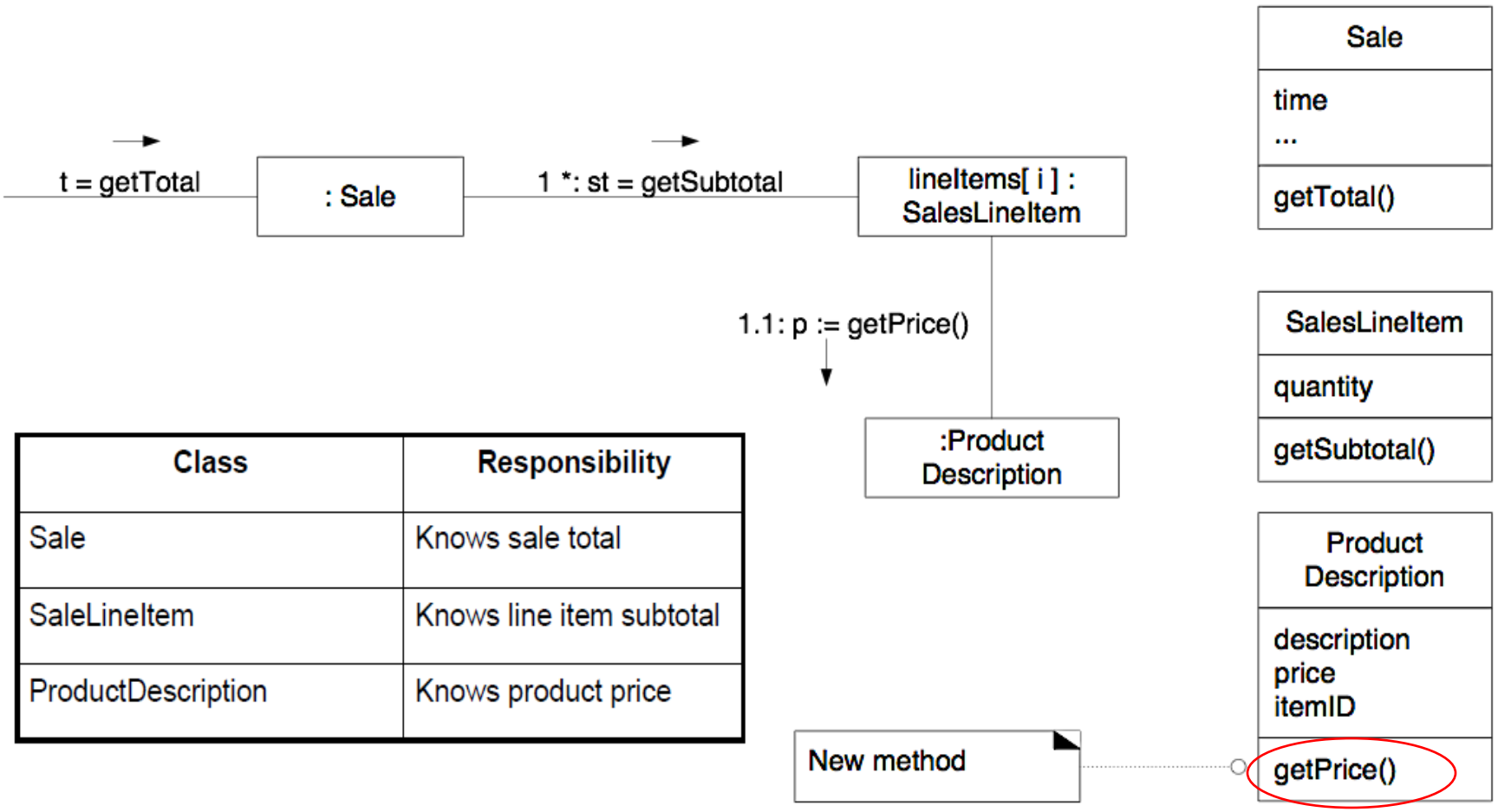
3. How does the SalesLineItem find out the product price?

3. How does the **SalesLineItem** find out the product price?



Class	Responsibility
Sale	Knows sale total
SaleLineItem	Knows line item subtotal
ProductDescription	Knows product price

"*Partial*" information experts **collaborate** to fulfill the responsibility.



Class	Responsibility
Sale	Knows sale total
SaleLineItem	Knows line item subtotal
ProductDescription	Knows product price

2. Creator

Problem: Who creates object A?

Solution: Assign class B the responsibility to create object A if one of these is true (more is better)

- B contains or compositely aggregates A
- B records A
- B closely uses A
- B has the initializing data for A

2. Creator

Problem: Who creates object A?

Solution: Assign class B the responsibility to create object A if one of these is true (more is better)

```
1  public class Customer : Entity, IAggregateRoot
2  {
3      private readonly List<Order> _orders;
4
5      public void AddOrder(List<OrderProduct> orderProducts)
6      {
7          var order = new Order(orderProducts); // Creator
8
9          if (this._orders.Count(x => x.IsOrderedToday()) >= 2)
10         {
11             throw new BusinessRuleValidationException("You cannot order
12         }
13
14         this._orders.Add(order);
15
16         this.AddDomainEvent(new OrderAddedEvent(order));
17     }
18 }
```

3. Controller

Problem: What first object *beyond the UI layer* receives and coordinates ("controls") of a system operation?
Solution: Assign the responsibility to an object representing one of these choices

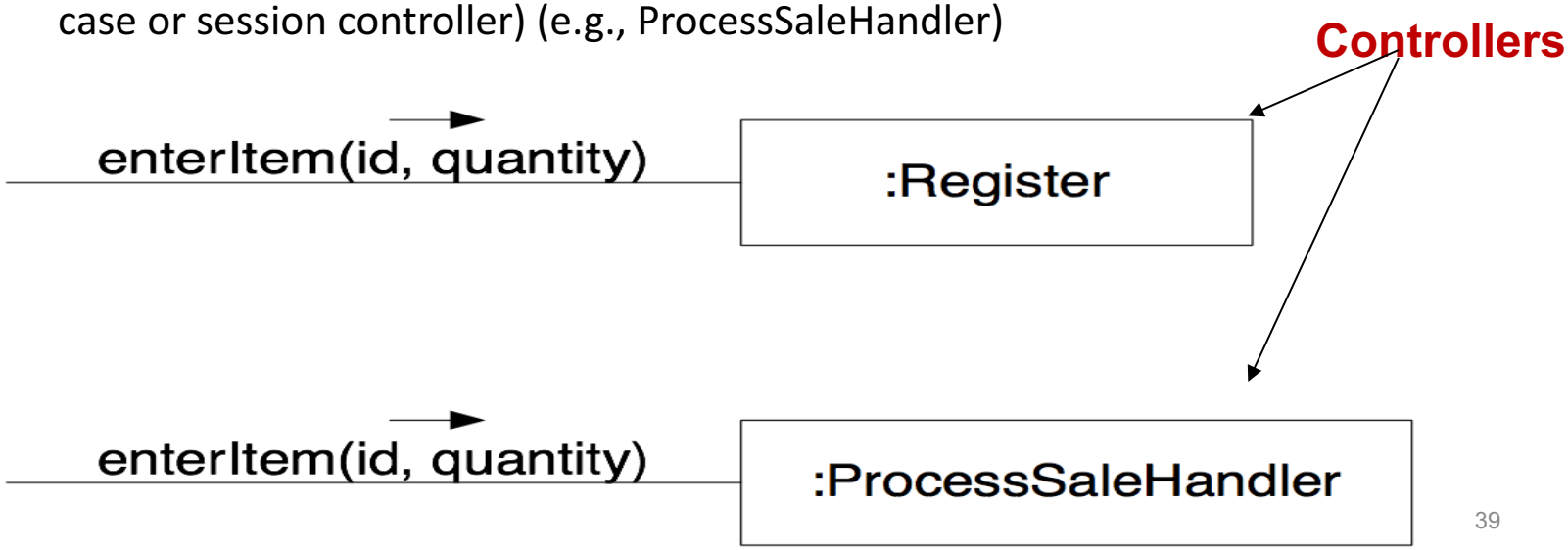
1. Represents a) the overall "system", "root object" (e.g., MonopolyGame), b) the device that the software is running within (e.g., BankCashMachine), or c) a major subsystem (e.g., Register) (these are all variations of a *facade* controller)
2. Represents a use case scenario within which the system operation occurs (a use case or session controller) (e.g., ProcessSaleHandler)

- Controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.
  - It delegates the work to other class and coordinates the overall activity.

3. Controller

Problem: What first object *beyond the UI layer* receives and coordinates ("controls") of a system operation?
Solution: Assign the responsibility to an object representing one of these choices

1. Represents a) the overall "system", "root object" (e.g., MonopolyGame), b) the device that the software is running within (e.g., BankCashMachine), or c) a major subsystem (e.g., Register) (these are all variations of a *facade* controller)
2. Represents a use case scenario within which the system operation occurs (a use case or session controller) (e.g., ProcessSaleHandler)

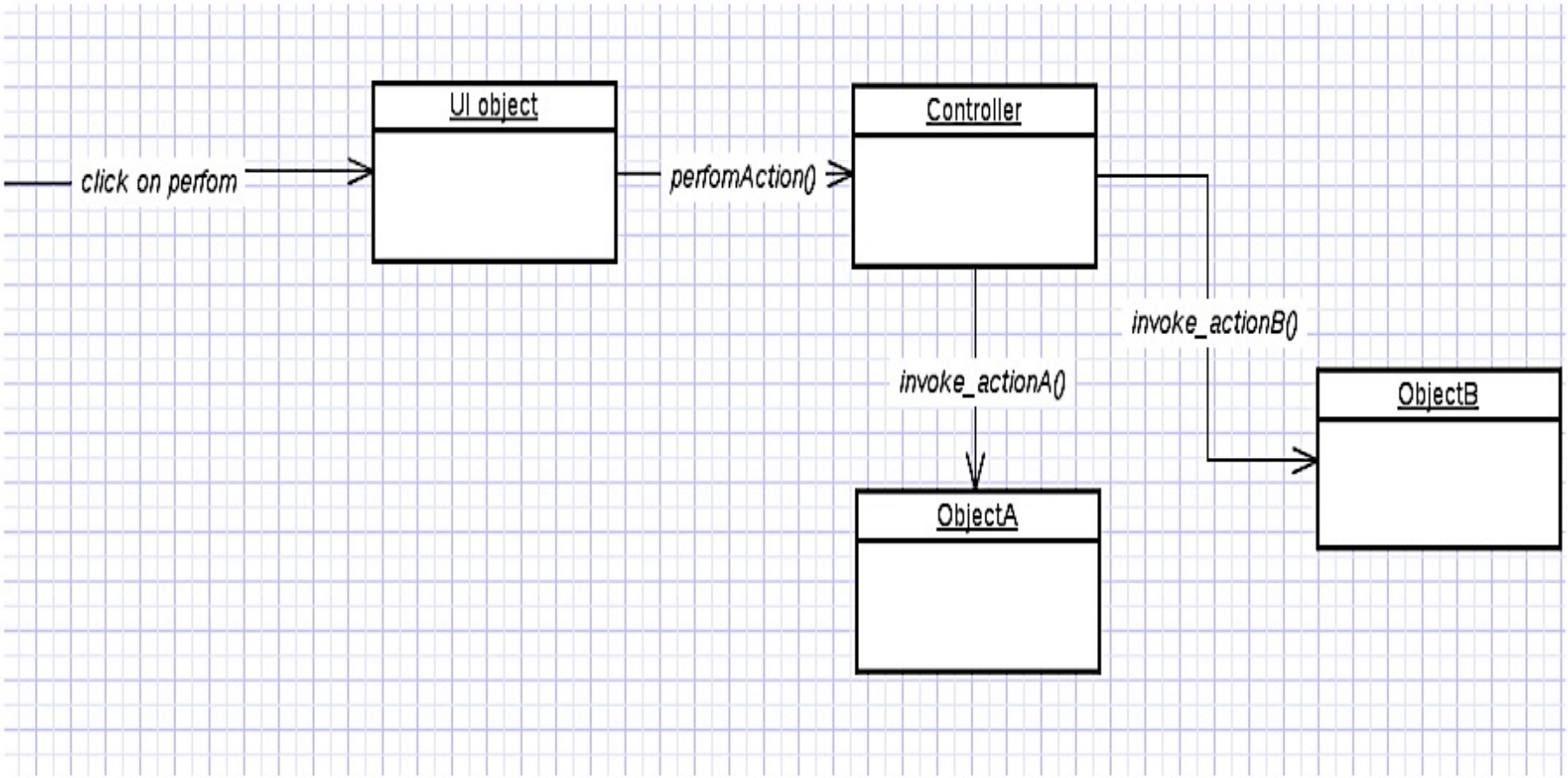


3. Controller

Problem: What first object *beyond the UI layer* receives and coordinates ("controls") of a system operation?
Solution: Assign the responsibility to an object representing one of these choices

1. Represents a) the overall "system", "root object" (e.g., MonopolyGame), b) the device that the software is running within (e.g., BankCashMachine), or c) a major subsystem (e.g., Register) (these are all variations of a *facade* controller)
2. Represents a use case scenario within which the system operation occurs (a use case or session controller) (e.g., ProcessSaleHandler)

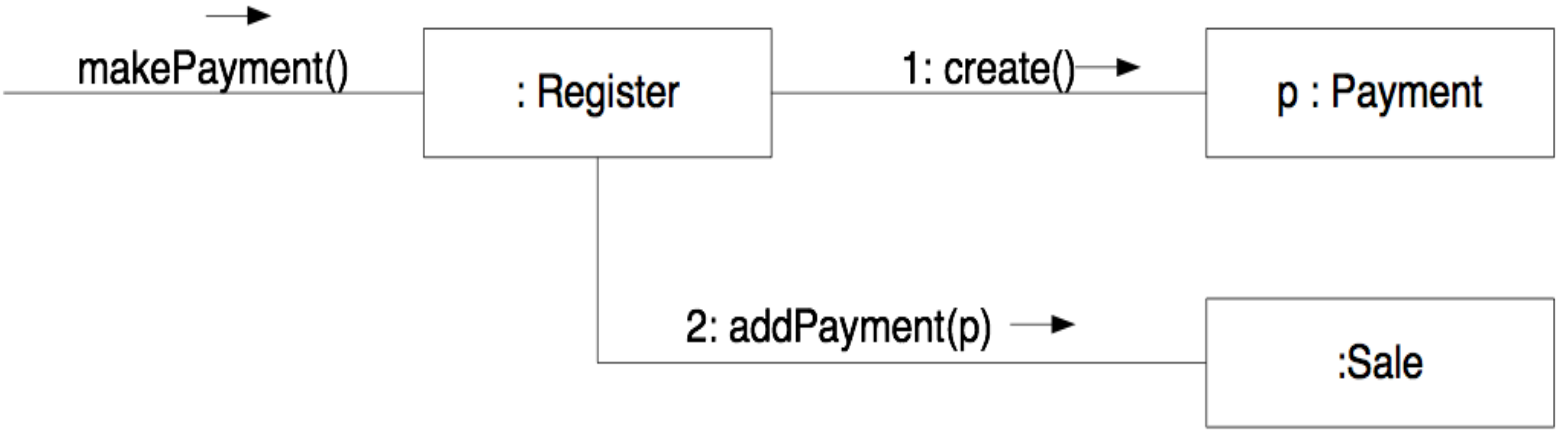
- Controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.
  - It delegates the work to other class and coordinates the overall activity.


UI object
Controller
click on perfom
perfomAction()
invoke_actionB()
invoke_actionA()
ObjectB
ObjectA

4. Low Coupling

Problem: How to reduce the impact of change? How to support low dependency and increased reuse?

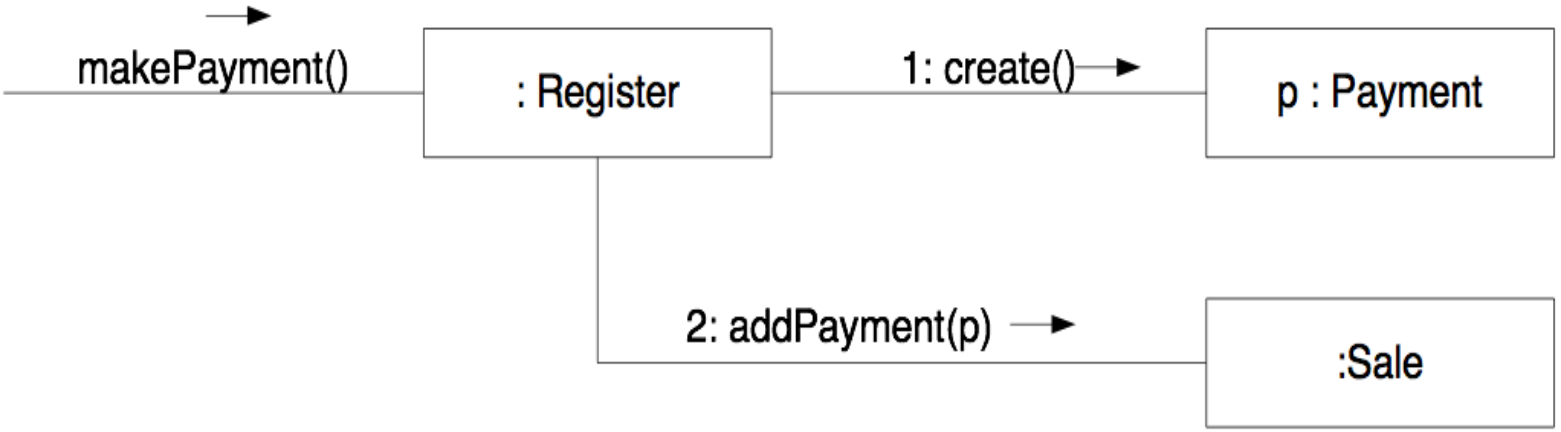
Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.



4. Low Coupling

Problem: How to reduce the impact of change? How to support low dependency and increased reuse?

Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

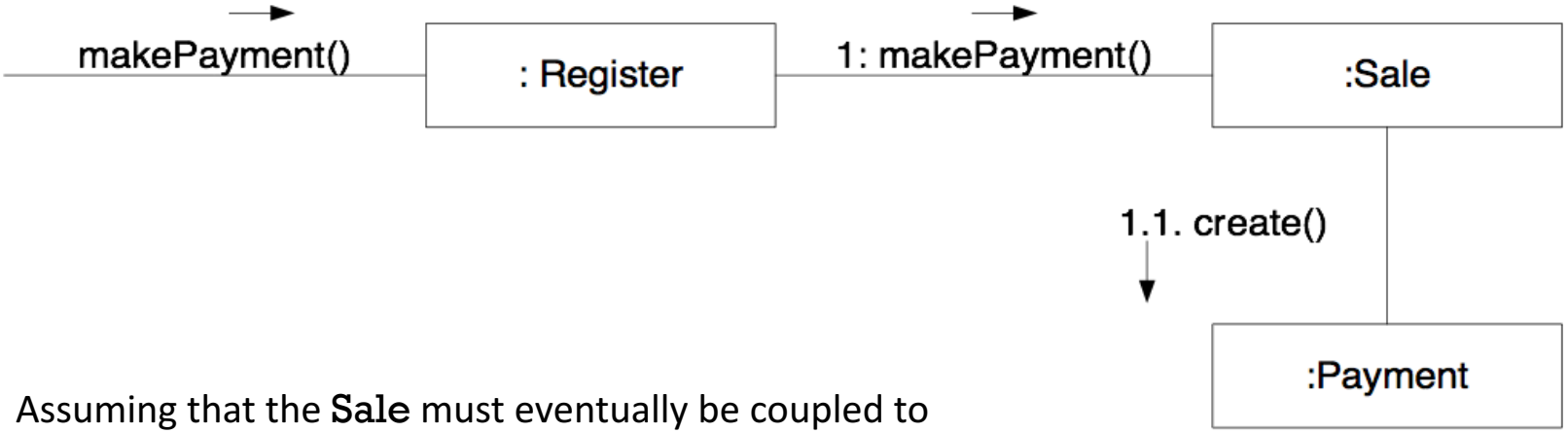


Register is coupled to both Sale and Payment.

4. Low Coupling

Problem: How to reduce the impact of change? How to support low dependency and increased reuse?

Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

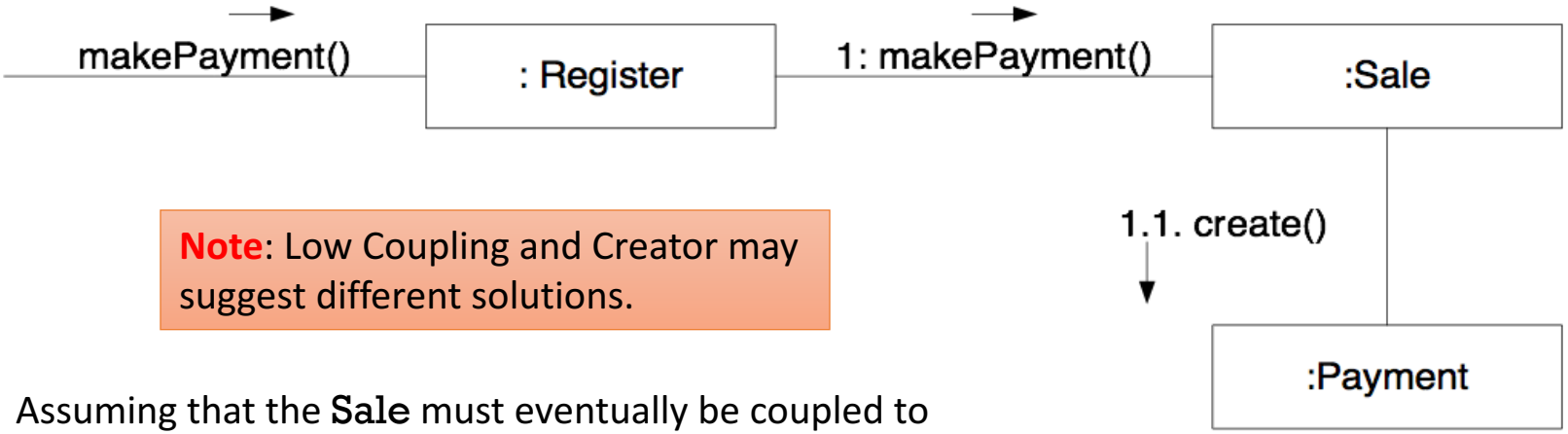


Assuming that the **Sale** must eventually be coupled to knowledge of a **Payment**, having Sale create the **Payment** does not increase coupling.

4. Low Coupling

Problem: How to reduce the impact of change? How to support low dependency and increased reuse?

Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

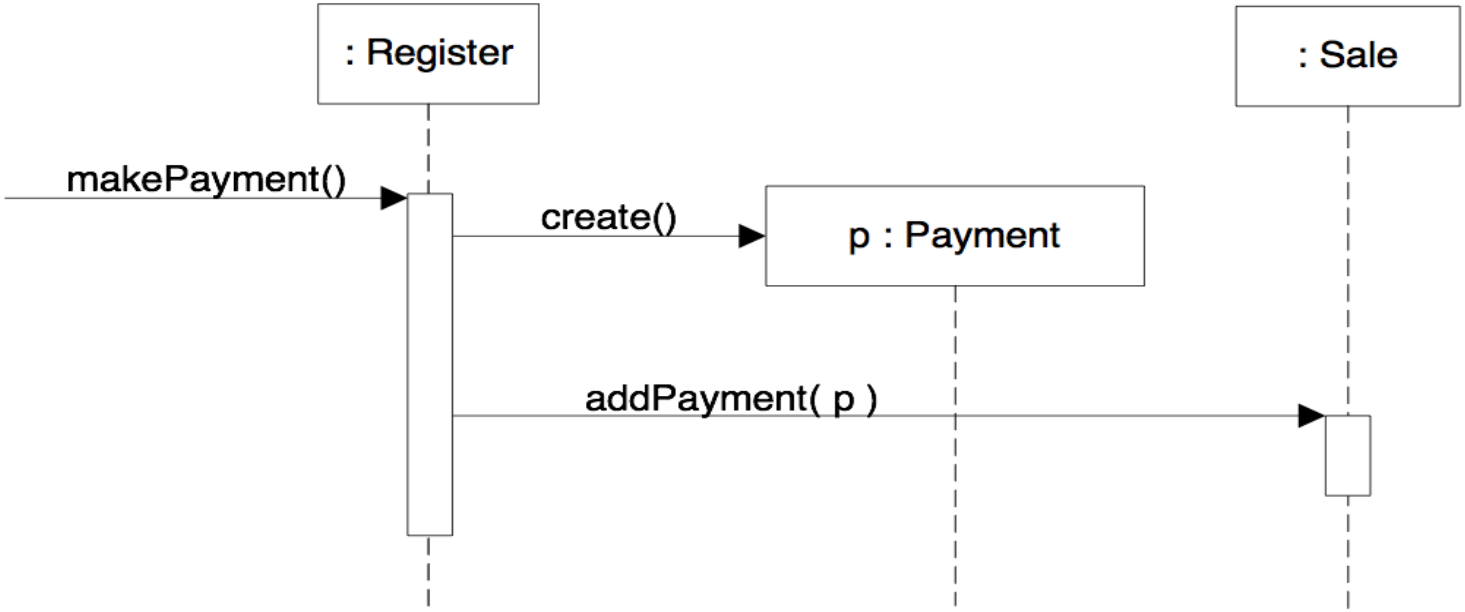


Note: Low Coupling and Creator may suggest different solutions.

Assuming that the **Sale** must eventually be coupled to knowledge of a **Payment**, having Sale create the **Payment** does not increase coupling.

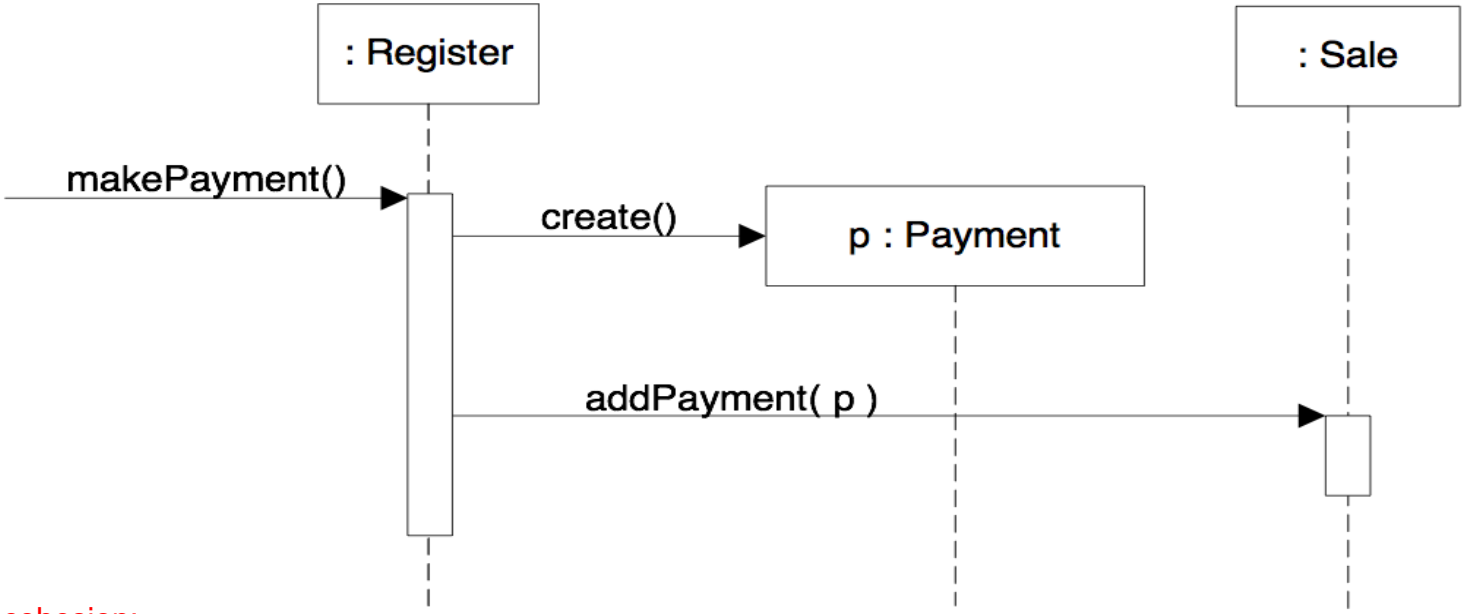
5. High Cohesion

Problem: How to keep objects focused, understandable, manageable and as a side effect support Low Coupling?
Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.



5. High Cohesion

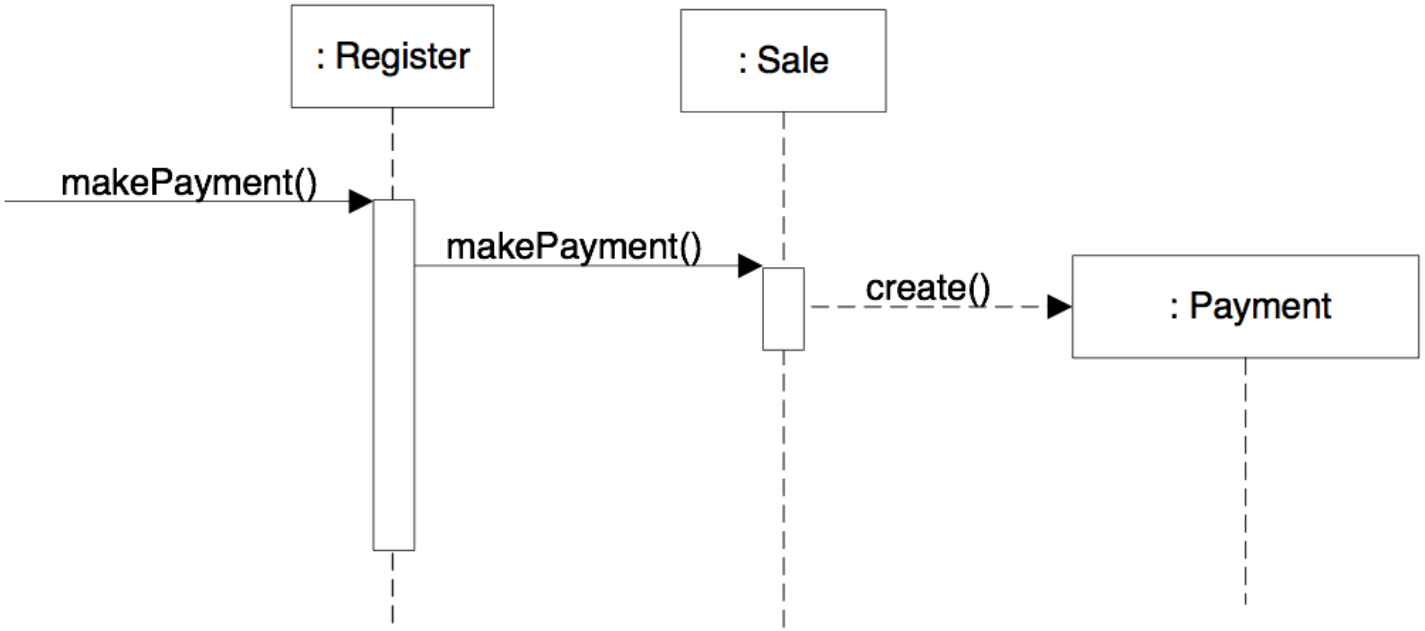
Problem: How to keep objects focused, understandable, manageable and as a side effect support Low Coupling?
Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.



Low cohesion:
Register is taking part of the responsibility for fulfilling “makePayment” operation and many other unrelated responsibility (50 system operations all received by Register). Then it will become burden with tasks and become incohesive.

5. High Cohesion

Problem: How to keep objects focused, understandable, manageable and as a side effect support Low Coupling?
Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.



Solution:
Delegate the payment creation responsibility to “Sale” to support high cohesion (and low coupling).

6. Indirection

Problem: Where to assign a responsibility to avoid direct coupling between two or more things?
Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

- Mediator Pattern

```
1  public class CustomerOrdersController : Controller
2  {
3      private readonly IOrdersService _ordersService;
4
5      public CustomerOrdersController(IOrdersService ordersService)
6      {
7          this._ordersService = ordersService;
8      }
9  }
```

```
public class CustomerOrdersController : Controller
{
    private readonly IOrdersService _ordersService;

    public CustomerOrdersController(IOrdersService ordersService)
    {
        this._ordersService = ordersService;
    }
```

```
public class CustomerOrdersController : Controller
{
    private readonly IMediator _mediator;

    public CustomerOrdersController(IMediator mediator)
    {
        this._mediator = mediator;
    }

    public async Task<IActionResult> AddCustomerOrder(
        [FromRoute]Guid customerId,
        [FromBody]CustomerOrderRequest request)
    {
       await _mediator.Send(new AddCustomerOrderCommand(customerId, req

       return Created(string.Empty, null);
    }
}
```

6. Indirection

Problem: Where to assign a responsibility to avoid direct coupling between two or more things?
Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

- Mediator Pattern

✓ Indirection => low coupling

✘ Indirection => low readability and reasoning about the whole system!
  - You don't know which class handles the command from the Controller definition.

- There is a trade off to take into consideration!

7. Polymorphism

Problem: How handle alternatives based on type?
Solution: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior (using polymorphism operations) to the types for which the behavior varies.

7. Polymorphism

Problem: How handle alternatives based on type?
Solution: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior (using polymorphism operations) to the types for which the behavior varies.

```
1  public Customer(string email, string name, ICustomerUniquenessChecker c
2  {
3      this.Email = email;
4      this.Name = name;
5
6      var isUnique = customerUniquenessChecker.IsUnique(this); // doing -
7      if (!isUnique)
8      {
9          throw new BusinessRuleValidationException("Customer with this e
10     }
11
12     this.AddDomainEvent(new CustomerRegisteredEvent(this));
13 }
```

8. Pure Fabrication

Problem: What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling but solutions offered by other principles are not appropriate?
Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept.

```
public interface IForeignExchange
{
    List<ConversionRate> GetConversionRates();
}

public class ForeignExchange : IForeignExchange
{
    private readonly ICacheStore _cacheStore;

    public ForeignExchange(ICacheStore cacheStore)
    {
        _cacheStore = cacheStore;
    }

    public List<ConversionRate> GetConversionRates()
    {
        var ratesCache = this._cacheStore.Get(new ConversionRatesCacheKey());

        if (ratesCache != null)
        {
            return ratesCache.Rates;
        }

        List<ConversionRate> rates = GetConversionRatesFromExternalApi();

        this._cacheStore.Add(new ConversionRatesCache(rates), new ConversionRatesCac

        return rates;
    }

    private static List<ConversionRate> GetConversionRatesFromExternalApi()
    {
        // Communication with external API. Here is only mock.

        var conversionRates = new List<ConversionRate>();

        conversionRates.Add(new ConversionRate("USD", "EUR", (decimal)0.88));
        conversionRates.Add(new ConversionRate("EUR", "USD", (decimal)1.13));

        return conversionRates;
    }
}
```

9. Protected Variations

Problem: How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution: Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

- One of the most important software metrics is the **ease of change.**
  - SOLID principles, especially the Open-Close principle (but all of them supports change)
  - Gang of Four (GoF) Design Patterns
  - Encapsulation
  - Law of Demeter
  - asynchronous messaging, Event-driven architectures
  - ….

9. Protected Variations

Problem: How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution: Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

- One of the most important software metrics is the **ease of change.**
  - SOLID principles, especially the Open-Close principle (but all of them supports change)
  - Gang of Four (GoF) Design Patterns
  - Encapsulation
  - Law of Demeter
  - asynchronous messaging, Event-driven architectures
  - ….

More on patterns!

- GoF - Design Patterns: Elements of Reusable Object-Oriented Software
- “Analysis Patterns”, Martin Fowler
  - patterns in domain models of businesses
- “Patterns of Enterprise Application Architecture”, Martin Fowler et al.
- “Reactive Design Patterns”, Roland Kuhn et al.
  - building message-driven distributed systems that are resilient, responsive, and elastic
- ……..

break

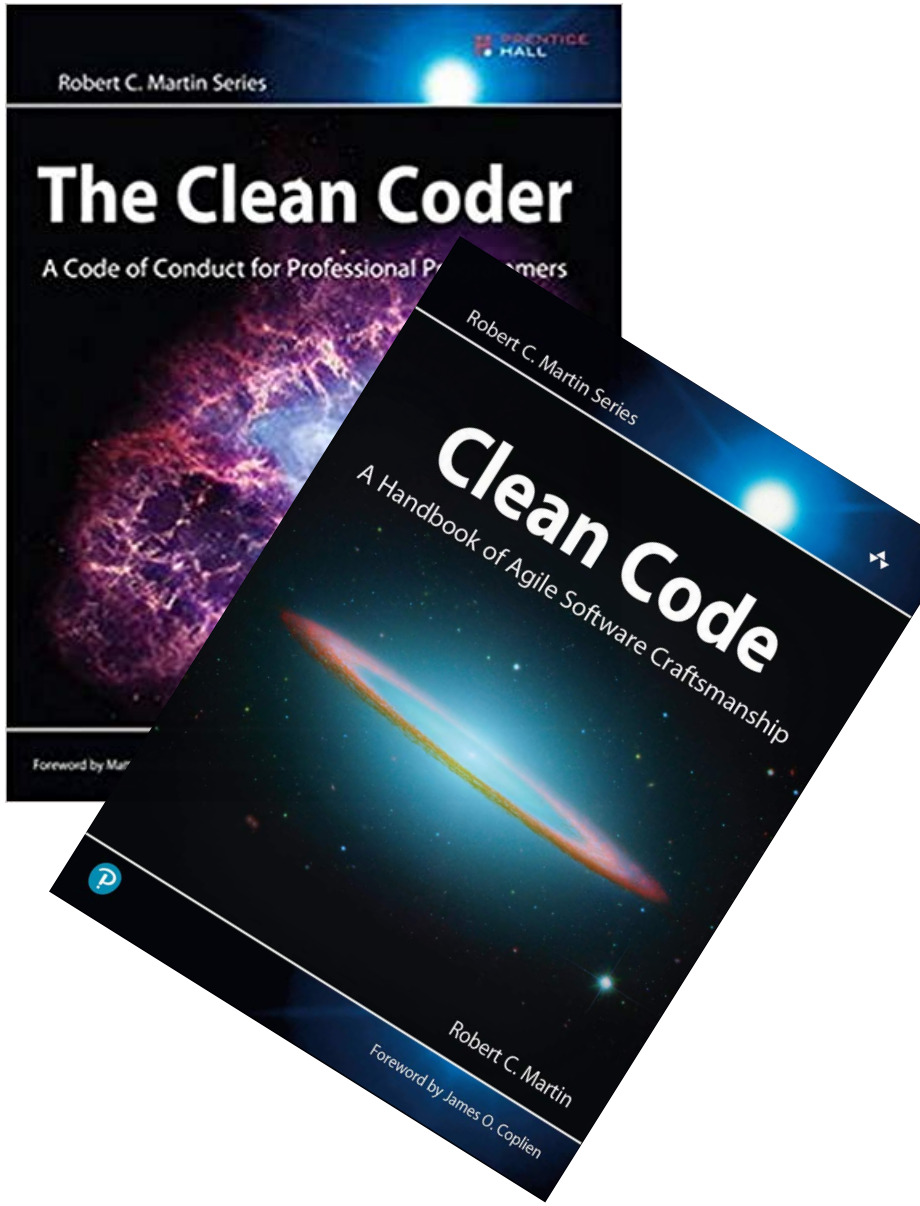
Robert C. Martin

"Uncle bob"

- One of the authors of the Agile Manifesto
- Also known for "SOLID principles."

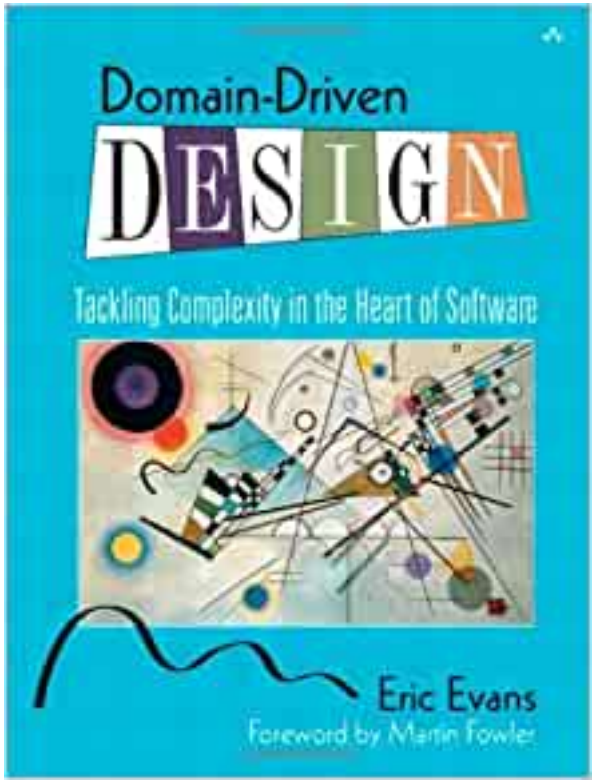
"Even bad code can function. But if code isn't clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code. But it doesn't have to be that way."

--R. Martin



Domain-Driven Design: Tackling Complexity in the Heart of Software

- how you can make the design of your software match your mental model of the problem domain you are addressing.
- about how you think of your domain, the language you use to talk about it, and how you organize your software to reflect your improving understanding of it.



"I see this book as essential reading for software developers—it is a future classic."

—**Ralph Johnson**, author of *Design Patterns*