UNIT 2-DESIGN PATTERNS

GRASP Designing Objects with responsibilities-Creator-Information Expert-Low Coupling-High Cohesion-Controller-Design Patterns-Creational-Factory method-Structural-Bridge-Adapter-Behavoural-Strategy-Observer-Applying GOF design patterns.

GRASP

GRASP Means **General Responsibility Assignment Software Patterns .**

- **GRASP** is a learning aid that helps to understand essential object design and apply design reasoning in a methodical, rational and explainable way.
- **GRASP** is used as a tool to help master the basics of OOD and understanding responsibility assignment in object design.

There are nine basic OO design principles in **GRASP.** They are,

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

CREATOR

Creation of objects is one of the most common activities in an object oriented system. Which class is responsible for creating objects is a fundamental property of relationship between objects of particular classes.

Problem

Who should be responsible for creating a new instance of some class?

Solution:

Assign class B the responsibility to create an instance of a class A if one of these is true,

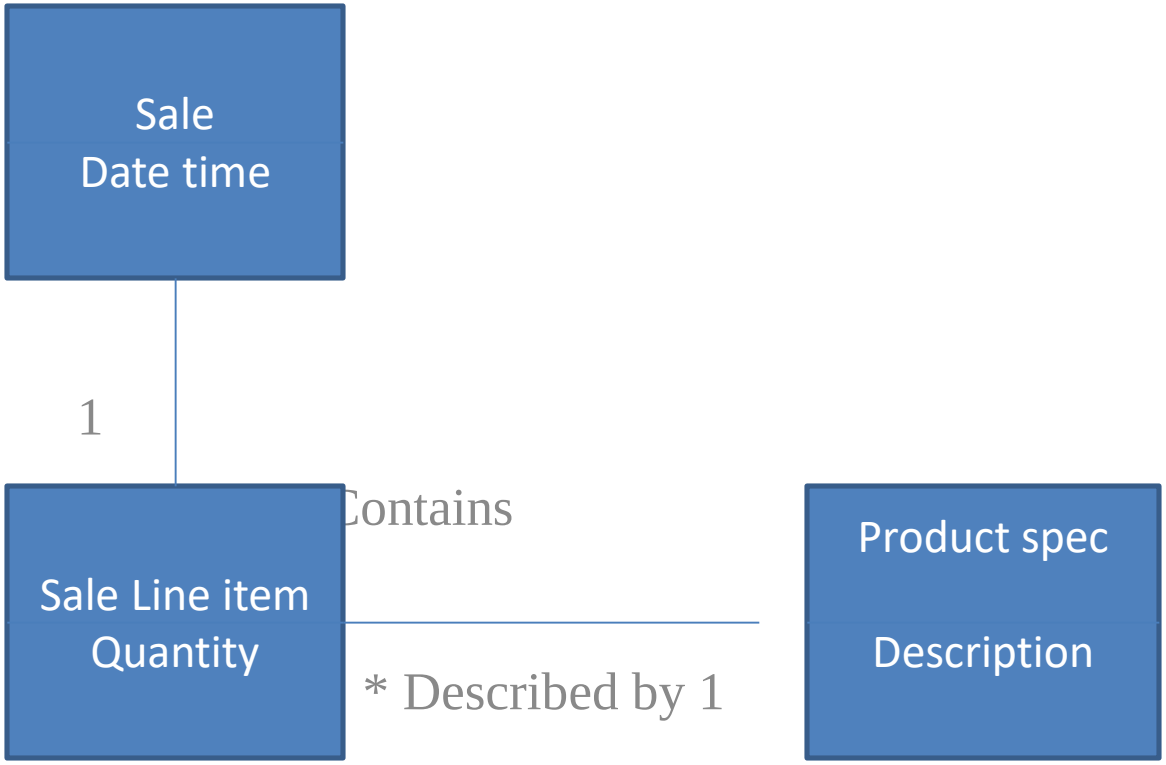
- ➢ B contains or compositely aggregates A
- ➢ B records A
- ➢ B closely uses A
- ➢ B has the initializing data for A that will be passed to A when it is created.

Thus B is an expert with respect to creating A

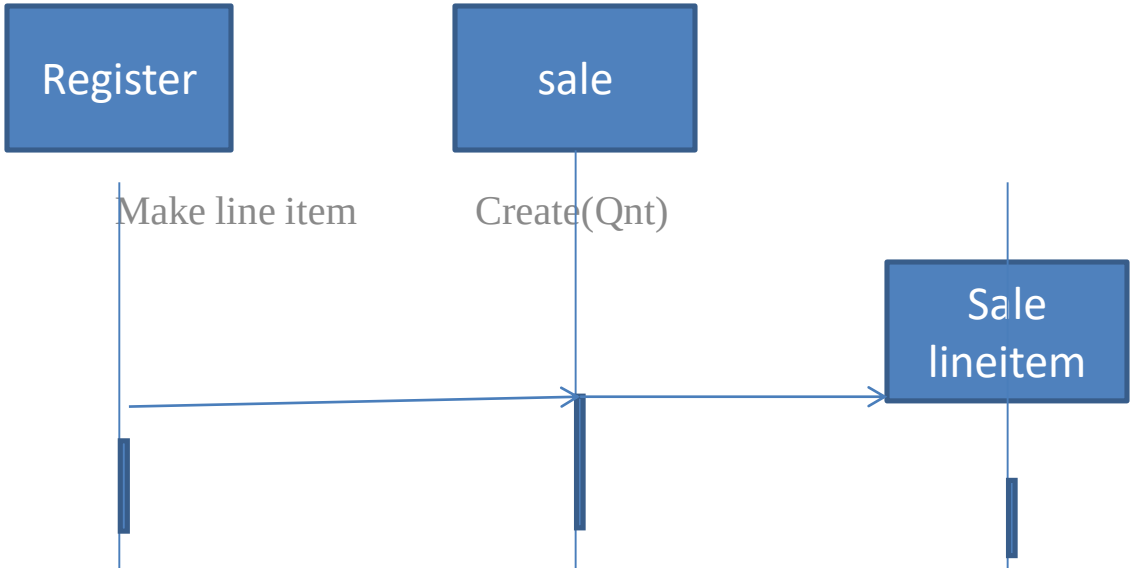
B is a creator of A objects

If more than one option applies, usually prefer a class B which aggregates or contains A.

Partial Domain Model



Creating a Sales line item



Since a Sale contains many Sales LineItem objects, the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.

This assignment of responsibilities requires that a makeLineItem method be defined in Sale. The method section of class diagram can then summarize the responsibility assignment results, concretely realized as methods.

INFORMATION EXPERT

Problem

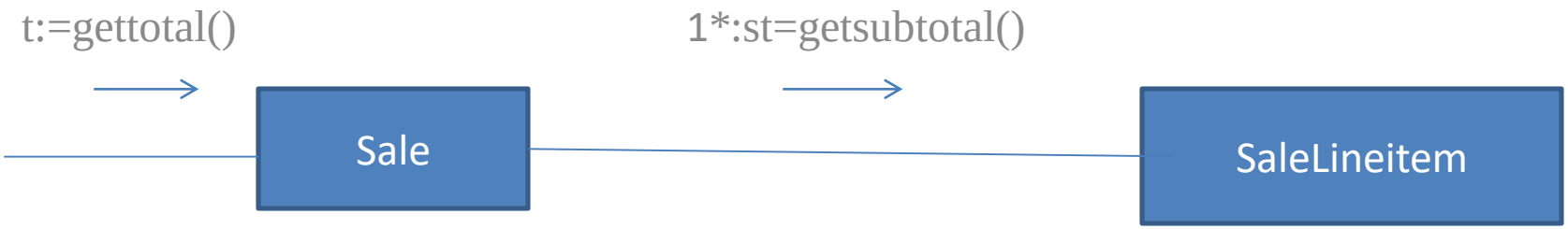
What is a general principle of assigning responsibilities to objects?

Solution:

Assign a responsibility to the information expert-the class that has the information

necessary to full fill the responsibility.

- What information is needed to determine the grand total? A Sale instance contains these; therefore, by the guideline of Information Expert, Sale is a suitable class of object for this responsibility.
- The Sales Line ltem knows its quantity and its associated Product Specification; therefore, by Expert, Sales Line ltem should determine the subtotal; it is the information expert.



Partial Interaction and Class Diagram

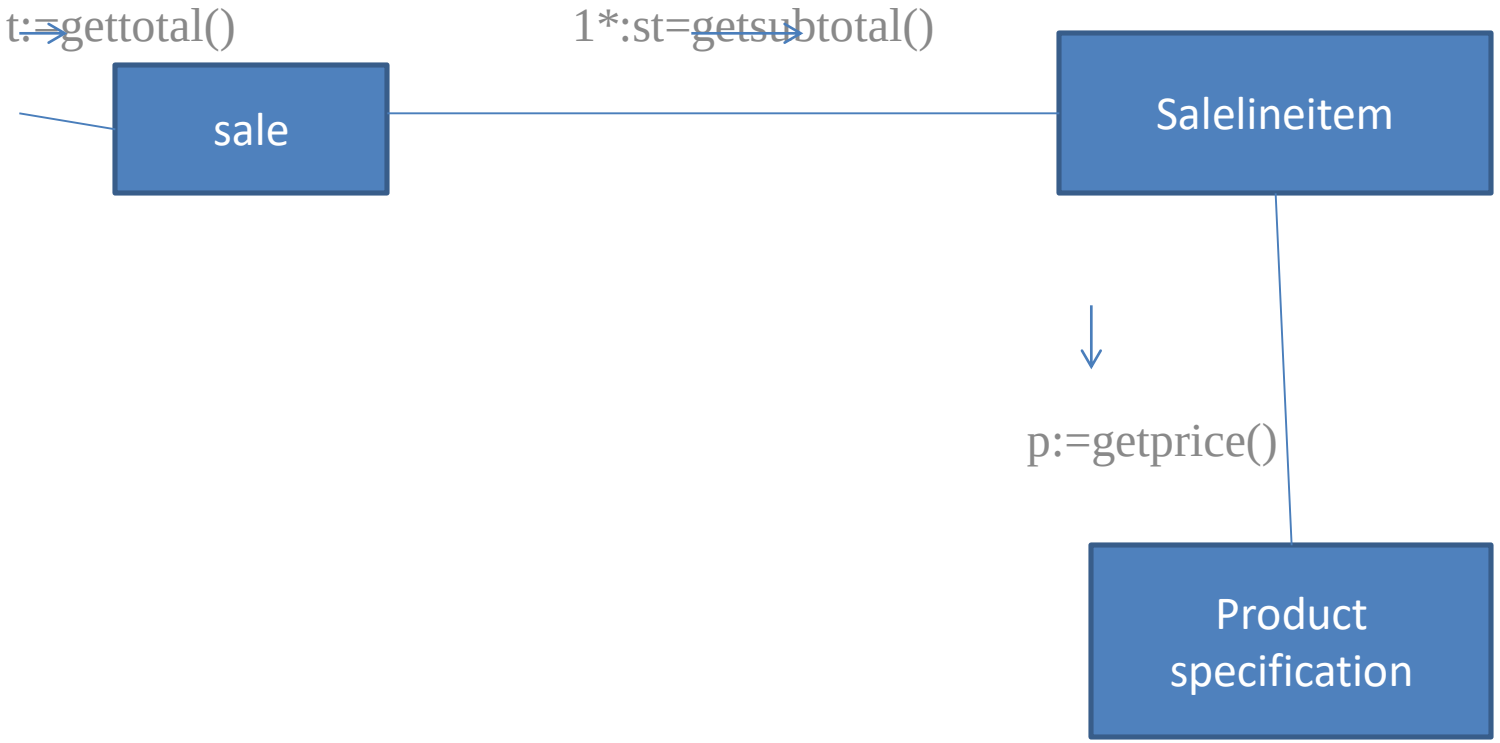


Fig: Calculating the Sale total

- The Product Specification is an Information Expert on answering its price, therefore SalesLineItem send it a message asking for the product price.

To full fill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects as follows.

DESIGN CLASS	DESCRIPTION
Sale	Knows sale total
Sales line item	Knows the line item subtotal
Product specification	Knows product price

LOW COUPLING

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements.

A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems,

- Forced local changes because of changes in related classes.
- Harder to understand in isolation.
- Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Problem:

How to support low dependency, low change impact, and increased reuse?

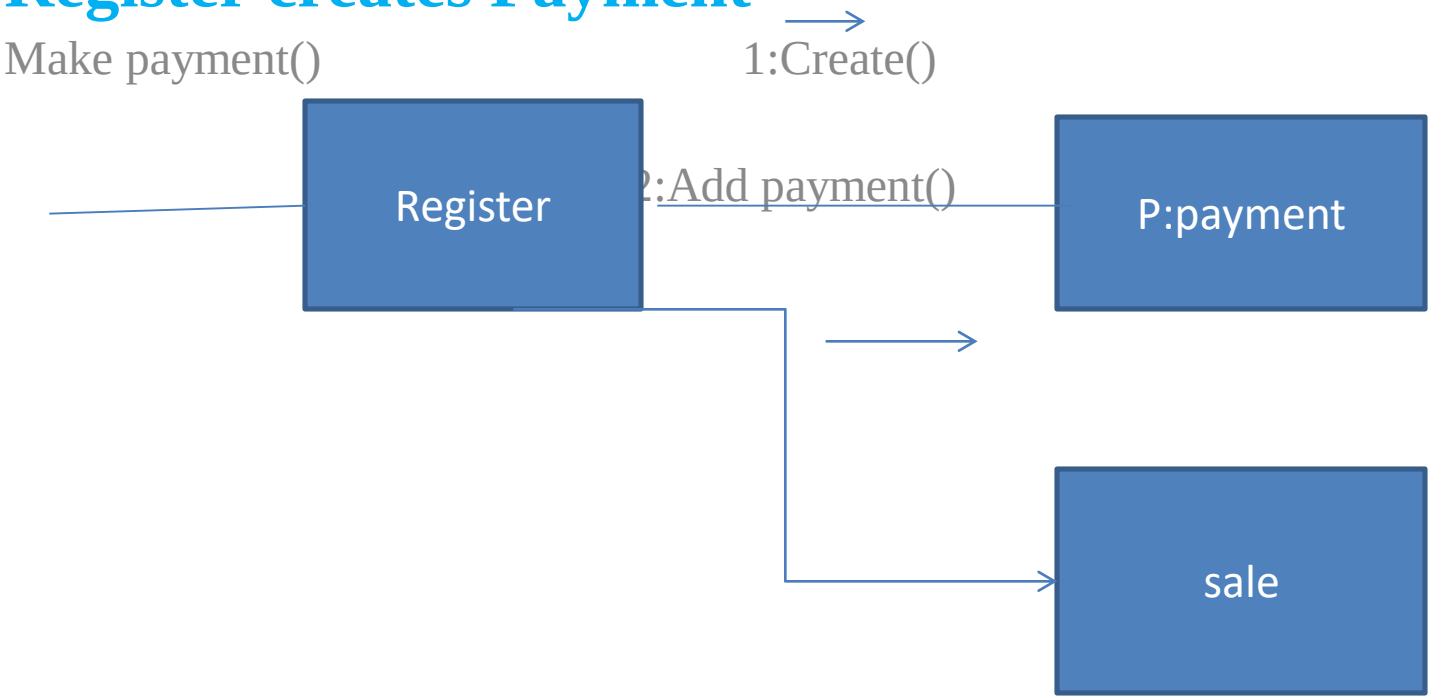
Solution:

Assign a responsibility so that coupling remains low.

Partial Class Domain:

Assume that a Payment instance is to be created and associated with the Sale. What class should be responsible for this? Since a Register "records" a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment. The Register instance could then send an addpayment message to the Sale, passing along the new Payment as a parameter.

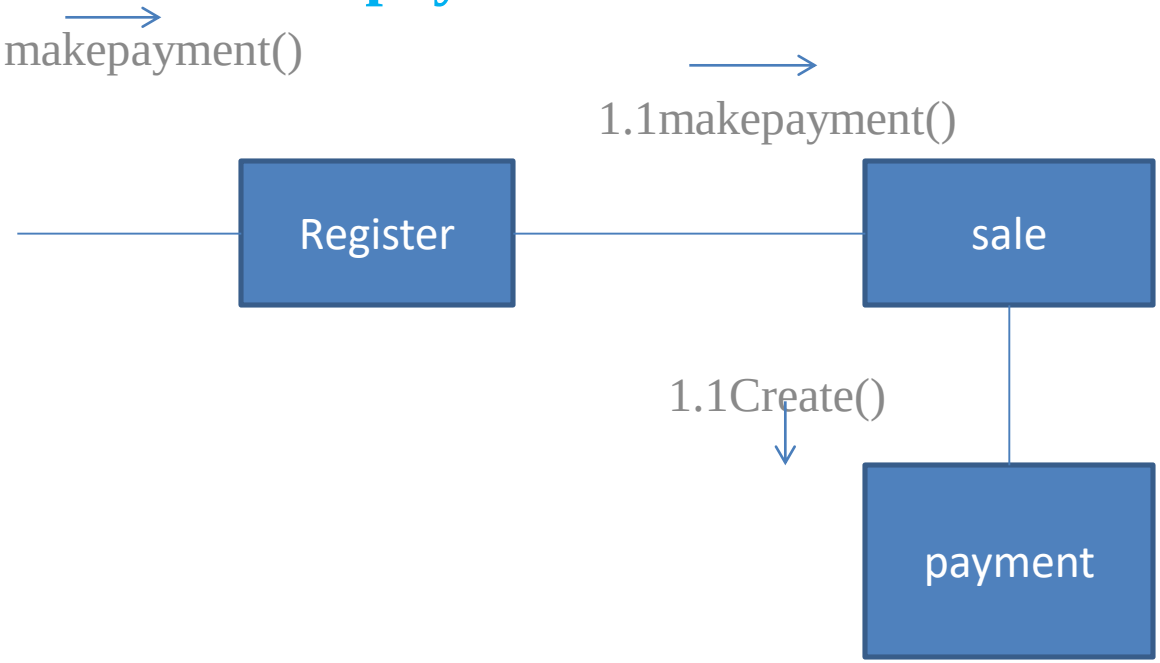
Register creates Payment



Assignment of responsibilities couples the Register class to knowledge of payment class.

Alternative solution to create payment and associate it with Sale.

Sales creates payment



HIGH COHESION

Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities of an element

are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.

Problem

How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

Solution:

Assign a responsibility so that cohesion remains high.

A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems:

✓ Hard to comprehend

✓ Hard to reuse

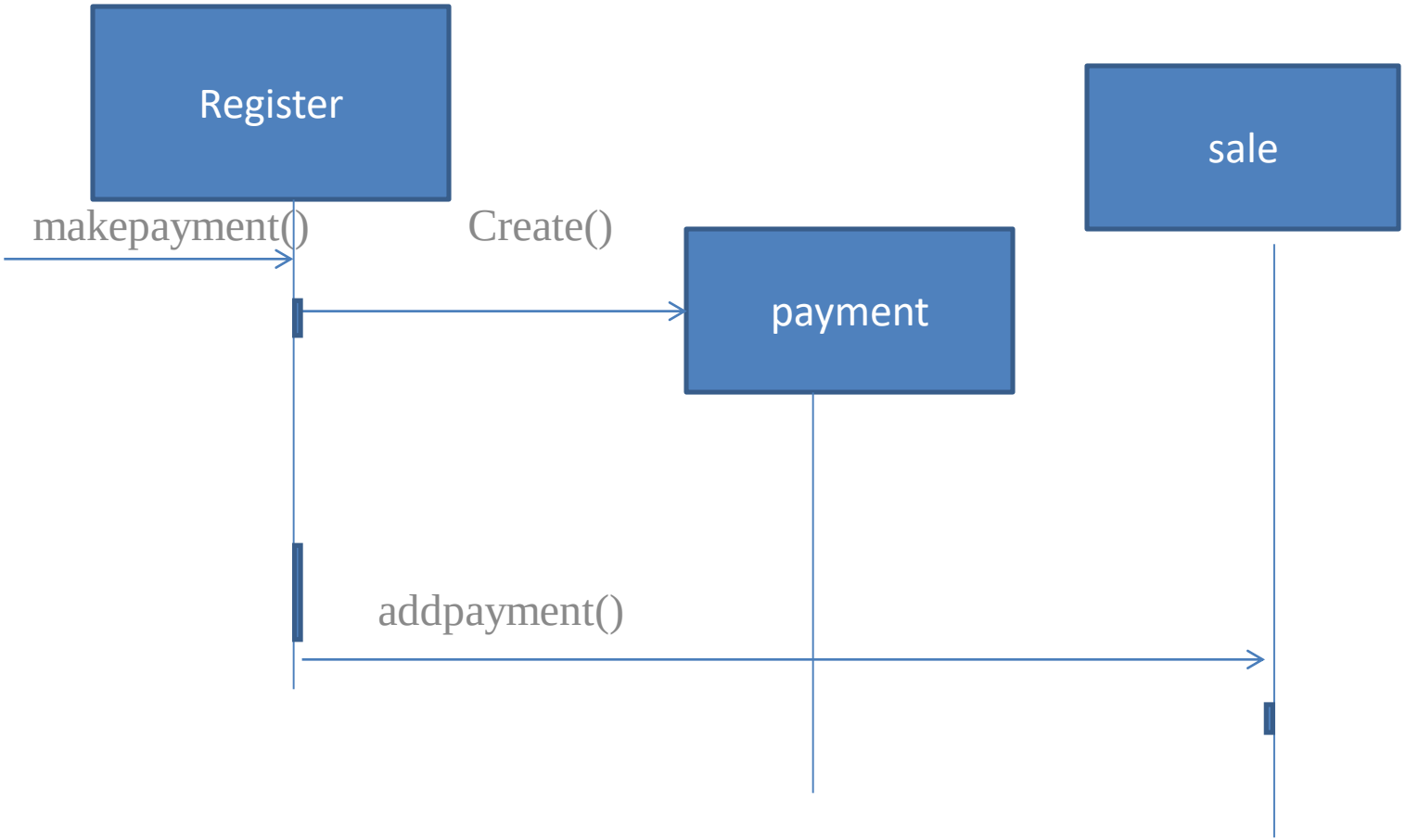
✓ Hard to maintain

✓ Delicate; constantly affected by change.

Example

Assume that a Payment instance is to be created and associate it with the Sale. What class should be responsible for this? Since Register records a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment. The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.

Register Creates Payment



CONTROLLER

A Controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

Problem

What first object beyond the UI layer receives and coordinates(controls) a system operation?

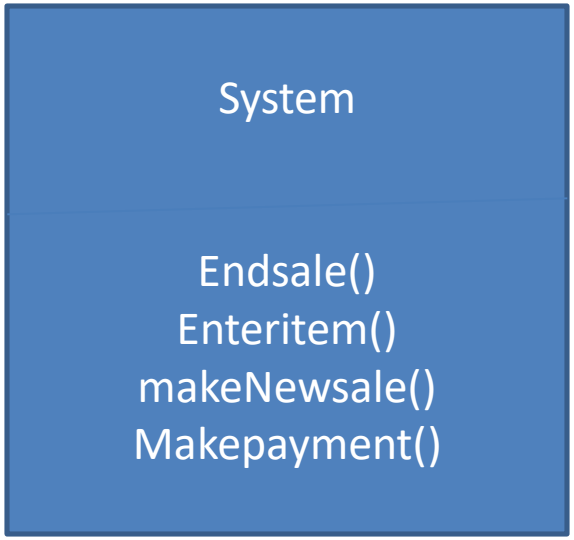
Solution:

Assign the responsibility to a class representing one of the following choices,

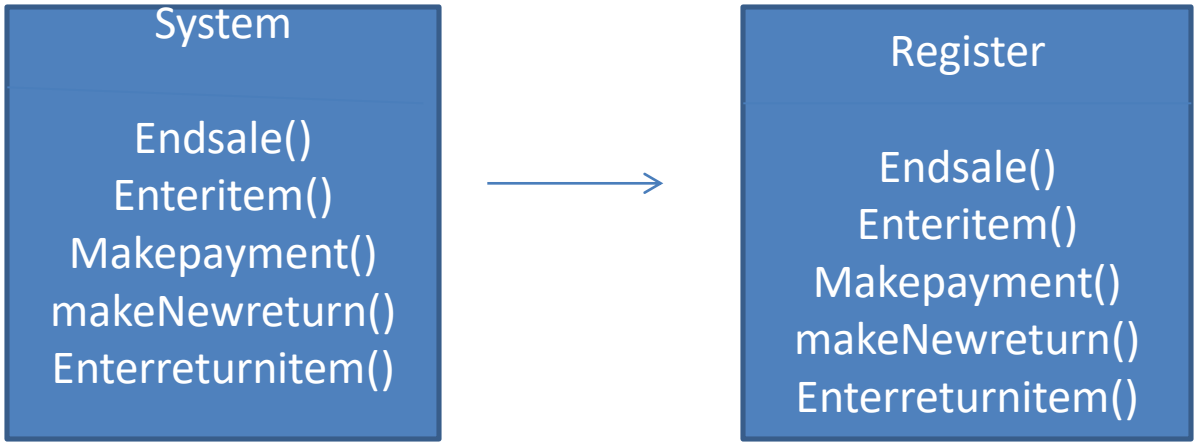
✓ Represents the overall system, “a root object”, a device that the software is running within, or a major subsystem.

✓ Represents a use case scenario within which the system event occurs.

Example: NextGen POS application



Controller Class



During design, a controller class is assigned the responsibility for system operation.

The system Operations identified during system behaviour analysis are assigned to one or more controller classes, such as Register,

Bloated Controller

Poorly designed, a controller class will have low cohesion. unfocused and handling

too many areas of responsibility; this is called a bloated controller.

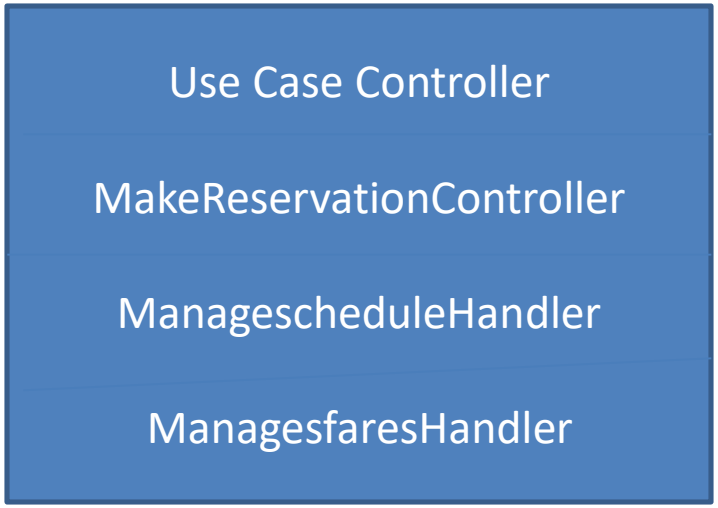
Signs of bloating include:

✓ There is only a single controller class receiving all system events in the system, and there are many of them.

✓ The controller itself performs many of the tasks necessary to fullfill the system event, without delegating the work

✓ A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.

Cures for a bloated controller



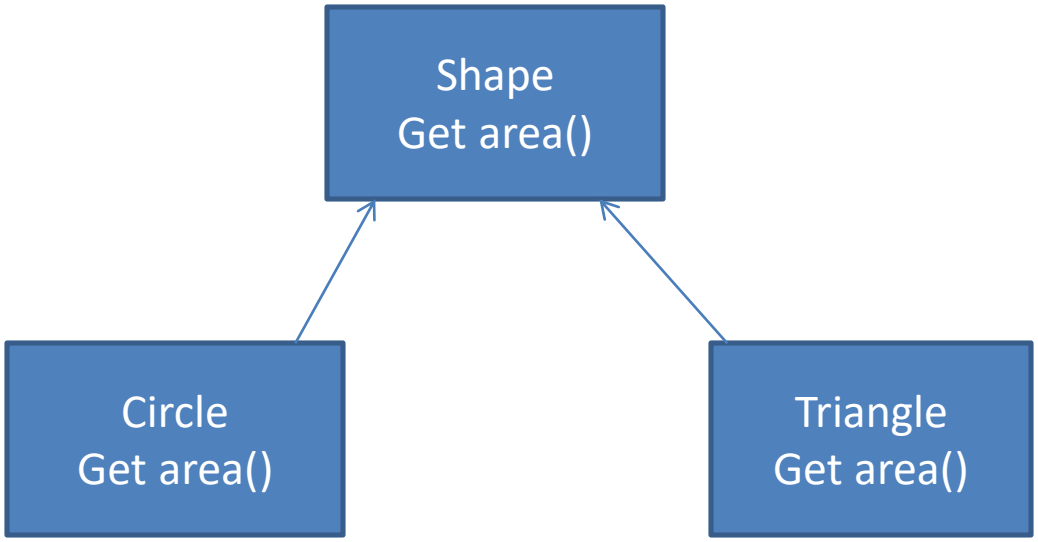
✓ Add more controllers-a system does not have to have only one. For example, consider an application with many system events, such as an airline reservation system.

✓ Design the controller so that it primarily delegates the fullfillment of each system operation responsibility on to other objects.

POLYMORPHISM

- How to handle related but varying elements based on element type?
- Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- Benefits: handling new variations will become easy.

Examples for polymorphism



- the getArea() varies by the type of shape, so we assign that responsibility to the subclasses.
- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.

PURE FABRICATION

- Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.
- Provides a highly cohesive set of activities.
- Behavioural decomposed – implements some algorithm.
- Examples: Adapter, Strategy
- Benefits: High cohesion, low coupling and can reuse this class.

Example

- Suppose we Shape class, if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape in cohesive.

INDIRECTION

- How can we avoid a direct coupling between two or more elements.
- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
- Benefits: low coupling
- Example: Adapter, Facade, Observer

Example

- Here polymorphism illustrates indirection
- Class Employee provides a level of indirection to other units of the system.

PROTECTED VARIATION

- How to avoid impact of variations of some elements on the other elements.
- It provides a well defined interface so that the there will be no affect on other units.
- Provides flexibility and protection from variations.
- Provides more structured design.

DESIGN PATTERN

- Design patterns represent solutions to problems that arise when developing software within a particular context.

  "Patterns == problem/solution pairs in a context"

- Patterns capture the static and dynamic *structure* and *collaboration* among key participants in software designs.

  Especially good for describing how and why to resolve *non-functional issues*

- Patterns facilitate reuse of successful software architectures and designs.

APPLICATIONS

- Wide variety of application domains:

  drawing editors, banking, CAD, CAE, cellular network management, telecomm switches, program visualization

- Wide variety of technical areas:

  user interface, communications, persistent objects, O/S kernels, distributed systems

What is Design Pattern?

Each pattern describes a problem which occurs over and over again in our environment and then 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”

Christopher Alexander, A Pattern Language, 1977

A pattern has 4 essential elements:

- Pattern name
- Problem
- Solution
- Consequences

Pattern Name

- A handle used to describe:
  - a design problem,
  - its solutions and
  - its consequences

- Increases design vocabulary

- Makes it possible to design at a higher level of abstraction

- Enhances communication

Problem

•Describes when to apply the pattern

• Explains the problem and its context

• Might describe specific design problems or class or object structures

• May contain a list of conditions

- must be met
- before it makes sense to apply the pattern

Solution

- Describes the elements that make up the
  - design,
  - their relationships,
  - responsibilities and
  - collaborations
- Does not describe specific concrete implementation
- Abstract description of design problems and
  - how the pattern solves it

Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - evaluate design alternatives and
  - understand costs and
  - understand benefits of applying the pattern
- Includes the impacts of a pattern on a system's:
  - flexibility,
  - extensibility
  - portability

Where Design Patterns Are Used

- Object-Oriented Programming Languages:
  - more amenable to implementing design patterns

- Procedural languages: need to define
  - *Inheritance*,
  - *Polymorphism* and
  - *Encapsulation*

TYPES OF DESIGN PATTERN

- Creational
- Structural
- Behavioral

Creational:

Class: defer some part of object creation to subclasses

Object: Defer object creation to another object

Structural:

Class: use inheritance to compose classes

Object: describe ways to assemble classes

Behavioral:

Class: use inheritance to describe algs and flow of control

Object: describes how a group of objects cooperate to perform task that no single object can complete

Creational Patterns

Factory Method:

method in a derived class creates associations

Abstract Factory:

Factory for building related objects

Builder:

Factory for building complex objects incrementally

Prototype:

Factory for cloning new instances from a prototype

Singleton:

Factory for a singular (sole) instance

Structural Patterns

Adapter:

Translator adapts a server interface for a client

Bridge:

Abstraction for binding one of many implementations

Composite:

Structure for building recursive aggregations

Decorator:

Decorator extends an object transparently

Facade:

simplifies the interface for a subsystem

Flyweight:

many fine-grained objects shared efficiently.

Proxy:

one object approximates another

Behavioral Patterns

Chain of Responsibility

request delegated to the responsible service provider

Command:

request is first-class object

Iterator:

Aggregate elements are accessed sequentially

Interpreter:

language interpreter for a small grammar

Mediator:

coordinates interactions between its associates

Memento:

snapshot captures and restores object states privately

Observer:

dependents update automatically when subject changes

State:

object whose behavior depends on its state

Strategy:

Abstraction for selecting one of many algorithms

Template Method:

algorithm with some steps supplied by a derived class

Visitor:

operations applied to elements of a heterogeneous object structure

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures

  also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs

  make it more widely available
- Patterns help improve developer communication

  Pattern names form a vocabulary
- Patterns help ease the transition to OO technology

Drawbacks to Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple

Teams may suffer from pattern overload

- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a SW development process is a human-intensive activity