MSO
Object-oriented design 2

Hans Philippi
(based on the course slides of Wouter Swierstra)

September 3, 2018

Recap & topics

- Last lecture: design principles, cohesion & coupling
- Today: GRASP principles for assigning *responsibilities*

Back to our case

1. Customer arrives at checkout with goods to purchase
2. Cashier starts a new sale
3. Cashier records item description, price, and running total for each item
4. System computes sales total, including tax
5. Cashier informs Customer of the total and processes payment
6. System logs completed sale, sends payment info to Accounting and Inventory departments
7. System prints a receipt
8. Happy Customer leaves with receipt and goods

Object oriented design

Responsibilities are assigned to classes during design:

- Does the system compute the Sales total? Or is it part of the Sale?
- Who is responsible for pricing information?
- How does information flow to the Accounting and Inventory departments?

Taking a step back...

Martin Fowler identifies three separate perspectives on the software development process:

- Conceptual
- Specification
- Implementation

Conceptual perspective

This perspective "represents the concepts in the domain under study. . . a conceptual model should be drawn with little or no regard for the software that might implement it"

Specification perspective

"Now we are looking at software, but we are looking at the interfaces of the software, not the implementation"

Implementation perspective

"This is probably the most often-used perspective, but in many ways the specification perspective is often a better one to take"

Object-oriented design

Shalloway and Trott argue that these three levels are useful to understand objects:

- Conceptual – an object is a set of responsibilities
- Specification – an object is a set of methods
- Implementation – an object is code and data

The object-oriented design thinks in terms of objects and responsibilities first

Example: shape database

Our functional design yielded the following steps:

1. Locate the list of shapes in the database
2. Open up the list of shapes
3. Sort this list of shapes in some manner
4. Display every shape in the list individually

Example: object-oriented design

- ShapeDB class
  - `getCollection` method returns the set of shapes
- Abstract Shape class
  - `display` method
- Subclasses of Shape such as Square or Circle
- Collection class
  - `display_collection` method, displays all Shapes in the Collection
  - `sort` method, sorts the shapes in some order

Example execution

1. Main program creates a ShapeDB object
2. Main program asks the ShapeDB object for the collection of Shapes
3. Main program asks the Collection to sort the Shapes
4. Main program asks the Collection to display the Shapes
5. The Collection asks each Shape to draw itself

Robust to change?

- What happens if we add a new Shape?
- What happens if we want to sort differently?

Robust to change?

- What happens if we add a new Shape?
  - We can define a new subclass of the Shape, with its own `display` method
- What happens if we want to sort differently?
  - We change the sorting method in the Collection class.

Both these changes are local to the objects responsible

Encapsulation

Why did this approach work?

Encapsulation is any form of information hiding:

- The private methods of an object are hidden to the outside world
- An abstract class ensures the implementation of its methods are 'hidden'
- Encapsulation enables classes to keep responsibilities exclusive: *stay away from my responsibilities*

Writing designs

- So we have a better understanding of object oriented design. . .
- . . . and we understand how to write use cases and requirements. . .
- . . . and we know what a domain model is. . .
- . . . and we are familiar with concepts like cohesion and coupling. . .
- .. but how do we write a design?

Writing designs

- Coming up with a good design is hard – it requires creativity, technical insight, and an excellent understanding of the problem domain
- During analysis, you gather information about the problem
- During design, you come up with a solution
- There is no single *right* or *wrong* answer
- There is no *one weird trick* I can teach you that will always yield a good design

Responsibility

One key question to ask during design is:
Which class is responsible?

- Which class is responsible for computing the sales total?
- Which class is responsible for logging sales? Or notifying the Accounting department?
- Which class is responsible for drawing a shape? Or sorting the shape collection?

Responsibility and methods

- Responsibility is an abstract concept
  - Being responsible for database storage involves a lot of classes and methods
  - Being responsible for the VAT calculation is usually much simpler
- A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities

GRASP principles

Larman identifies several principles [1] that help *guide* good design: the GRASP principles (General Responsibility Assignment Software Principles)

The GRASP principles help assign responsibility to the classes in your domain model

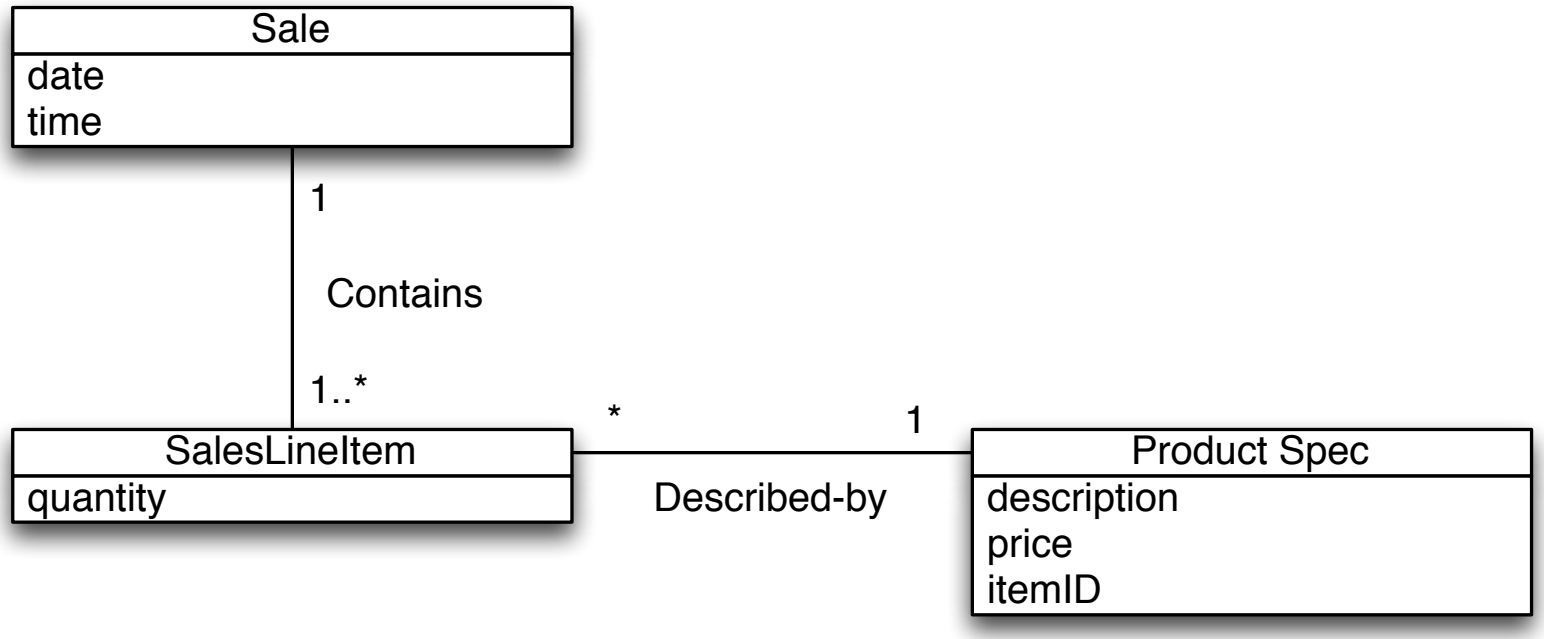
[1] Sometimes these are called GRASP patterns, but I will try to only use the term *pattern* for the *design patterns* we will see in Shalloway and Trott's book

GRASP principles

The GRASP principles help assign responsibility to the classes in your domain model:

- Information Expert *(who has the info?)*
- Creator *(who should create instances?)*
- Loose coupling *(general quality guideline)*
- Strong cohesion *(general quality guideline)*
- Controller *(use case: which subsystem handles the event?)*

Information Expert – example



Example: Who calculates the sales tax?

Information Expert – example

- Every ProductSpecification should manage its own price
- Every SalesLineItem records the price of selling these items
- Every Sale should compute its own total price and tax

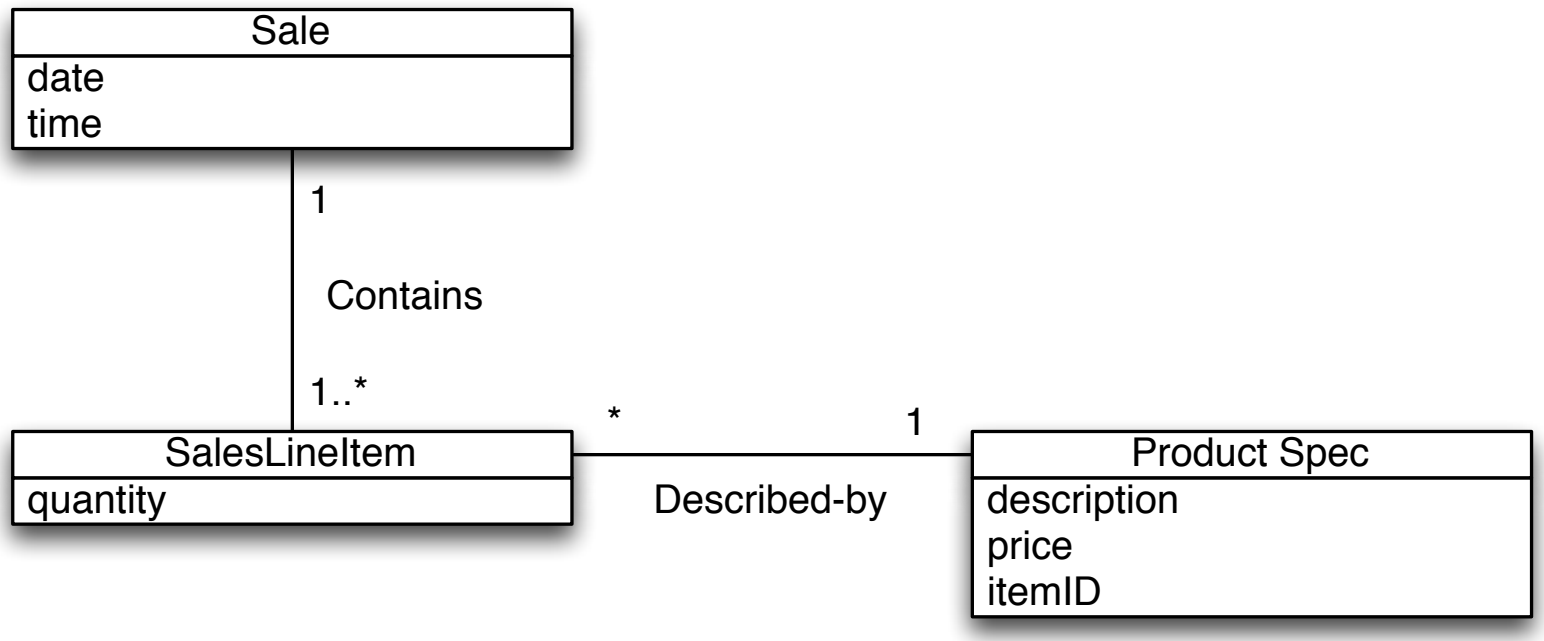
We can start to design the UML class diagrams recording the operations that these classes should support

Information Expert (GRASP principle)

1. What information is necessary to perform some computation?
2. Which class or classes have this information already?

Assign responsibility to those classes that have the required information readily available

Creator – example



Example: Who adds SalesLineItems to the current sale?

Creator – example

- A SalesLineItem is a part of a Sale
- A Sale should be responsible for adding new SalesLineItems
- And hence, the Sale class should support a makeLineItem method

Creator (GRASP principle)

Idea: Assign class B the responsibility to create instances of a class A if one or more of the following is true:

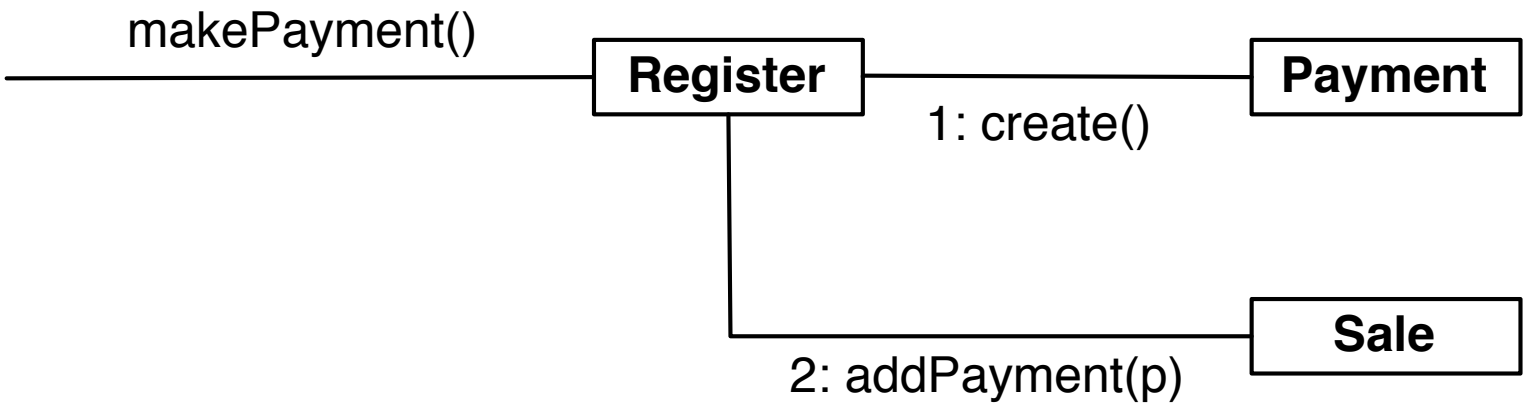
- B *aggregates* A objects
- B *contains* A objects
- B *records* instances of A objects
- B *closely uses* A objects
- B *has the initialization data* necessary to create A objects

(Note the similarity in the last case with the Information Expert pattern)

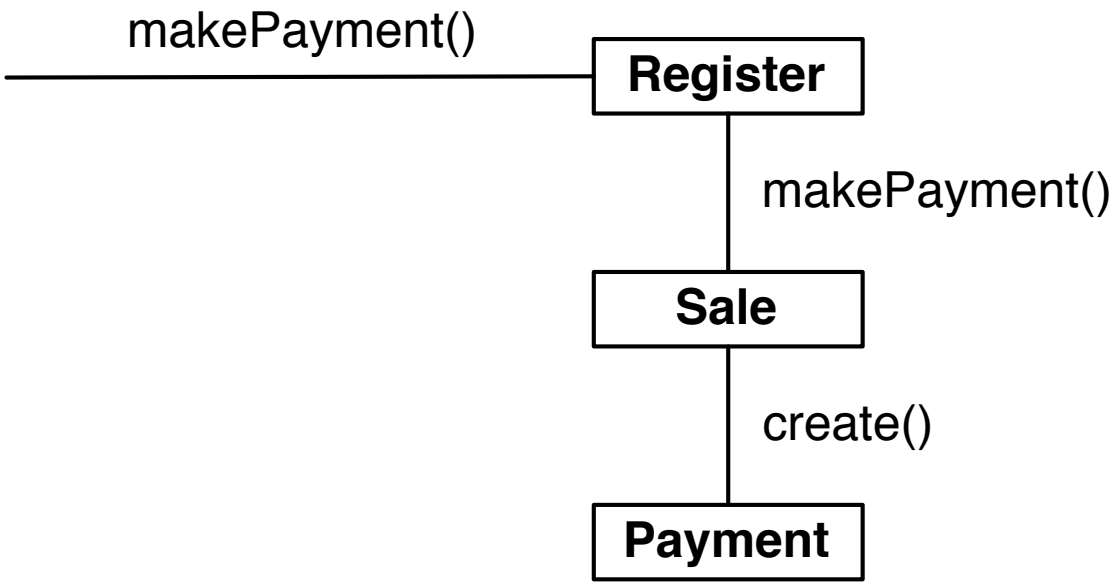
Loose coupling – example

A Register should support a makePayment() method that associates a new Payment with the current Sale

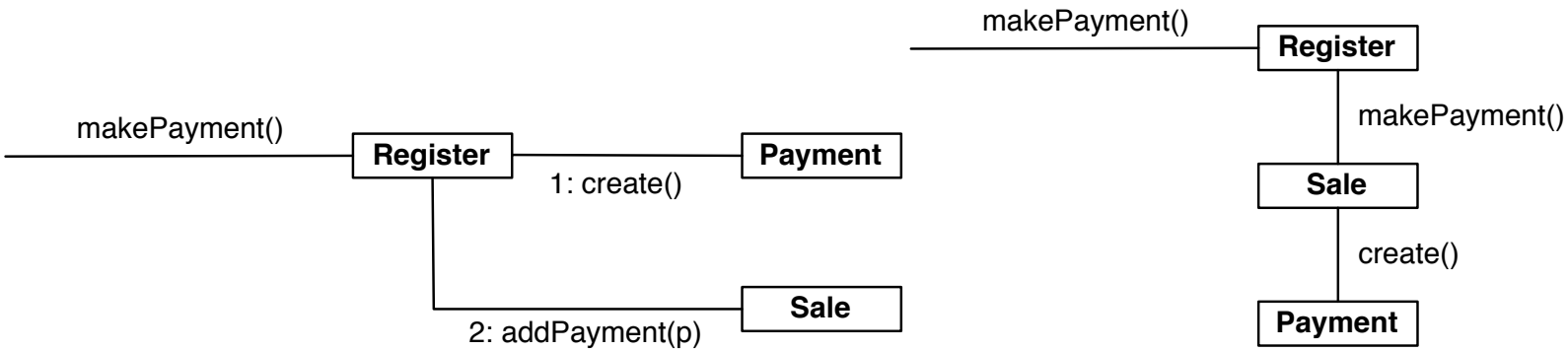
Loose coupling – example



Loose coupling – example

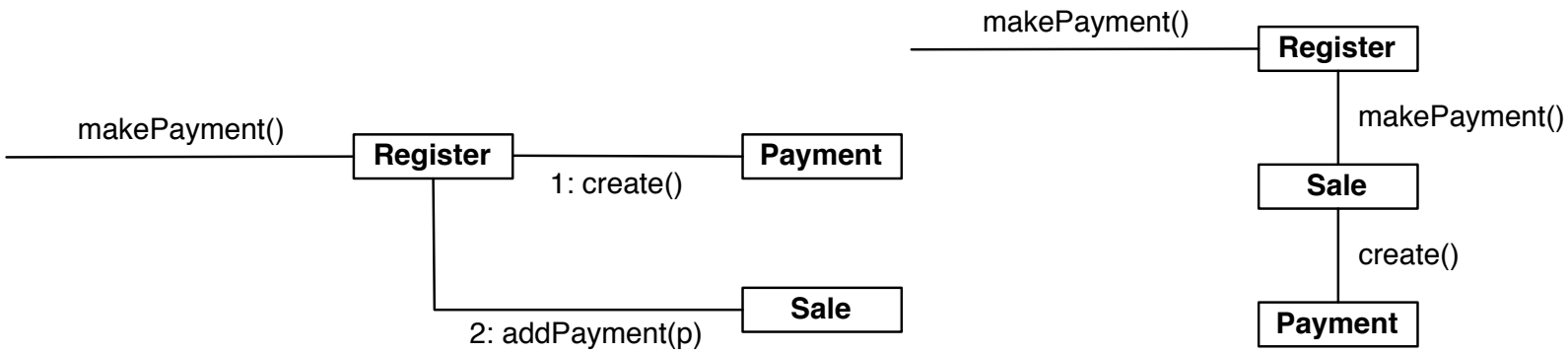


Loose coupling – example



Apply *loose coupling* and choose!

Loose coupling – example



- In the second diagram, the coupling is looser: the Register does not need to know about Payments
- So the Sale should be responsible for creating the associated Payment
- Try to assign responsibilities in such a way that coupling remains loose

This is a rule of thumb – that can be broken when necessary

Loose coupling (GRASP principle)

Idea: Favour the design with loose(st) coupling

Tight coupling may introduce problems:

- Propagation of code updates in related classes
- Harder to understand in isolation
- Harder to reuse

This principle is often combined with others, such as the Creator or Information Expert principles

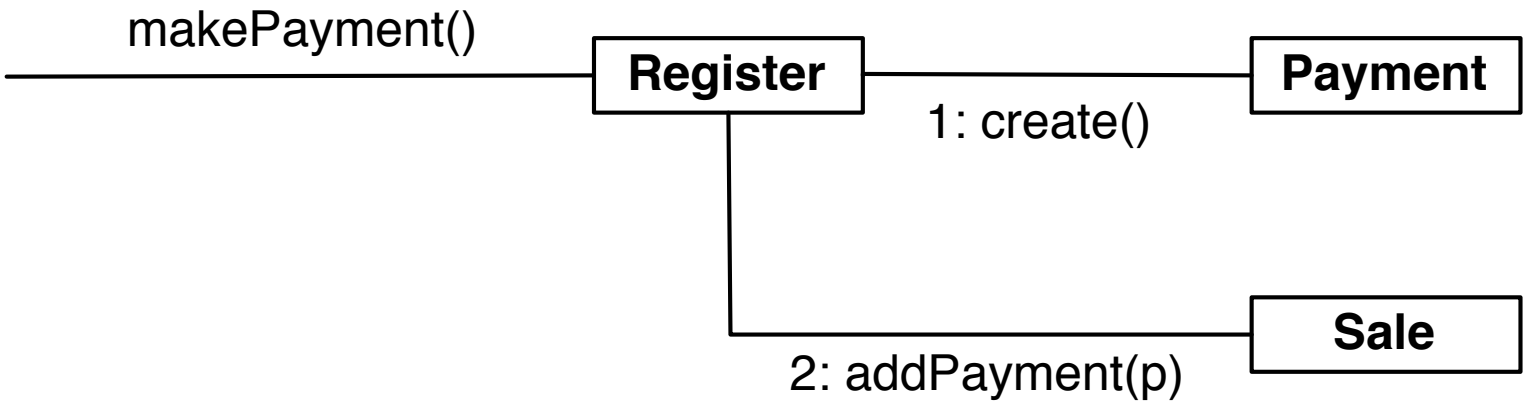
Strong cohesion

Idea: Favour the design with strong(er) cohesion

Weak cohesion may lead to code that is:

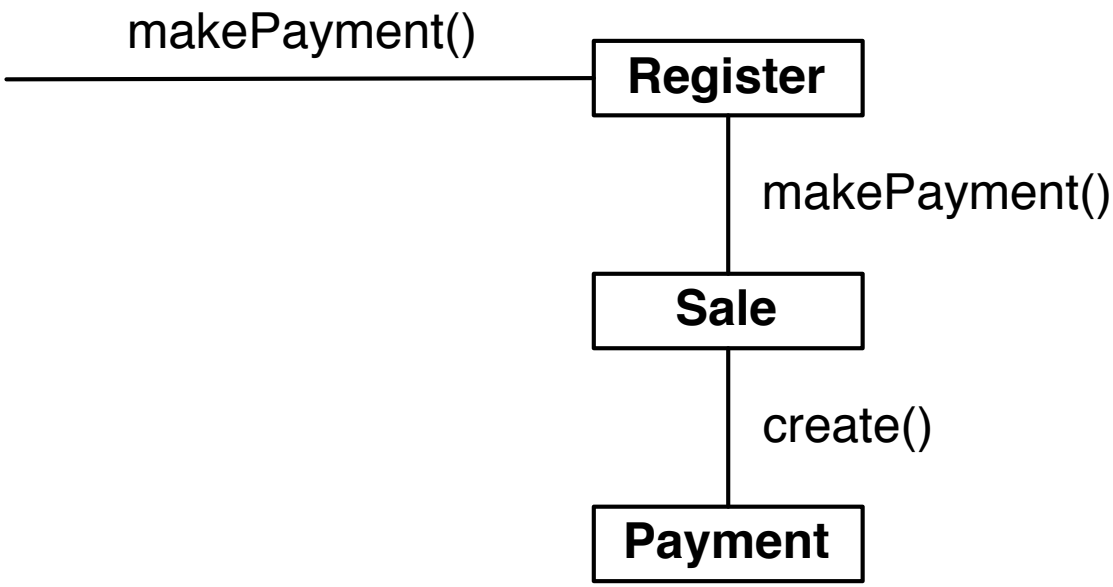
- hard to comprehend
- hard to reuse
- hard to maintain
- hard to test

Strong cohesion – example



- What if the Register class needs to do lots of other things besides creating payments?
- Remember that the prime responsabilities of Register concern the interaction with the customer
- So do not bother it with details concerning payments

Strong cohesion – example



In this design, the Register does not know anything about payments

Great!

Strong cohesion

Like the Loose coupling principle, assigning responsibilities to strengthen cohesion is a rule of thumb . . .

. . . but break it with caution

Controller

Idea: Assign the responsibility for receiving or handling system events to a class

- that represents the overall system, device, or subsystem
- that represents a specific use case within which the system event occurs. Such a class is often called the Handler or Controller.

Motivation: Many use cases or scenarios mention that "The system does X" – but what class should represent "the system"? The Controller principle suggests designing separate classes to handle individual use cases.

Applying GRASP principles

You can use the data you have gathered (domain model, use cases, requirements documents, interaction diagrams, etc.) to *design* the software system.

One way to do this is through *use case realizations* – which involves defining the classes in the software system and how they collaborate to execute a particular use case.

Here the GRASP principles can justify your design choices.

Case study: NextGen POS system

Consider the following use case (fragment):

1. Customer arrives
2. Cashier tells the system to create a new Sale
3. Cashier enters item identifier and quantity
4. System records item identifier and quantity
5. Cashier repeats steps 3 and 4 until all items have been processed
6. System presents total with all taxes calculated

Case study: NextGen POS system

Consider the following use case (fragment):

1. Customer arrives
2. Cashier tells the system to create a new Sale
3. Cashier enters item identifier and quantity
4. System records item identifier and quantity
5. Cashier repeats steps 3 and 4 until all items have been processed
6. System presents total with all taxes calculated

Question: Who is responsible for creating the new Sale object? What other data should be initialized once the Sale is created?

Case study: makeNewSale()

Once the Customer arrives, the Cashier should initiate a new Sale in the system.

Applying the Controller principle, we should define a Register class, that handles such requests (or if we have lots of different functionality, a specific SaleHandler class).

This Register class should provide a makeNewSale() method.

From our domain model, we have learned that a Sale also has a list of SalesLineItems. This list should also be initialized.

The Creator pattern suggests that this should be the responsibility of the Sale.

Case study: NextGen POS system

Consider the following use case (fragment):

1. Customer arrives
2. Cashier tells the system to create a new Sale
3. Cashier enters item identifier and quantity
4. System records item identifier and quantity
5. Cashier repeats steps 3 and 4 until all items have been processed
6. System presents total with all taxes calculated

Question: Who is responsible for computing the total price (including taxes)?

Case study: getTotal()

Calculating the price requires knowledge about all the items and quantities involved. This is information that the Sale has – we can now apply the Information Expert principle – the Sale class should support a getTotal() method.

Case study: TaxCalculator()

Or alternatively, perhaps you are worried about different sales tax calculations (in different countries or states, for instance).

Perhaps it would be better to have a specific TaxCalculator() Controller responsible for the calculation of sales tax.

Design is not about right and wrong – but balancing the different advantages and disadvantages.

Use case realizations

These examples show how to use the information in a use case during the design process.

Read Chapter 17 of Larman's *Applying UML and Patterns* for a thorough analysis.

Design validation

You can use the same techniques to check whether a design can implement a use case correctly. Does every class have the required information to perform some computation?

For example, when computing the total sale price, how does the Sale object know the price of the individual items? Or the quantities?

Different forms of information

There are four different ways that a class A may have access to certain information in a class B:

- Attribute visibility – B is an attribute of A
- Parameter visibility – B is a parameter of a method of A
- Local visibility – B is a local object in a method of A
- Global visibility – B is globally visible to A (this is the least common form)

Design validation

When checking a design, run through use cases, asking yourself the question:

How is the necessary information visible?

What have we seen today

- We should think of objects abstractly as managing *responsibilities*
- We can use the GRASP principles to assign responsibilities to the conceptual classes in our domain model

Doing so is a step towards turning our domain model into a concrete design

Material covered

- Design Patterns explained: chapter 1
- Applying UML and patterns: chapter 16–19