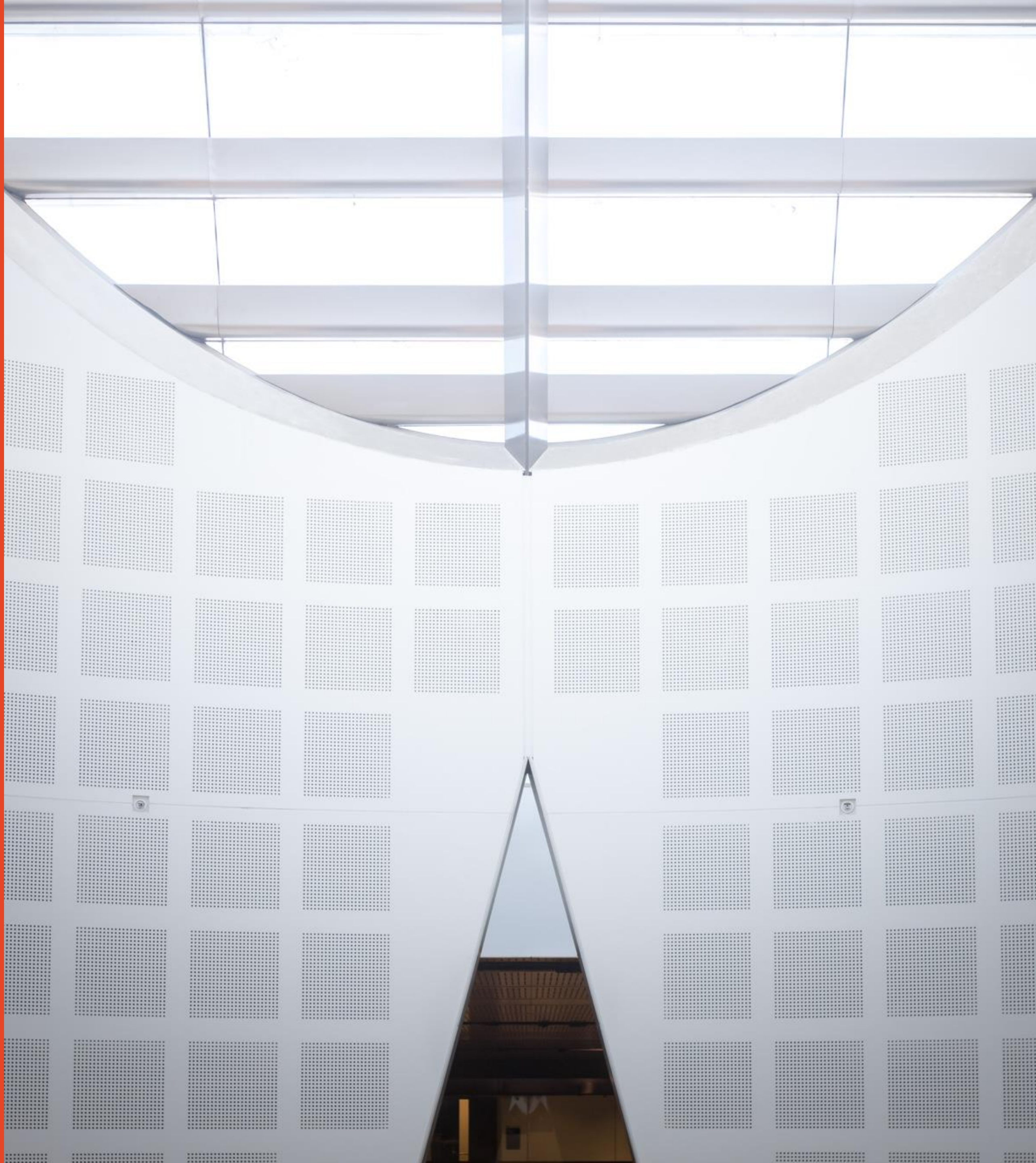
Software Design and Construction 2 SOFT3202 / COMP9202

Review Materials

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Additional Slides

API Design Principles

Information Hiding (1)

- Minimize the accessibility of classes, fields, and methods
  - Speed up development and maintenance

Information Hiding (2)

- In Public classes, use accessor methods, not public fields
  – Consider:

```
public class Point
{
public double x;
public double y;
}
```

vs

```
public class Point
{
private double x;
private double y;
public double getX() { /* … */ }
public double getY() { /* … */ }
}
```

Information Hiding (3) – Interfaces vs Abstract Classes

- Unlike Interfaces, abstract classes can contain implementations for some methods

- To implement the type define by an abstract class, a class must be a subclass of the abstract class

- Prefer interfaces over abstract classes
  - Interfaces provide greater flexibility, avoid needless implementation details
    - Exception: ease of evolution is more significant

Information Hiding (4) – Factory Methods

- Consider implementing a factory method instead of a constructor

– Factory methods provide additional flexibility
  - Can have a descriptive method name
    - BigInteger(int, int, Random) vs BigInteger.probablePrime
  - Not required to create a new object each time they are invoked
    - E.g., Boolean.valueof – never creates an object
  - Flexible in choosing the class of the returned type
    - Can return instance of any subtype of their return type

Information Hiding (4) – Factory Methods

- Disadvantages of factory method
  - Classes without public or protected constructors cannot be sub-classed
  - Factory methods not readily distinguishable from other static methods

Information Hiding (5) – Inheritance vs. Composition

- Inheritance may lead to fragile software if used inappropriately
  - Safe when:
    - Subclass and superclass implementations by same programmers
    - Extending classes specifically designed for extensions
  - Dangerous when:
    - Inheritance across package boundaries!
      - Inheritance violates encapsulation – a subclass depends on the implementation details of its superclass (which may change)
      - Superclass acquire new methods
  - Inheritance is appropriate when there is a Is-A relationship
    - Subclass is really a subtype of the superclass

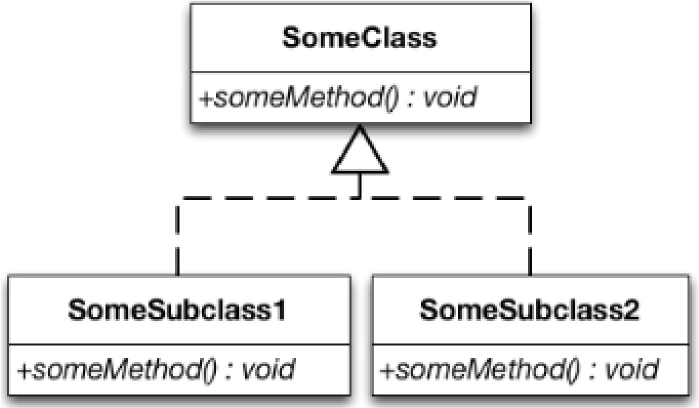
Information Hiding (5) – Inheritance vs. Composition

- Composition
  - Private field to reference an instance of existing class
  - Forwarding and forwarding methods
  - Better design
    - No dependencies on implementation details of existing class
    - No impact when adding new methods to the existing class
  - Using inheritance where composition is more appropriate will expose implementation details
    - API ties you to the original implementation

- Favor composition over inheritance!

Do not Violate Liskov's Behavioral Subtyping Rules

" If you have base class BASE and subclasses SUB1 and SUB2, the rest of your code should always refer to BASE Not SUB1 and SUB2"

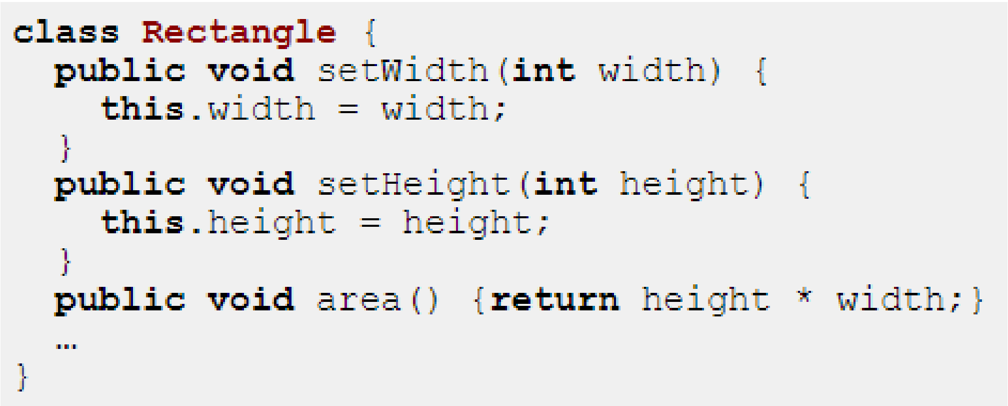


- Use inheritance only for true subtypes
  - Is-A relationship
- Favour composition over inheritance

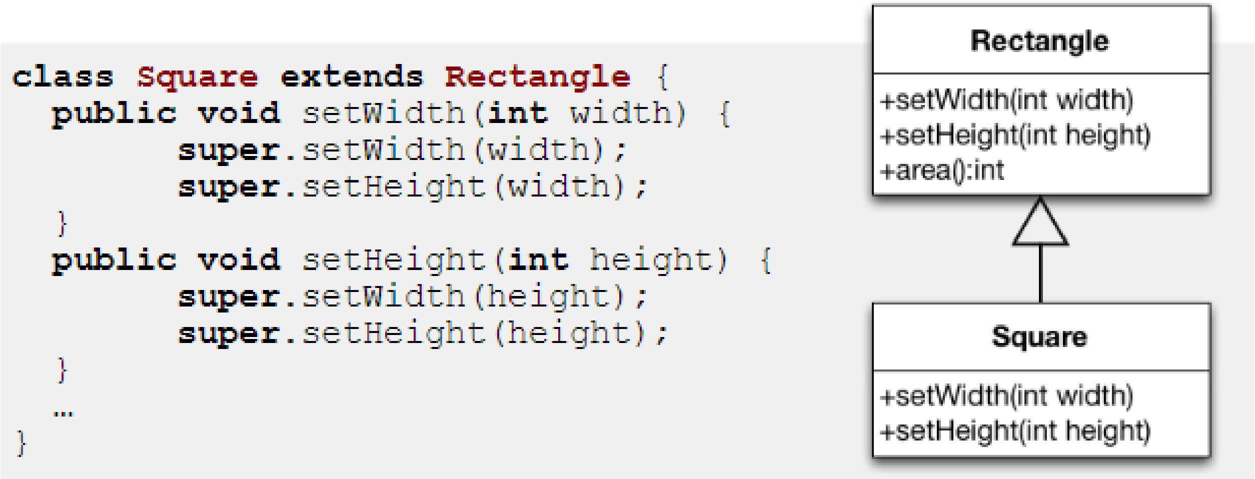
– Example: Square class deriving from Rectangle

http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document

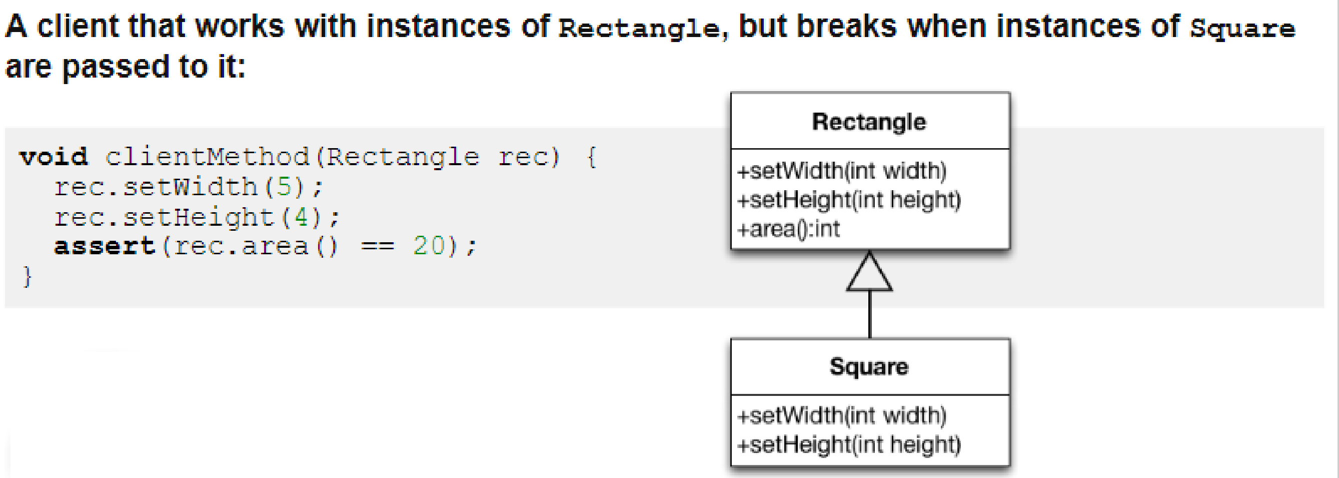
Liskov's Substitution – Example



```
class Rectangle {
  public void setWidth(int width) {
     this.width = width;
  }
  public void setHeight(int height) {
     this.height = height;
  }
  public void area() {return height * width;}
  …
}
```



```
class Square extends Rectangle {
  public void setWidth(int width) {
         super.setWidth(width);
         super.setHeight(width);
  }
  public void setHeight(int height) {
         super.setWidth(height);
         super.setHeight(height);
  }
  …
}
```



A client that works with instances of `Rectangle`, but breaks when instances of `Square` are passed to it:

```
void clientMethod(Rectangle rec) {
  rec.setWidth(5);
  rec.setHeight(4);
  assert(rec.area() == 20);
}
```

http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document

Minimize Mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
    - See java.lang.String
  - Disadvantage: separate object for each value

- Mutable objects require careful management of visibility and side effects
  - e.g. Component.getSize() returns a mutable Dimension

- Document mutability
  - Carefully describe state space

How to Make a Class Immutable?

- Don’t provide mutators (methods that modify the object’s state
- Ensure that the class cannot be extended (
  - Subclass (as final) to prevent it rom compromising the immutable behavior
- Make all fields final
- Make all fields private
  - Prevent clients from accessing to mutable objects
- Ensure exclusive access to any mutable components
  - Do not share object reference

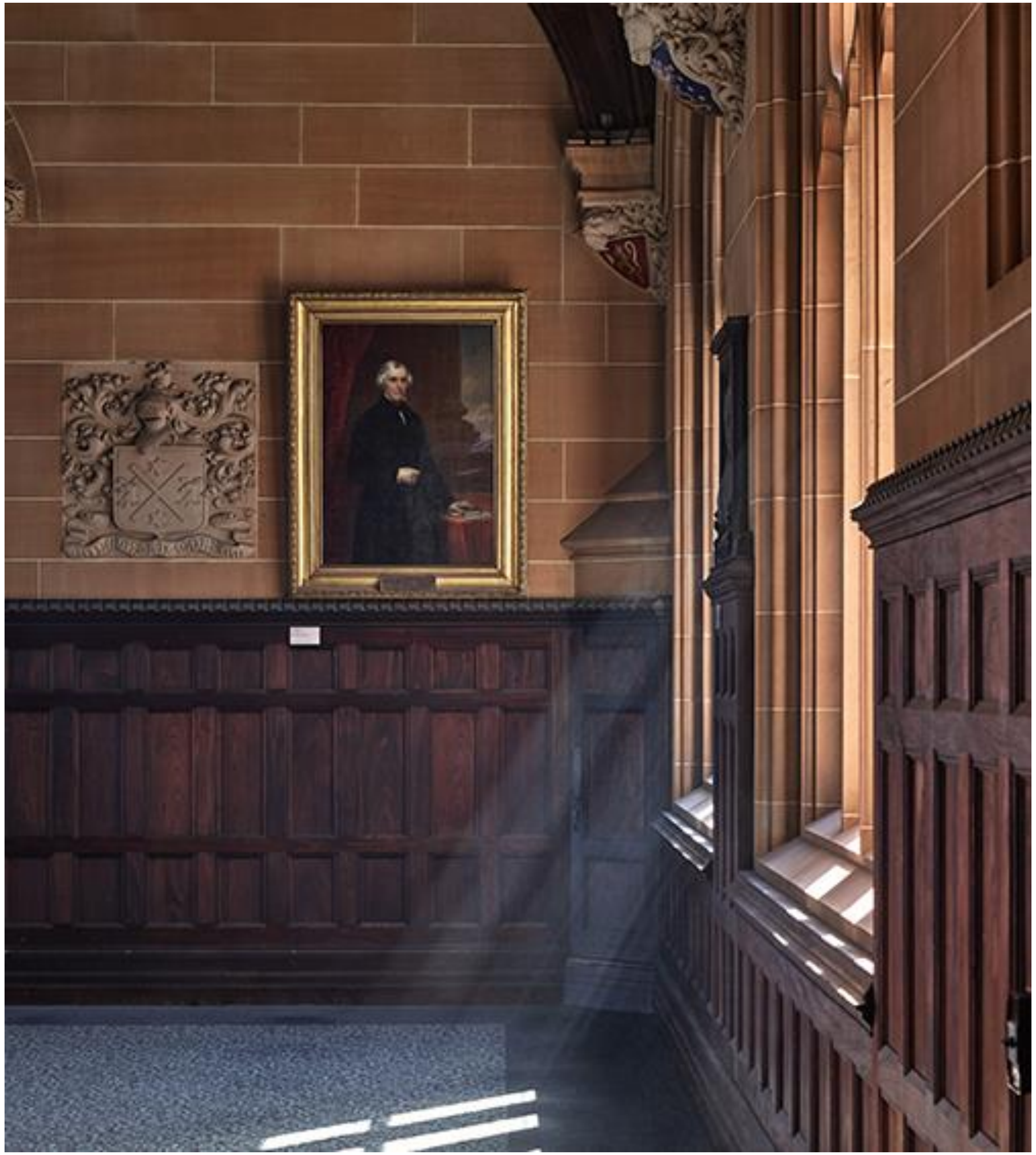
Use Appropriate Parameter and Return Types

- Favor interface types over classes for input
- Use most specific type for input type
- Do not return a String if a better type exists
- Do not use floating point for monetary values
- Use double (64 bits) instead of float (32 bits)

GRASP Principles

Revisit

THE UNIVERSITY OF SYDNEY



Responsibility Driven Design

- Responsibility is a contract or obligation of a class

- What must a class “know”? [knowing responsibility]
  - Private encapsulated data
  - Related objects
  - Things it can derive or calculate

- What must a class “do”? [doing responsibility]
  - Take action (create an object, do a calculation)
  - Initiate action in other objects
  - Control/coordinate actions in other objects

- Responsibilities are assigned to classes of objects during object design

Responsibilities: Examples

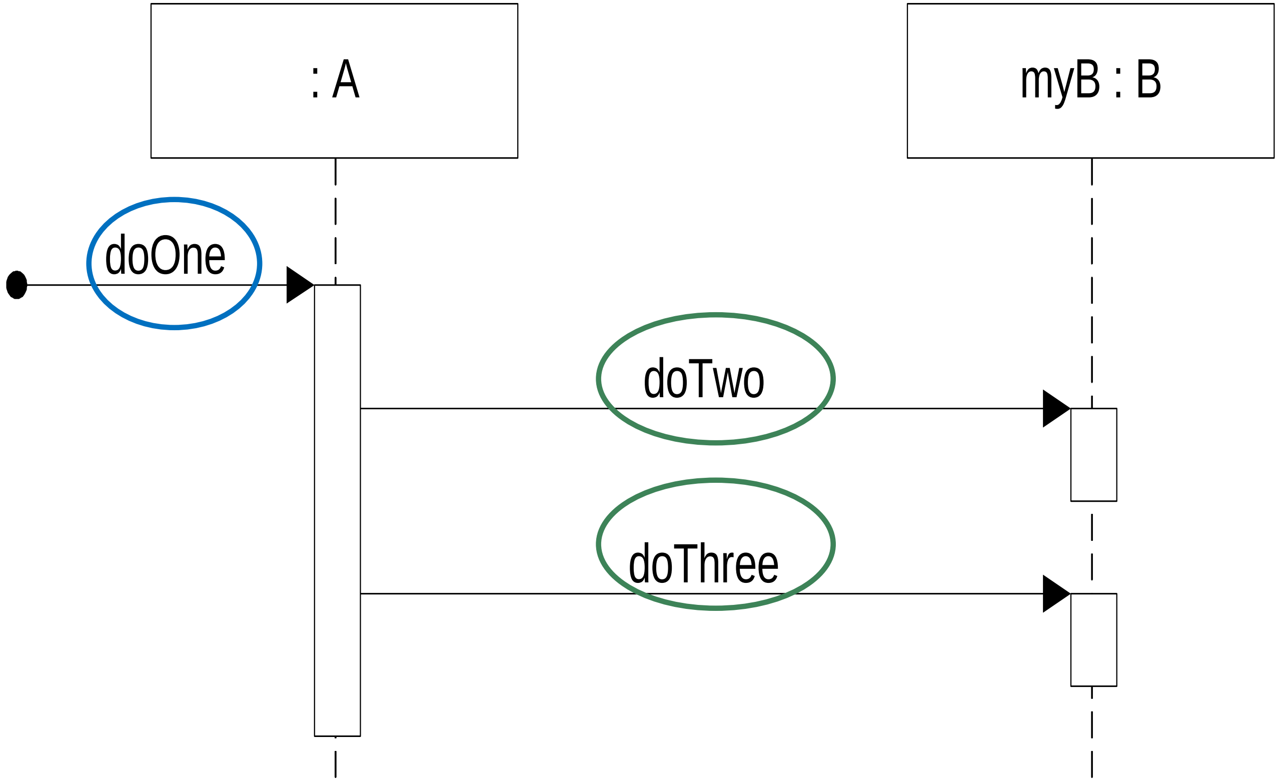
- “A *Sale* is responsible for creating *SalesLineItems*” (doing)

- “A *Sale* is responsible for knowing its total” (knowing)

- Knowing responsibilities are related to attributes, associations in the domain model

- Doing responsibilities are implemented by means of methods.

Doing Responsibilities: Example



GRASP: Methodological Approach to OO Design

General **R**esponsibility **A**ssignment **S**oftware **P**atterns

The five basic principles:

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller

GRASP: Creator Principle

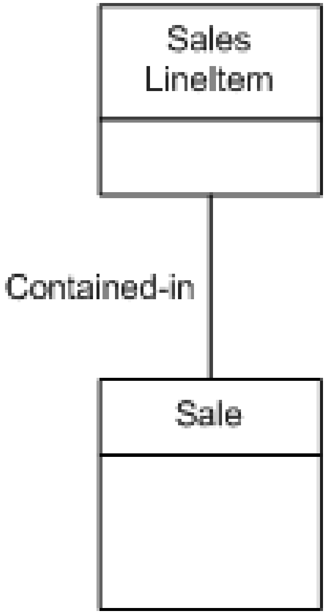
Problem

Who creates an A object

Solution

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

- B “contains” A
- B “records” A
- B “closely uses” A
- B “has the Initializing data for” A



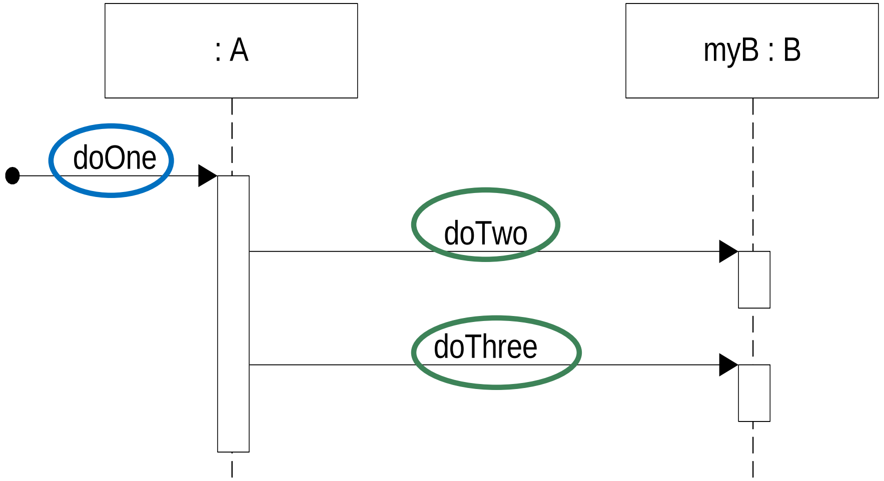
GRASP: Information Expert Principle

Problem

What is a general principle of assigning responsibilities to objects

Solution

Assign a responsibility to the class that has the information needed to fulfill it



Dependency

- A dependency exists between two elements if changes to the definition of one element (the **supplier**) may cause changes to the other (the **client**)

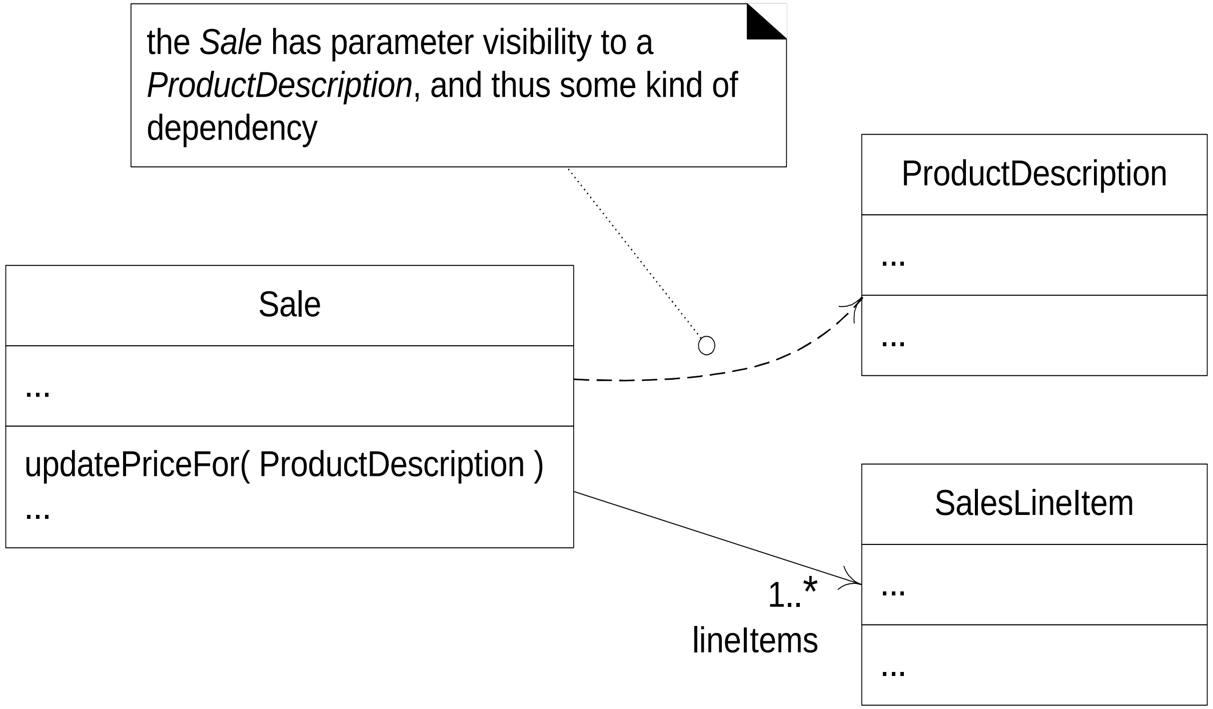
- Various reason for dependency
  - Class send message to another
  - One class has another as its data
  - One class mention another as a parameter to an operation
  - One class is a superclass or interface of another

When to show dependency?

- Be <u>selective</u> in describing dependency
- Many dependencies are already shown in other format
- Use dependency to depict global, parameter variable, local variable and static-method.
- Use dependencies when you want to show how changes in one element might alter other elements

Dependency: Parameter Variable

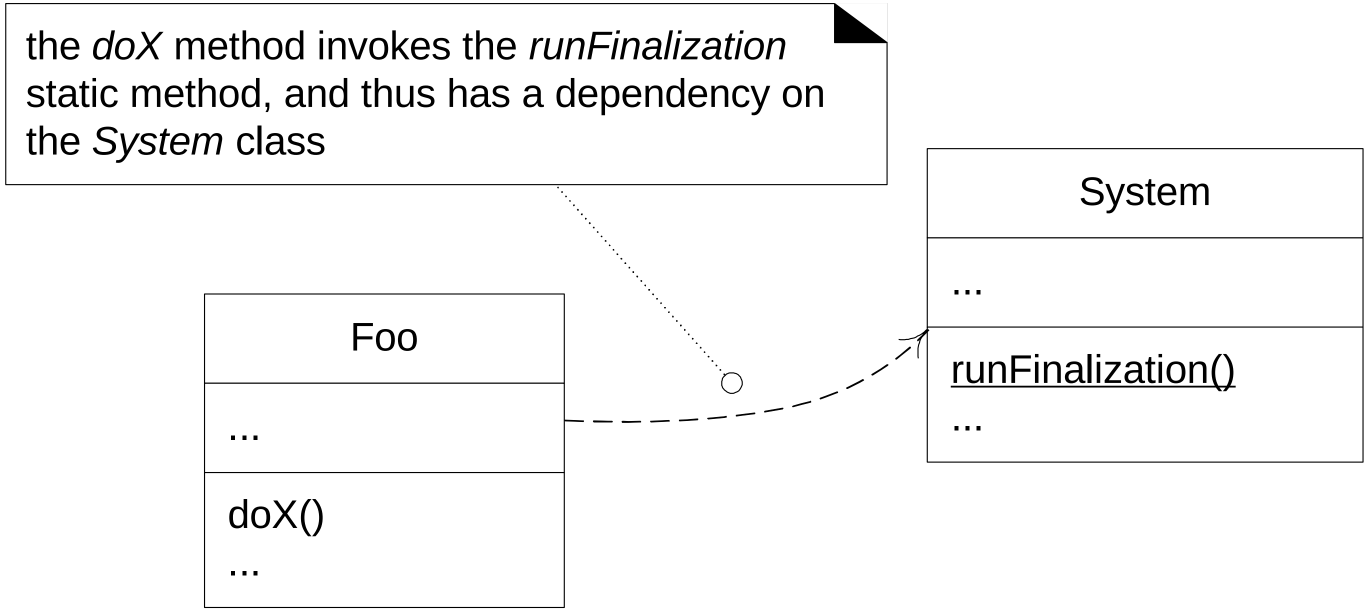
```
  public class Sale{
    public void updatePriceFor (ProductDescription description){
            Money basePrice = description.getPrice();
            //…
    }
}
```



the *Sale* has parameter visibility to a *ProductDescription*, and thus some kind of dependency

Dependency: static method

```
public class Foo{
        public void doX(){
          System.runFinalization();
          //..
        }
}
```



the *doX* method invokes the *runFinalization* static method, and thus has a dependency on the *System* class

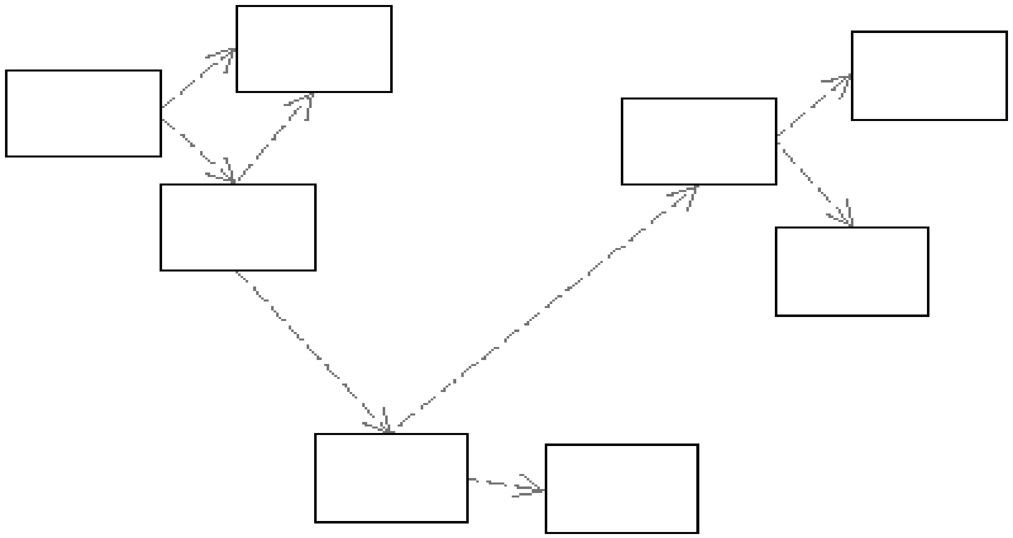
Dependency labels

- There are many varieties of dependency, use keywords to differentiate them

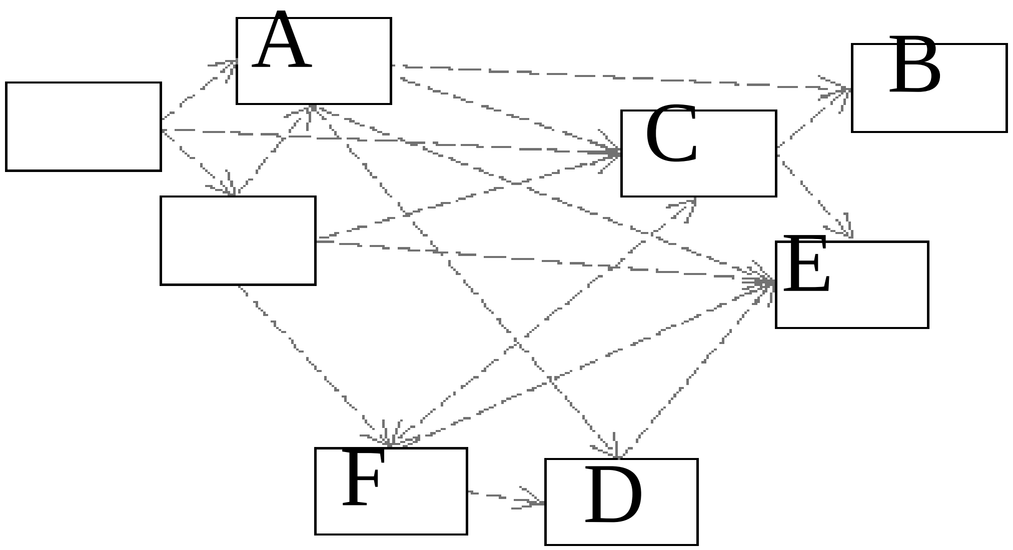
- Different tools have different sets of supported dependency keywords.
  - <<call>> the source calls an operation in the target
  - <<use>> the source requires the targets for its implementation
  - <<parameter>> the target is passed to the source as parameter.

Coupling

- How strongly one element is connected to, has knowledge of, or depends on other elements
- Illustrated as dependency relationship in UML class diagram



Low coupling



High coupling

GRASP: Low Coupling Principle

Problem

How to reduce the impact of change, to support low dependency, and increase reuse?

Solution

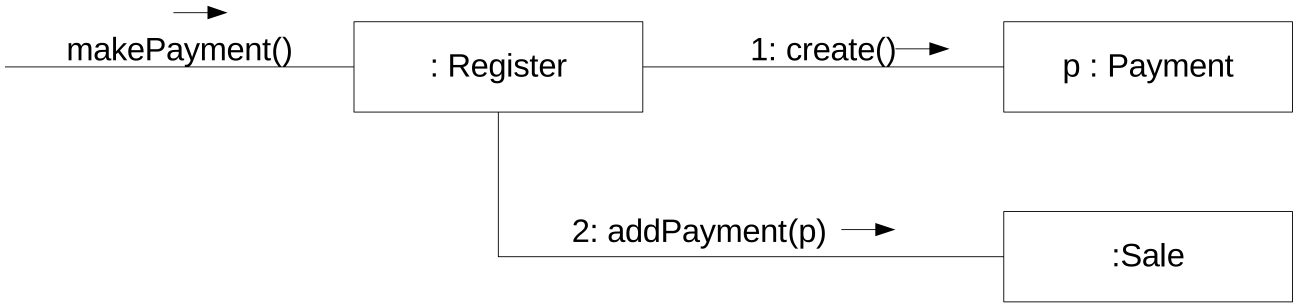
Assign a responsibility so that coupling remains low

Coupling – Example (NextGen POS)

We need to create a *Payment* instance and associate it with the *Sale*.
What class should be responsible for this?

Payment
Sale
Register

Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment

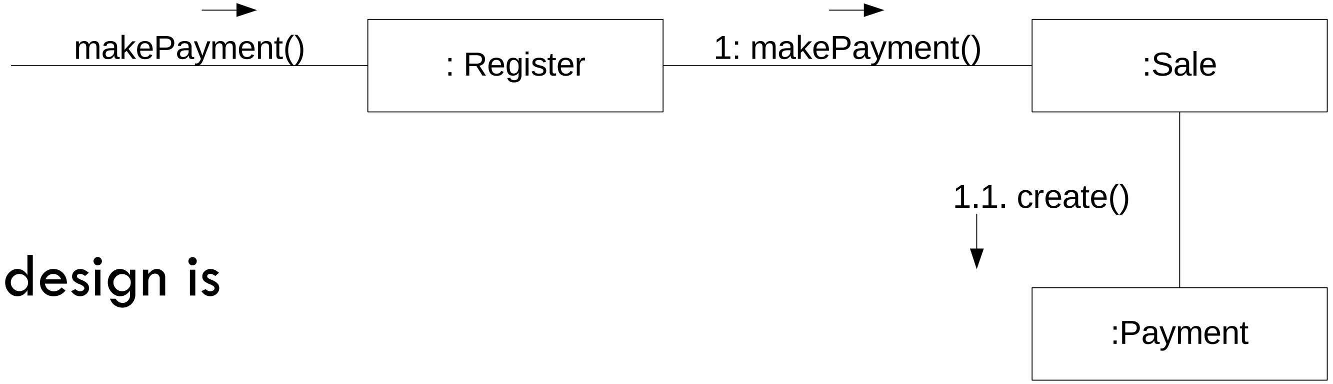


This assignment couple the *Register* class to knowledge of the *Payment* class which increases coupling

Coupling – Example (NextGen POS)

Better design from coupling point of view

It maintains overall lower coupling



From creator point of view, the previous design is better.

In practice, consider the level of couple along with other principles such as Expert and High Cohesion

Cohesion

- How strongly related and focused the responsibilities of an element are

- Formal definition (calculation) of cohesion
  - Cohesion of two methods is defined as the intersection of the sets of instance variables that are used by the methods
  - If an object has different methods performing different operations on the same set of instance variables, the class is cohesive

High Cohesion

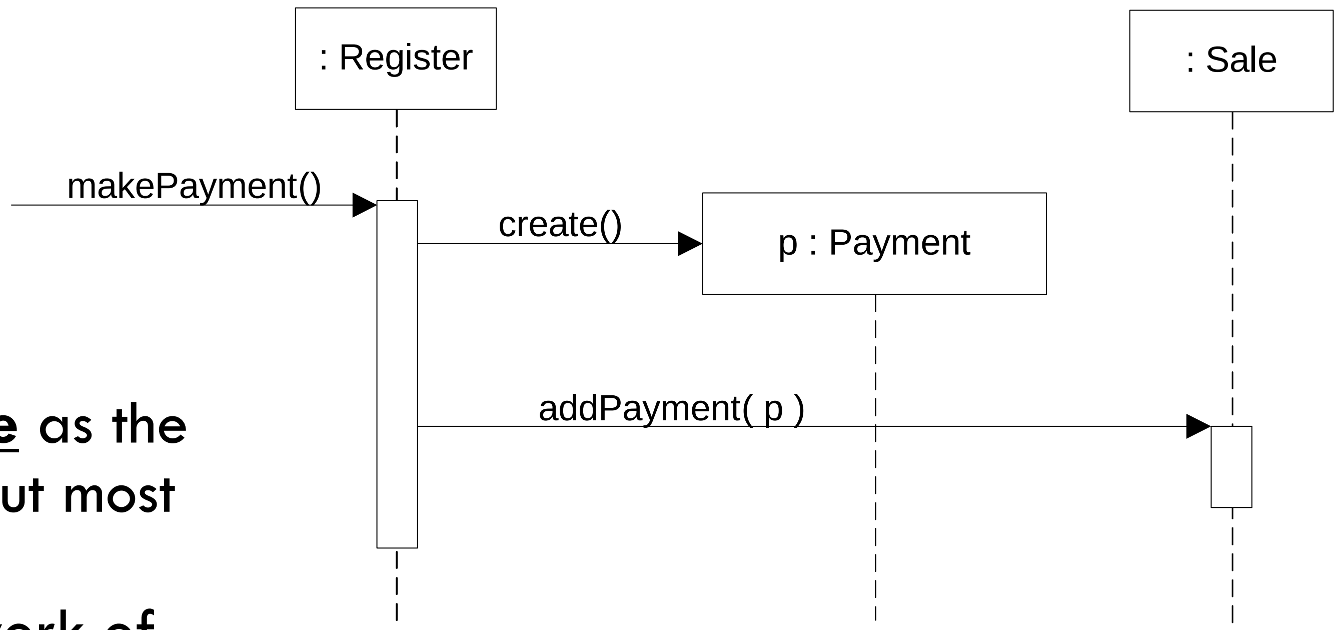
- Problem
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

- Solution
  - Assign responsibilities so that cohesion remains high

Cohesion – Example (NextGen POS)

We need to create a (cash) *Payment* instance and associate it with the *Sale*.
What class should be responsible for this?

Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment



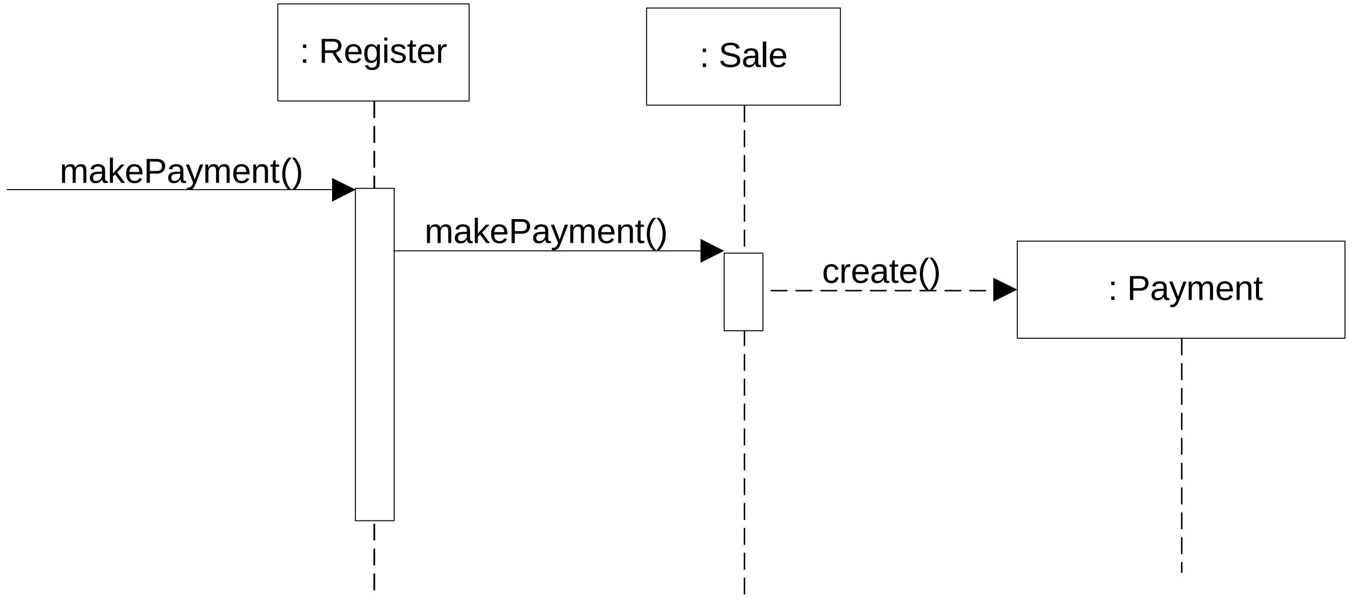
Acceptable but could become **<u>incohesive</u>** as the Register will increasingly need to carry out most of the system operations assigned to it
e.g., Register responsible for doing the work of 20 operations (overburden)

Cohesion – Example (NextGen POS)

Better design from cohesion point of view

The *payment* creation responsibility is delegated to the *Sale* instance

It supports high cohesion and low coupling



Coupling and Cohesion

- Coupling describes the inter-objects relationship
- Cohesion describes the intra-object relationship
- Extreme case of “coupling”
  - Only one class for the whole system
  - No coupling at all
  - Extremely low cohesion
- Extreme case of cohesion
  - Separate even a single concept into several classes
  - Very high cohesion
  - Extremely high coupling
- Domain model helps to identify concepts
- OOD helps to assign responsibilities to proper concepts

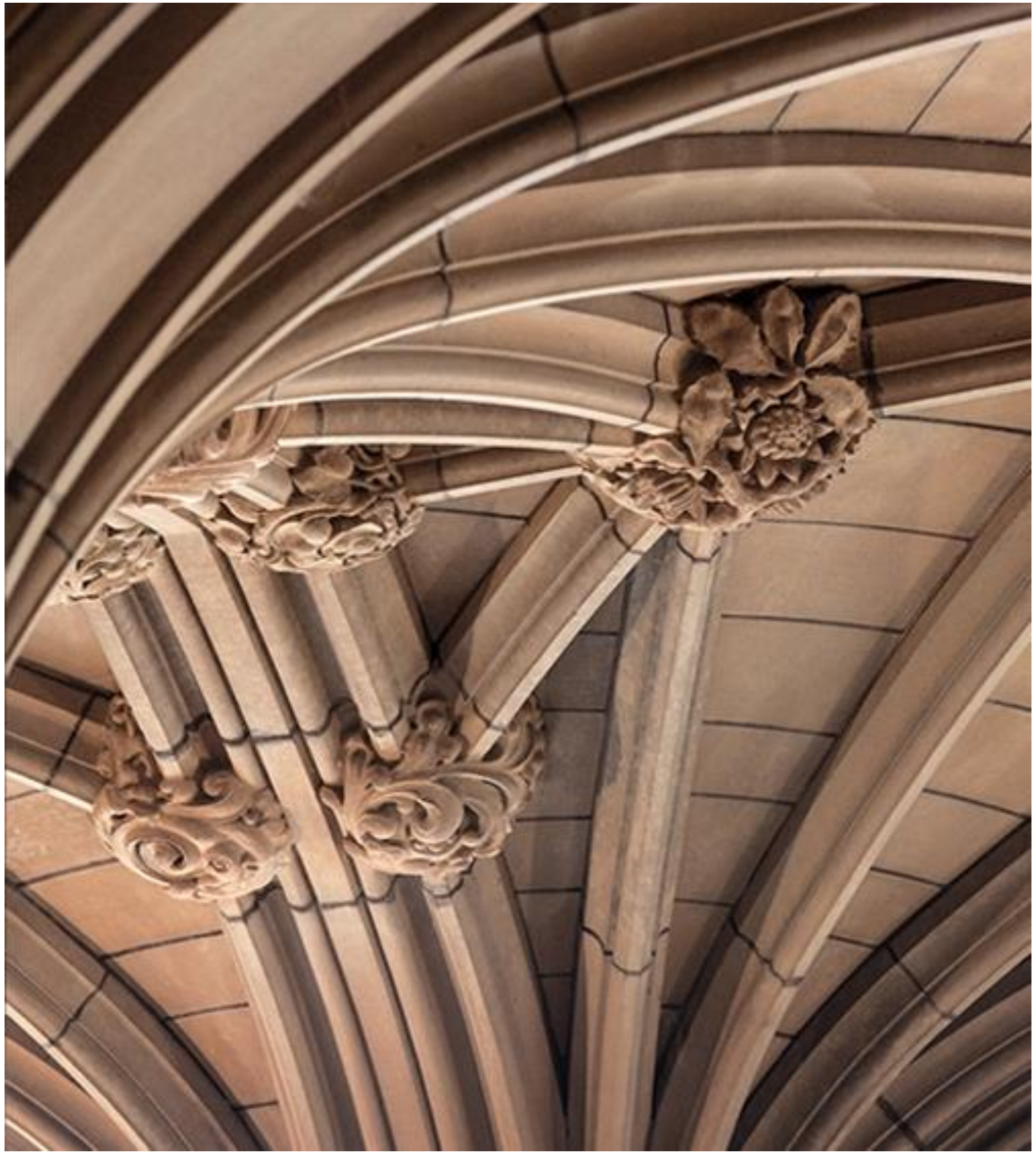
Controller

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

- Solution
  - Assign the responsibility to an object representing one of these choices
    - Represents the overall system, root object, device or subsystem (a façade controller)
    - Represents a use case scenario within which the system operations occurs (a use case controller)

NextGen POS System
OO Analysis & Design

Revision material – examples rely on POS analysis and design



Software Modelling Case Study

NextGen POS software modeling

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*.

THE UNIVERSITY OF SYDNEY

Next Gen Point-of-Sale (POS) System

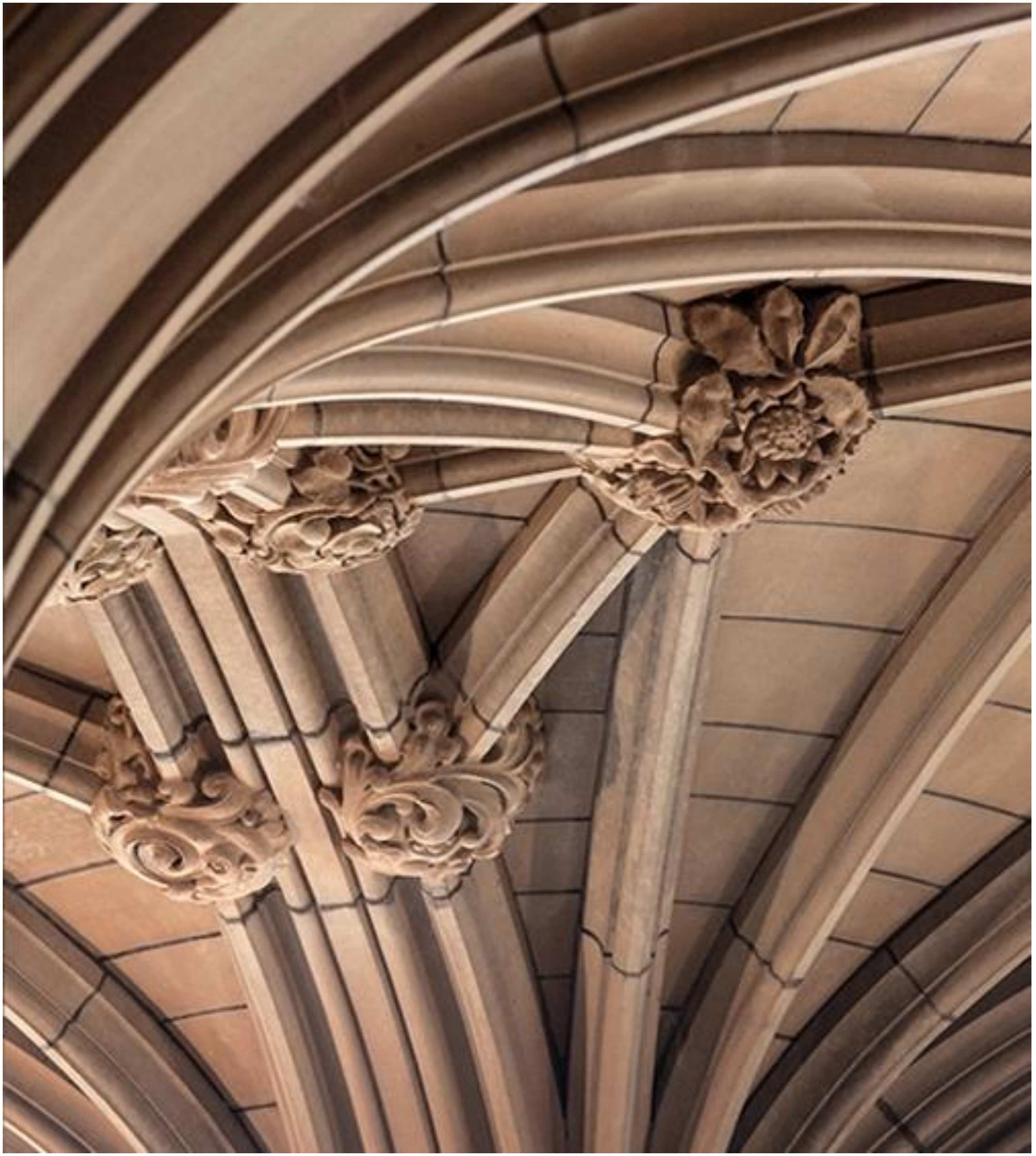
- A POS is a computerized application used (in part) to record sales and handle payments
  - Hardware: computer, bar code scanner
  - Software
  - Interfaces to service applications: tax calculator, inventory control
  - Must be fault-tolerant (can capture sales and handle cash payments even if remote services are temporarily unavailable
  - Must support multiple client-side terminals and interfaces; web browser terminal, PC with appropriate GUI, touch screen input, and Wireless PDAs
  - Used by small businesses in different scenarios such as initiation of new sales, adding new line item, etc.

Next Gen POS Analysis

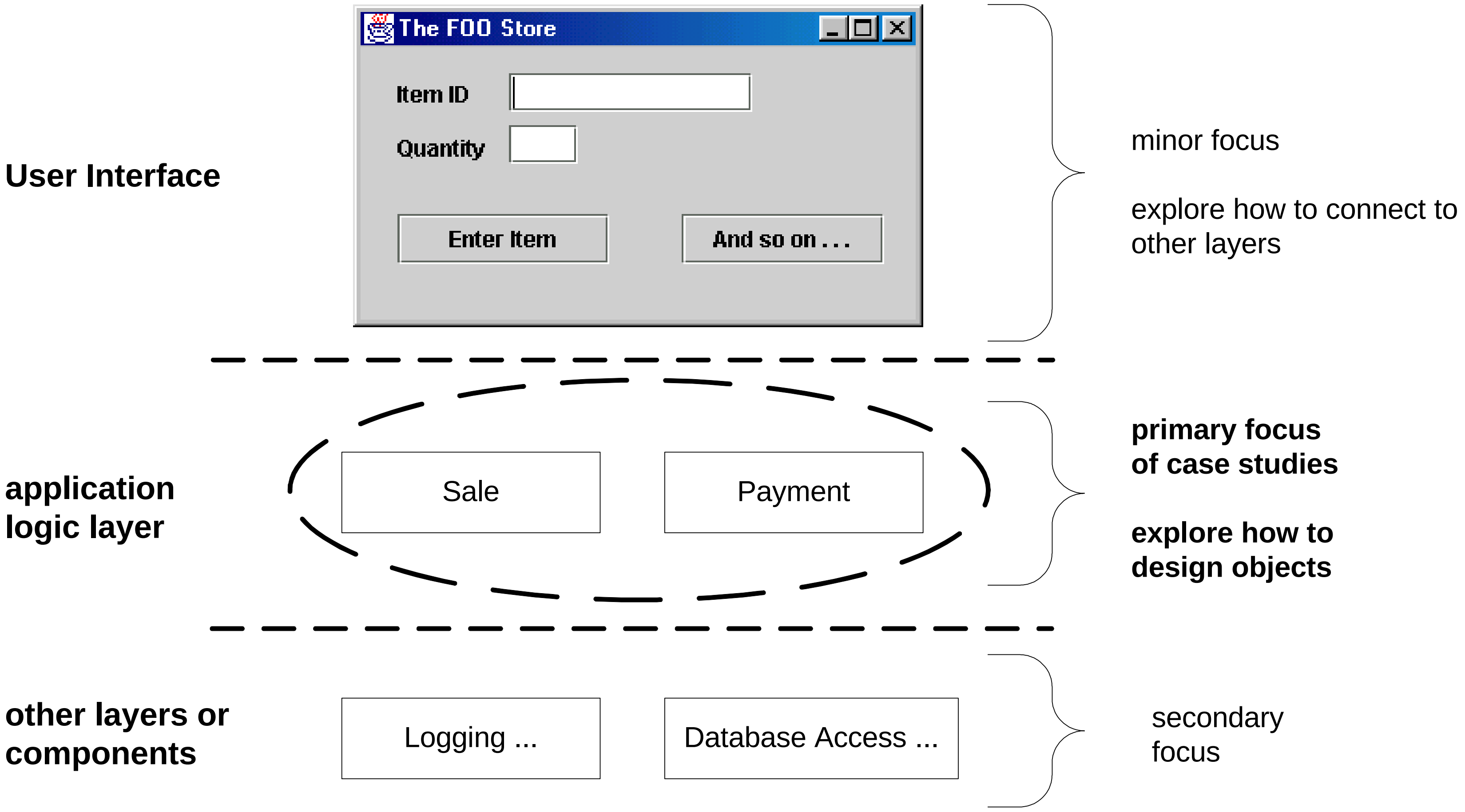
Scope of OOA & D and Process Iteration

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*.

THE UNIVERSITY OF SYDNEY



Next Gen POS – Scope (Analysis & Design)



User Interface

minor focus

explore how to connect to other layers

application logic layer

primary focus of case studies

explore how to design objects

other layers or components

secondary focus

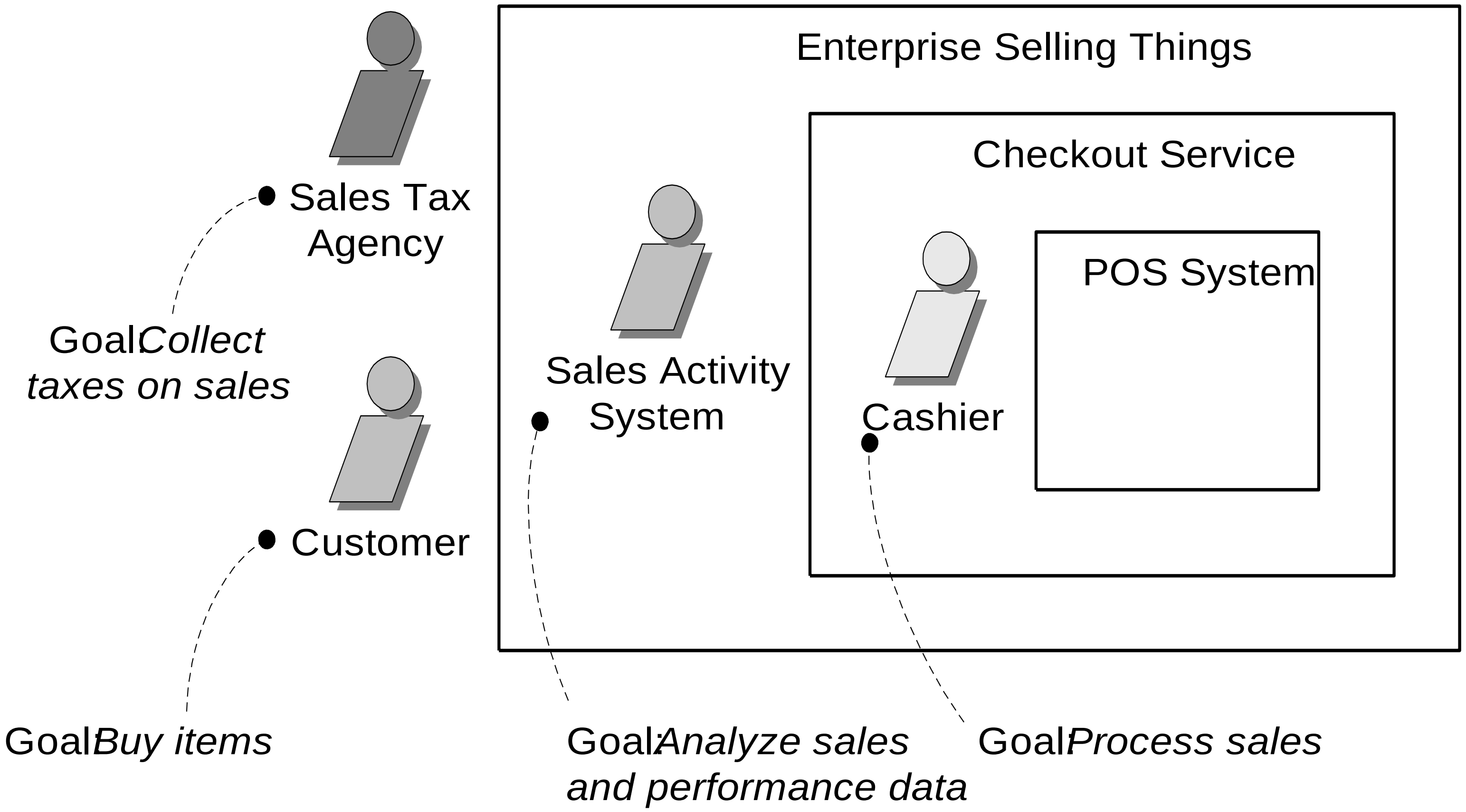
Next Gen POS Case Study: Analysis

OO Analysis with UML

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).*

THE UNIVERSITY OF SYDNEY

Analysis (Requirements): Actors, Goals, System Boundaries



NextGen POS: Process Sale Use Case Description

Use case UC1: Process Sale

Primary Actor: Cashier

Stakeholders and Interests:

-Cashier: Wants accurate and fast entry, no payment errors, …

-Salesperson: Wants sales commissions updated.

…

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions):

-Sale is saved. Tax correctly calculated.

…

Main success scenario (or basic flow): [see next slide]

Extensions (or alternative flows): [see next slide]

Special requirements: Touch screen UI, …

Technology and Data Variations List:

-Identifier entered by bar code scanner,…

Open issues: What are the tax law variations? …

Main success scenario (or basic flow):

The Customer arrives at a POS checkout with items to purchase.

The cashier records the identifier for each item. If there is more than one of the same item, the Cashier can enter the quantity as well.

The system determines the item price and adds the item information to the running sales transaction. The description and the price of the current

item are presented.

On completion of item entry, the Cashier indicates to the POS system that item entry is complete.

The System calculates and presents the sale total.

The Cashier tells the customer the total.

The Customer gives a cash payment (“cash tendered”) possibly greater than the sale total.

Extensions (or alternative flows):

If invalid identifier entered. Indicate error.

If customer didn’t have enough cash, cancel sales transaction.

Next Gen POS Use Case Diagram

UC1: ***Process Sale***

…

Main Success Scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.

…

7. Customer pays and System handles payment

Extensions:

7b. Paying by credit: Include *Handle Credit Payment.*

7c. Paying by check: Include *Handle Check Payment*

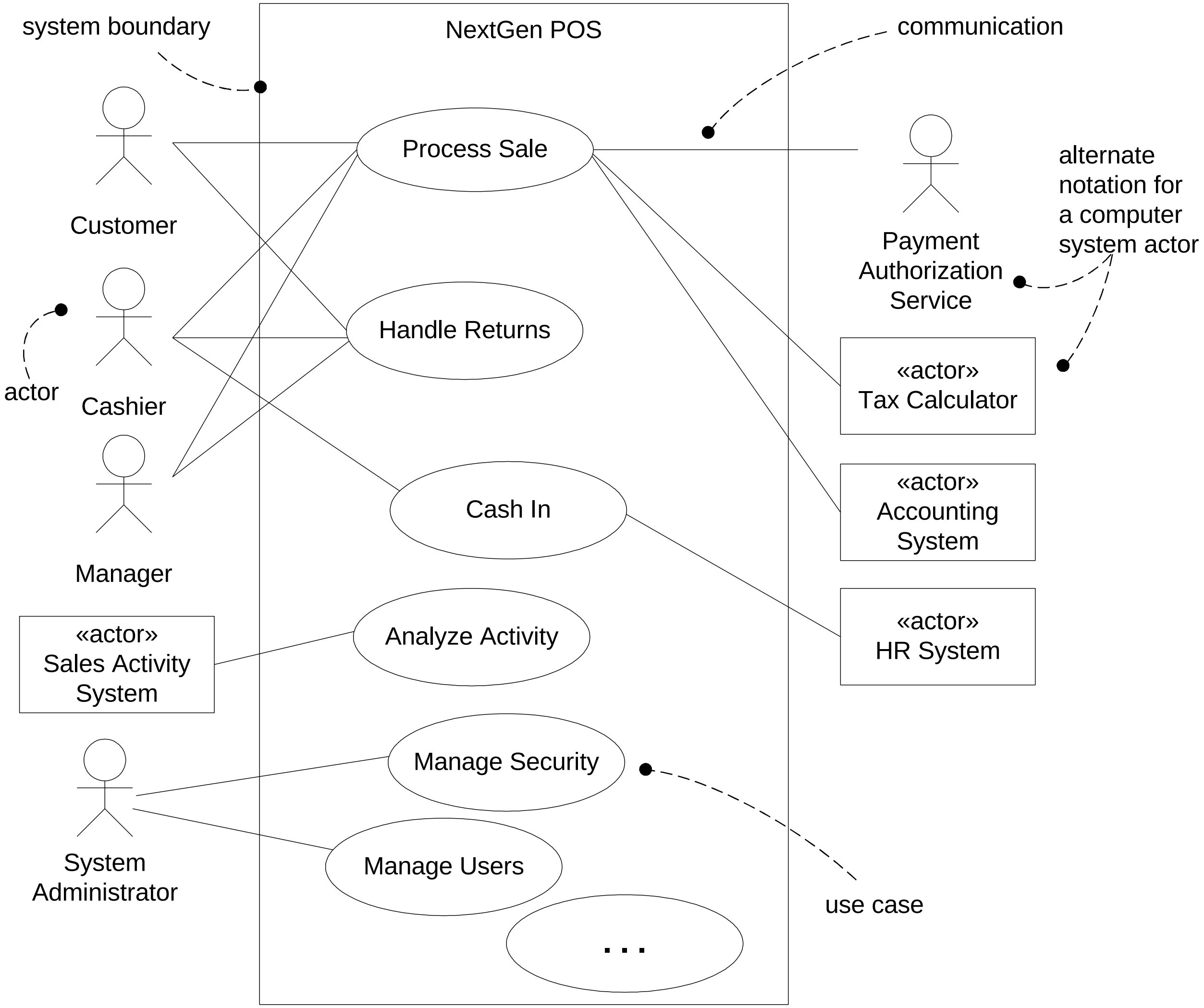
UC7: Process Rental

…

Extensions:

6b. Paying by credit: *Handle Credit Payment.*

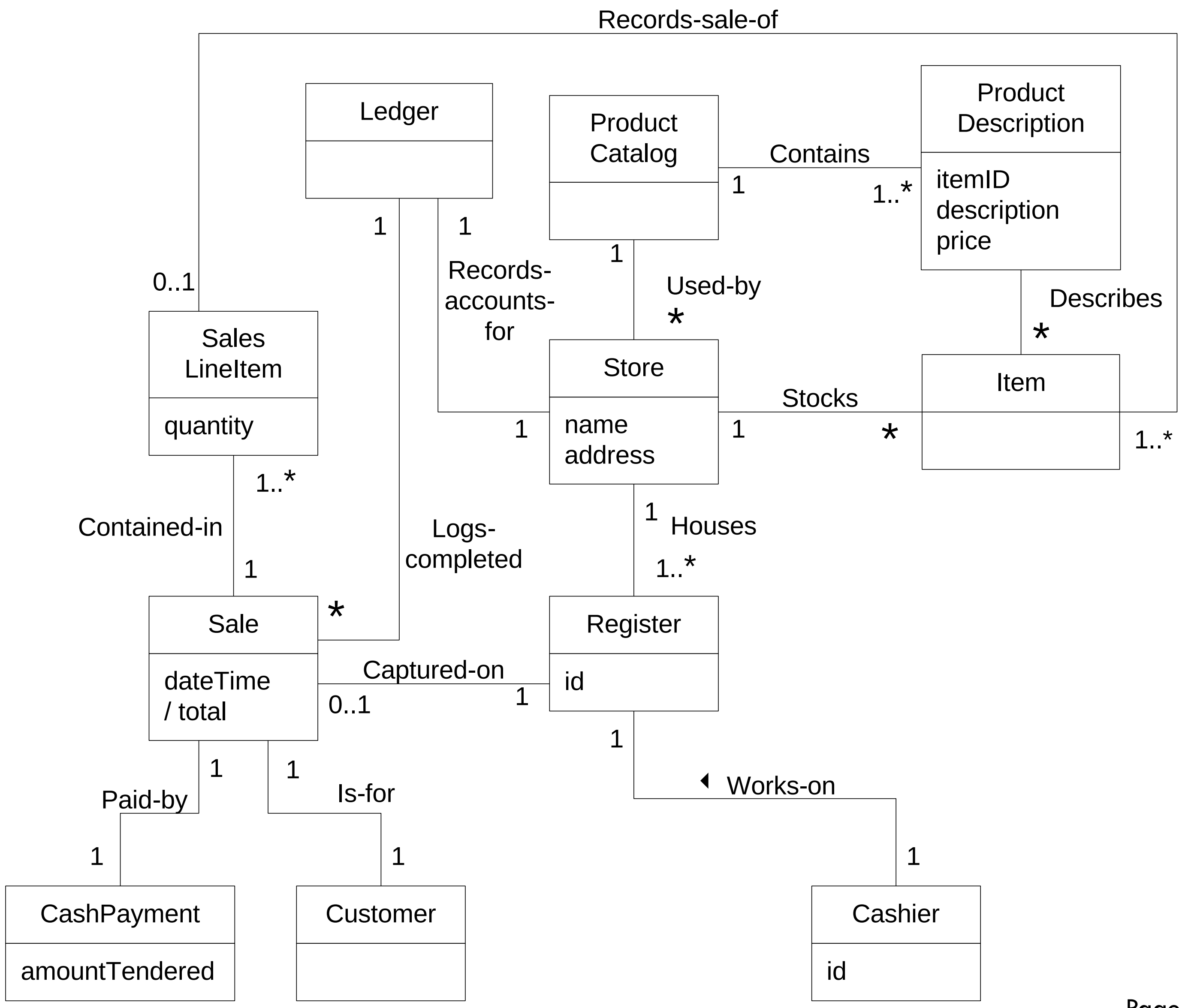
…



NextGen POS Analysis: Domain Model

A conceptual perspective model
Partial domain model drawn with UML class diagram

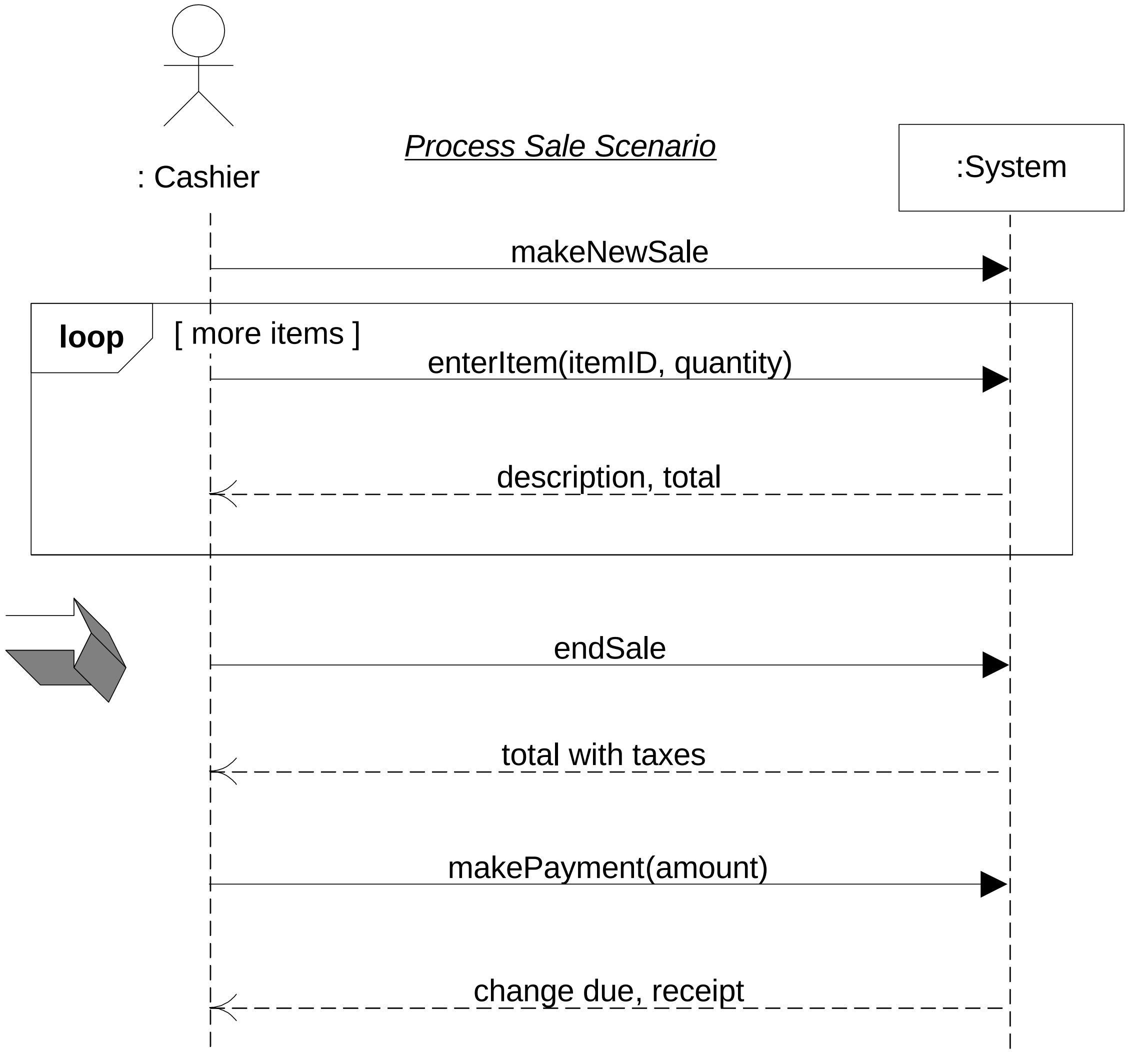
It shows conceptual classes with key associations



NextGen POS Analysis: System Sequence Diagram (SSD)

<u>Simple cash-only *Process Sale* scenario:</u>

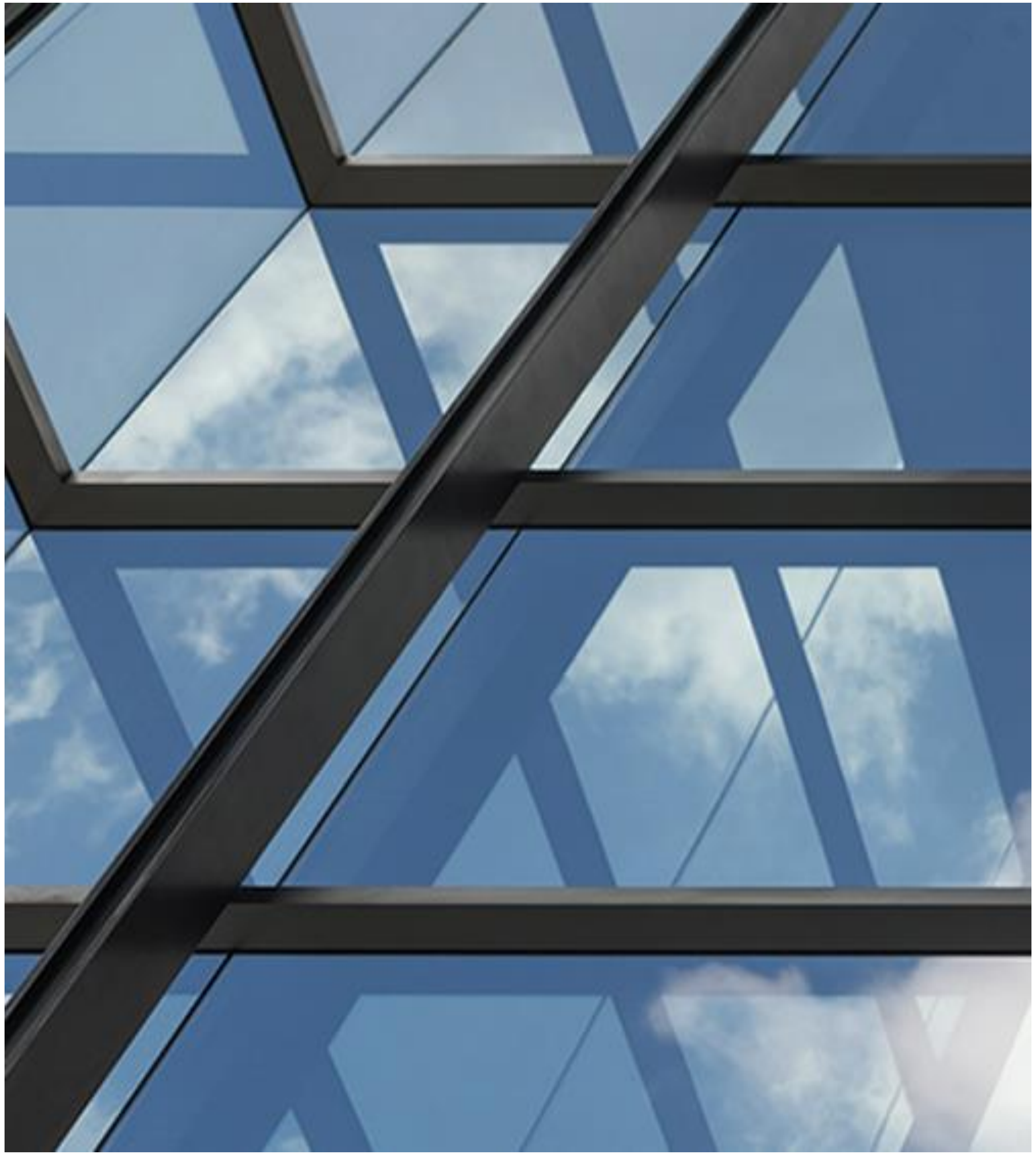
1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
...



NextGen POS Case Study: Design

OO Design with UML

THE UNIVERSITY OF SYDNEY



Next Gen POS:
From Analysis to Design

Requirements Analysis (OOA)

Business modelling – domain models

Use case diagrams

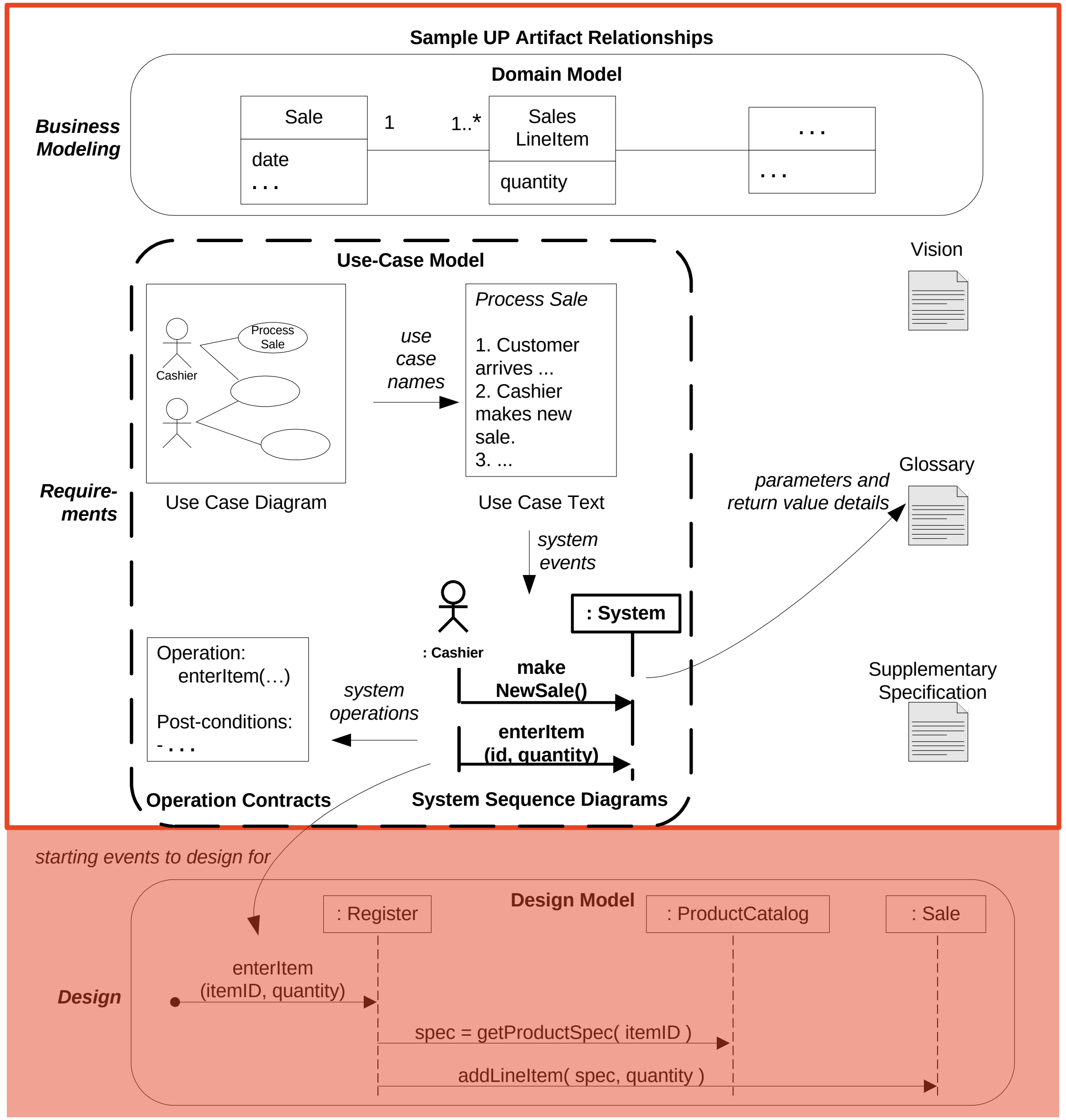
Use case description

System Sequence Diagrams

Design (OOD)

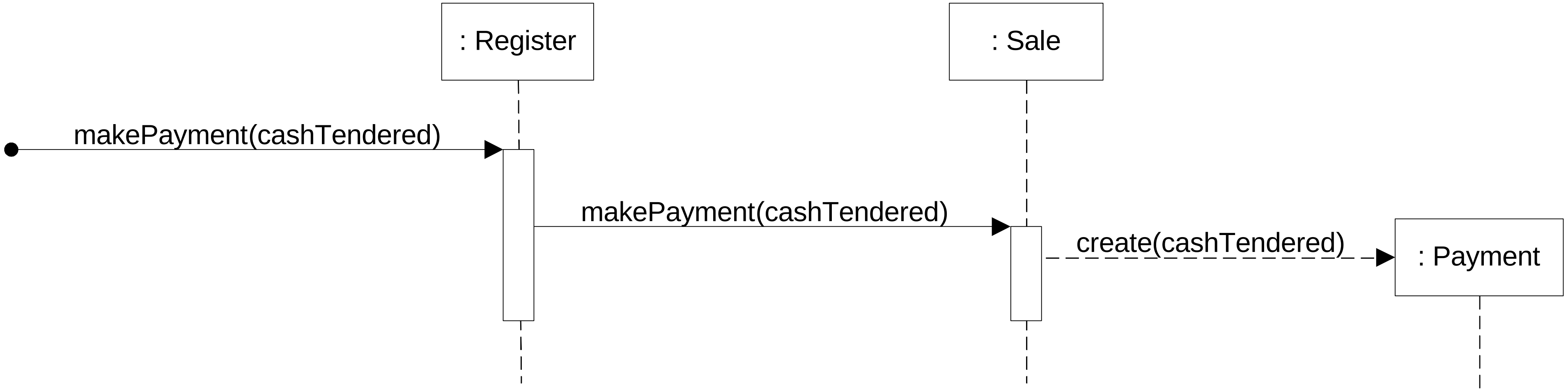
Sequence diagrams

Class diagrams

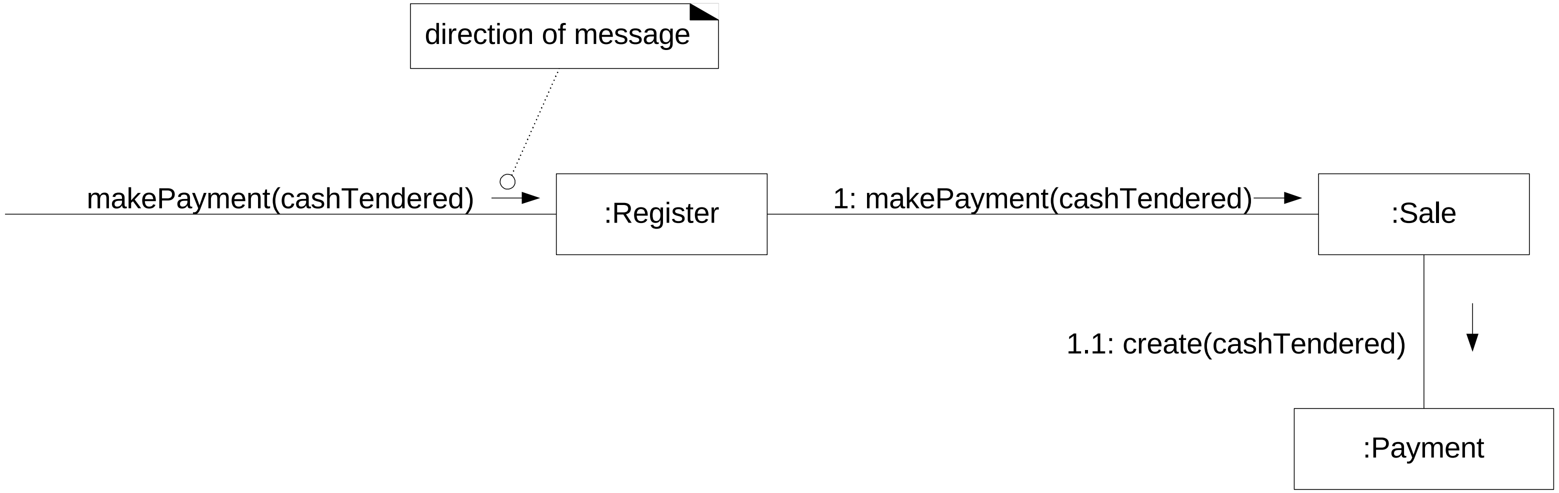


NextGen POS: Interaction Diagrams

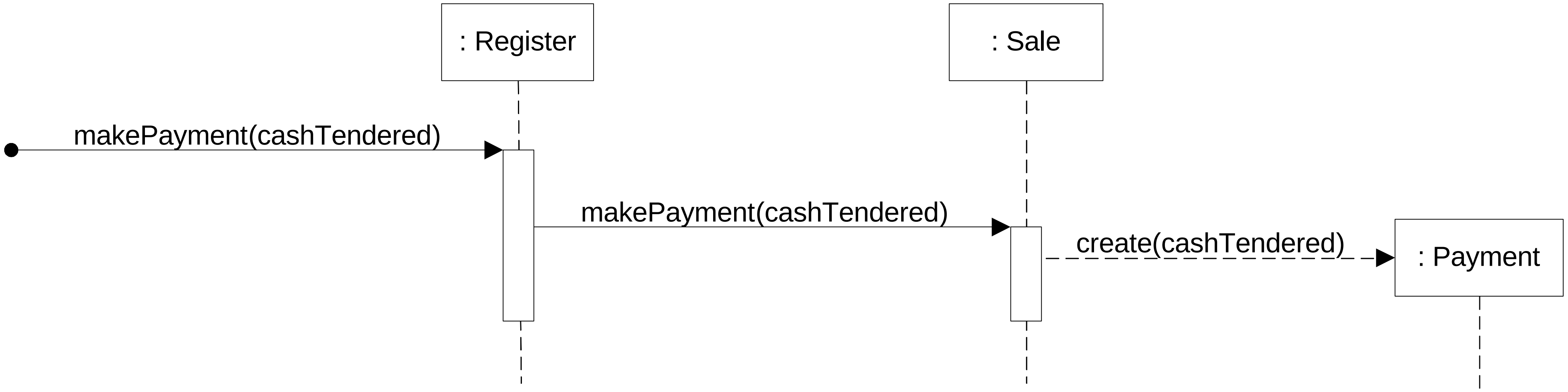
Sequence Diagram



Communication Diagram


direction of message
makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
:Sale
1.1: create(cashTendered)
:Payment

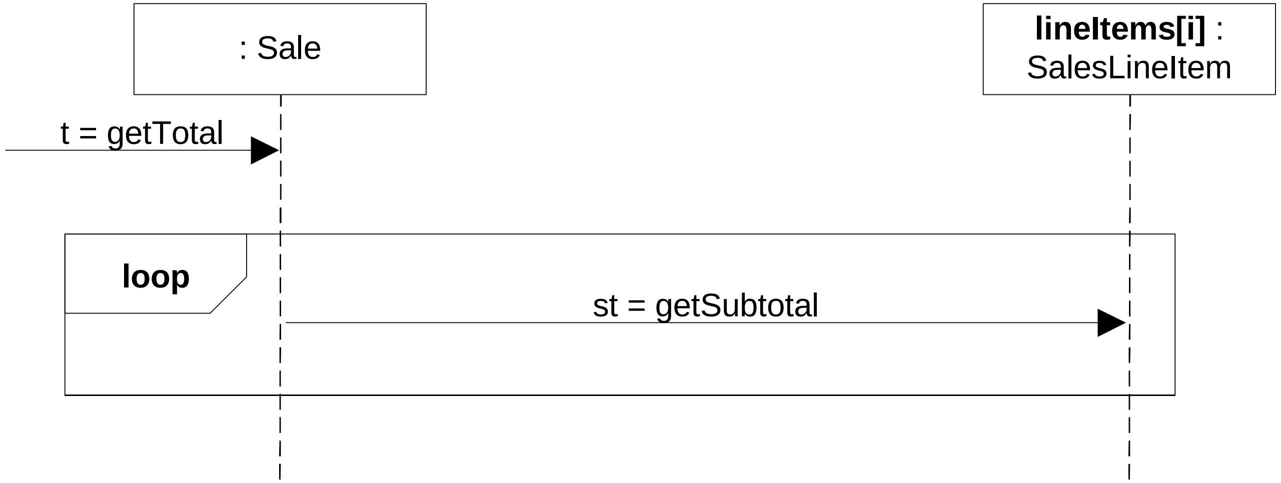
NextGen POS: Sequence Diagrams

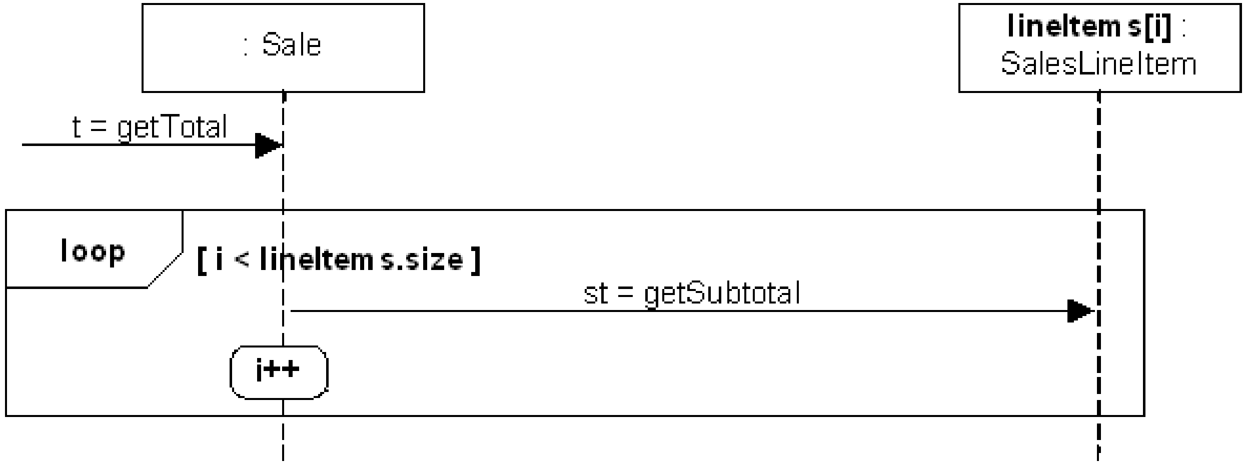


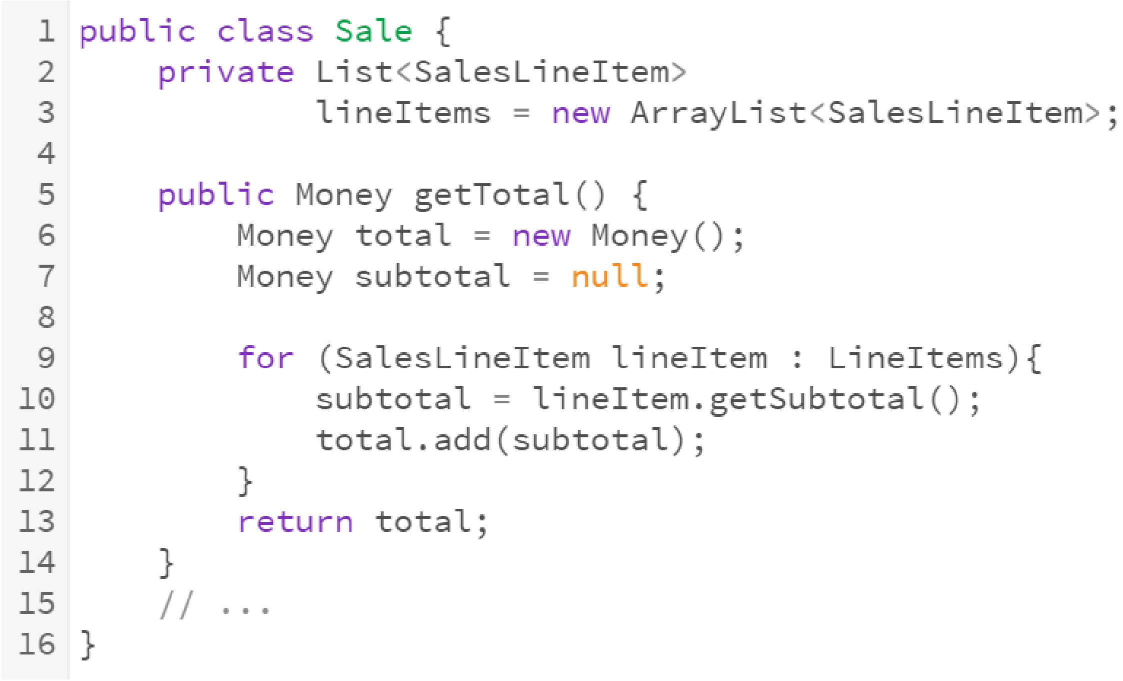
1. The message *makePayment* is sent to an instance of a *Register*. The sender is not identified
2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
3. The *Sale* instance creates an instance of a *Payment*.

How the skeleton of the Sale class should look like?

NextGen POS: Sequence Diagram (Iteration)





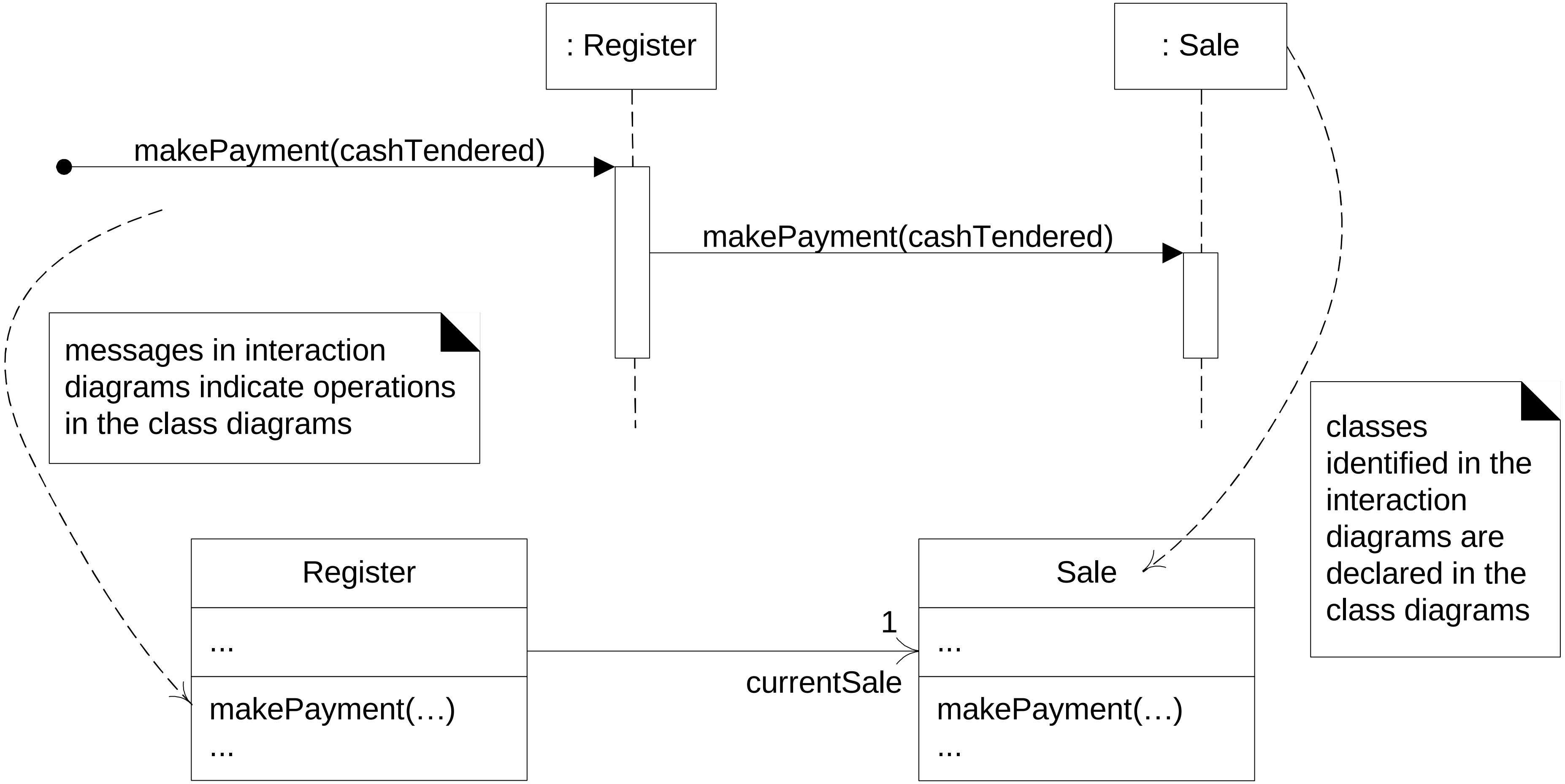


```java
public class Sale {
    private List<SalesLineItem>
            lineItems = new ArrayList<SalesLineItem>;

    public Money getTotal() {
        Money total = new Money();
        Money subtotal = null;

        for (SalesLineItem lineItem : LineItems){
            subtotal = lineItem.getSubtotal();
            total.add(subtotal);
        }
        return total;
    }
    // ...
}
```

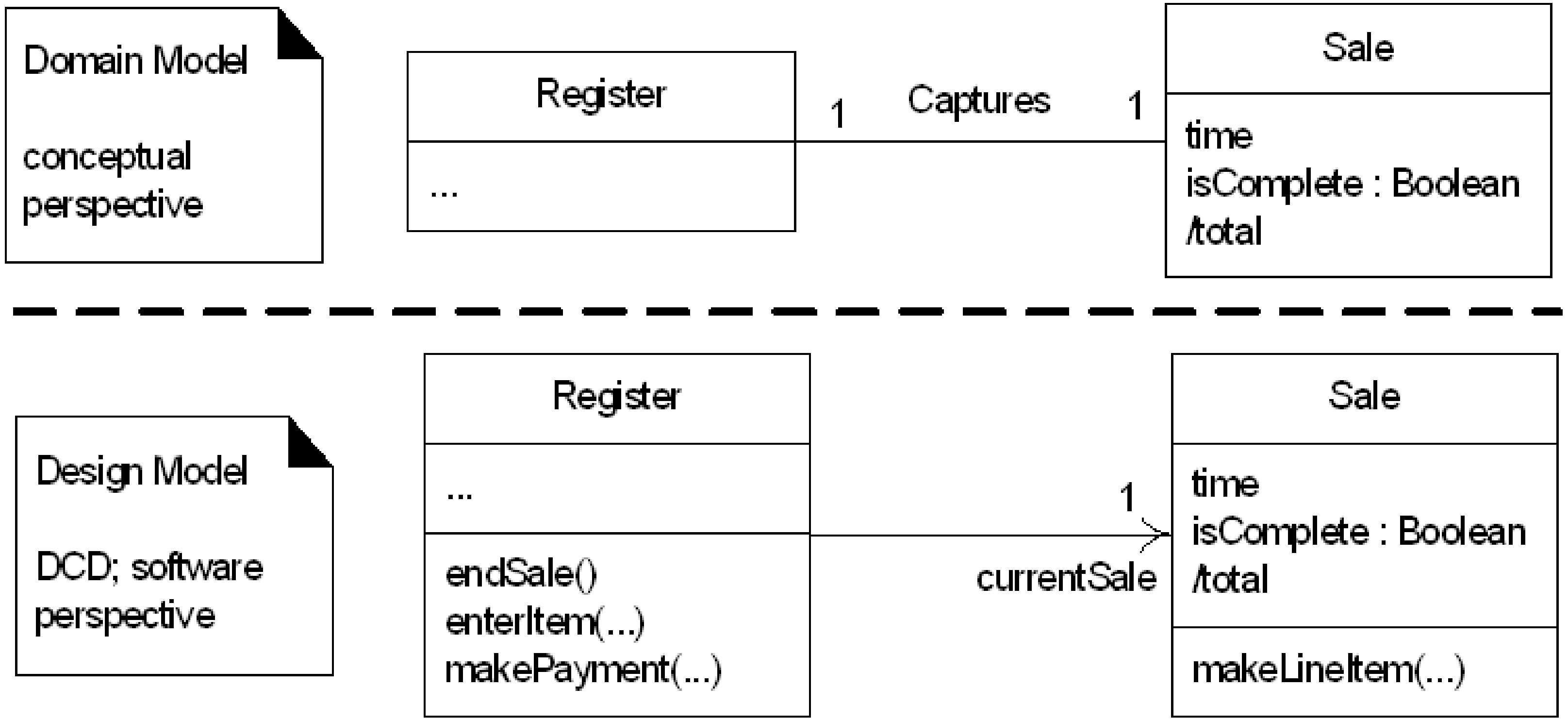
NextGen POS: Interaction and Class Diagrams



messages in interaction diagrams indicate operations in the class diagrams

classes identified in the interaction diagrams are declared in the class diagrams

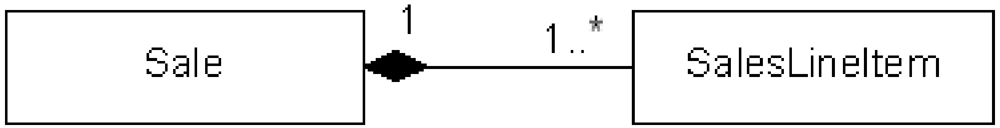
NextGen POS: Design Class Diagram



Next Gen POS: Composite Aggregation

SalesLineItem instance can only be part of one composite (*Sale*) at a time.

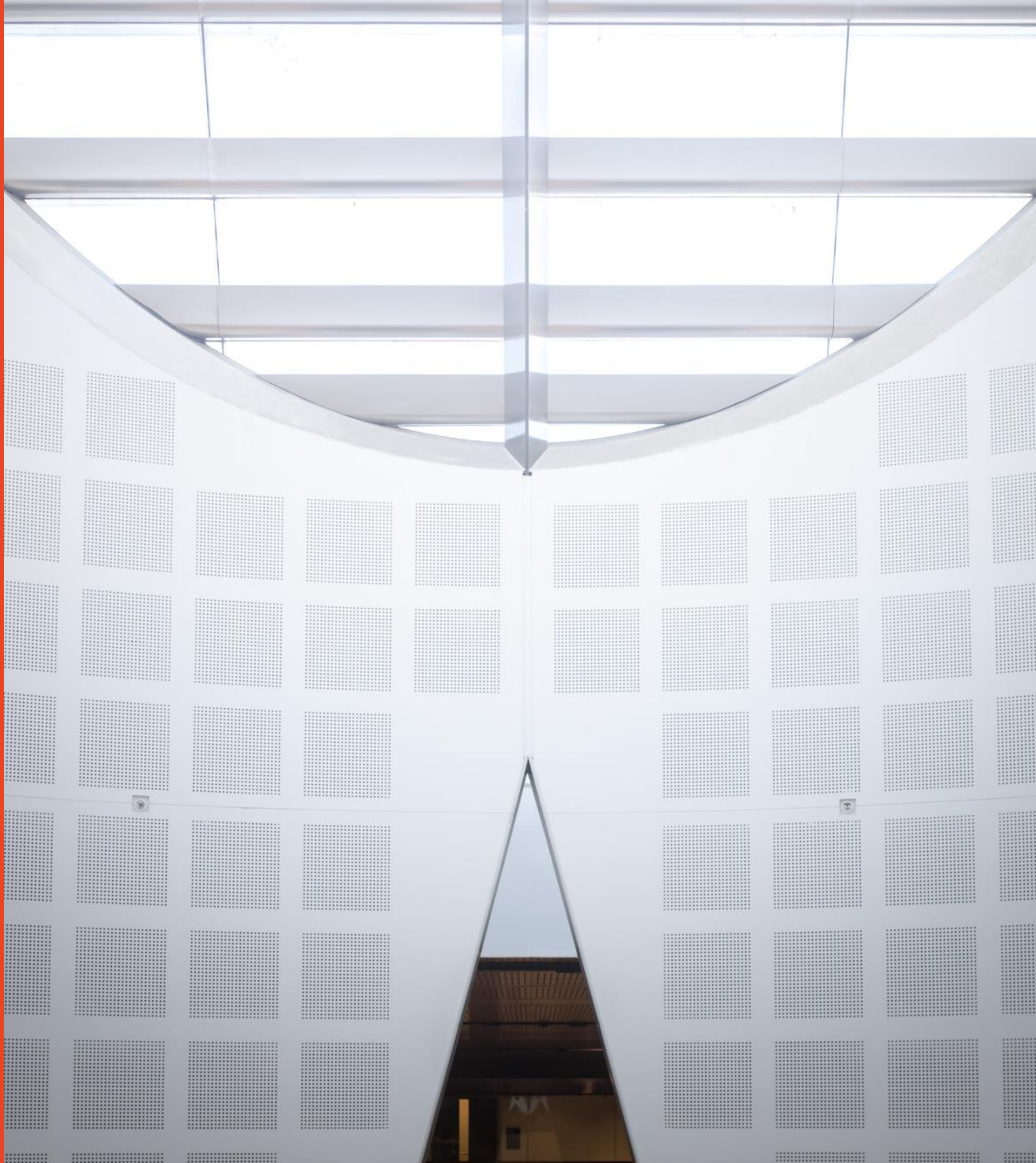
The composite has sole responsibility for management of its parts, especially creation and deletion



Composition (or Composite Aggregation is a strong kind of whole-part aggregation.
Use composition over aggregation as the latter was deemed by UML creators as "*Placebo*"

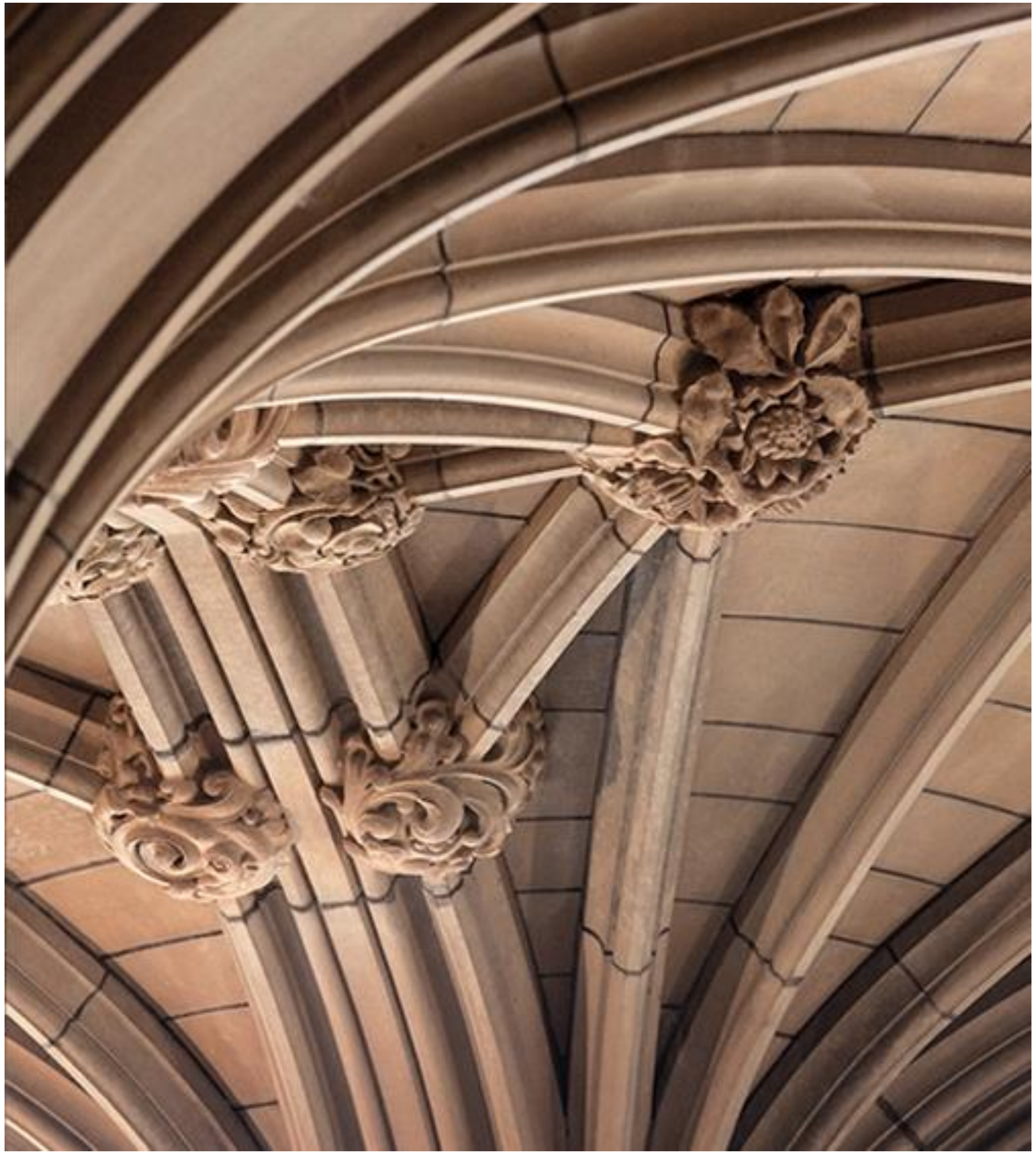
GoF Design Patterns

Review



Decorator Pattern

Object Structural (Wrapper)



THE UNIVERSITY OF SYDNEY

Decorator Pattern

- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality

- **Applicability**
  - to add responsibilities to individual objects dynamically and transparently, without affecting other objects
  - For responsibilities that can be withdrawn
  - When extension by sub-classing is impractical
    - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition maybe hidden or otherwise unavailable for sub-classing

Decorator Pattern – Why Not Inheritance?

- We want to add responsibilities to individual objects, not an entire class
  - E.g., A GUI toolkit should let you add properties like borders or behaviors like scrolling to any user interface component

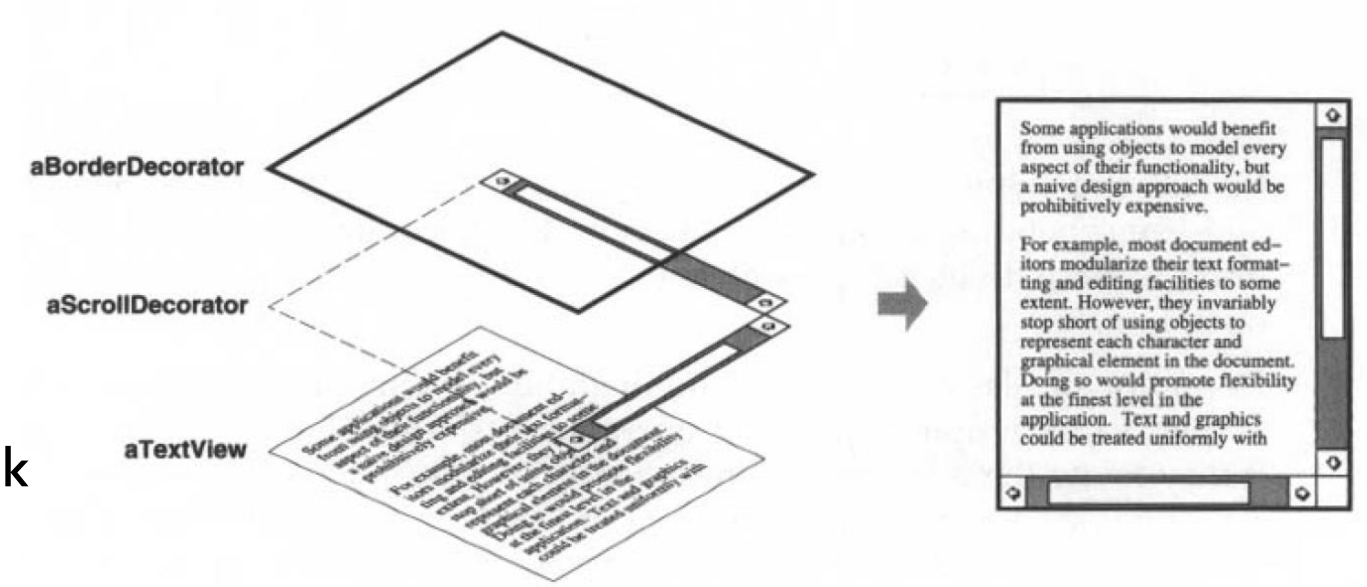
- Is adding responsibilities using inheritance a good design? For example, inheriting a border class puts a border every subclass instance
  - Why, why not?

Decorator Pattern – Why Not Inheritance?

- Why not adding responsibilities using inheritance is not a good design? For example inheriting a border class puts a border every subclass instance
  - This design is inflexible
    - The choice of border is made statically; a client cannot control how and when to decorate the component with a border
    - More flexible design is to enclose the component in another object that adds the border

Decorator Pattern – Text Viewer Example

- TextView object has no scroll bars and border by default (not always needed)
- ScrollDecorator to add them
- BorderDecorator to add a thick black border around the TexttView

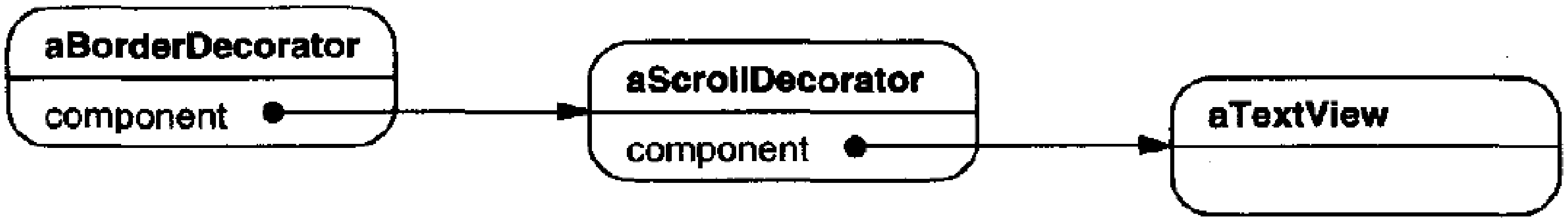


Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

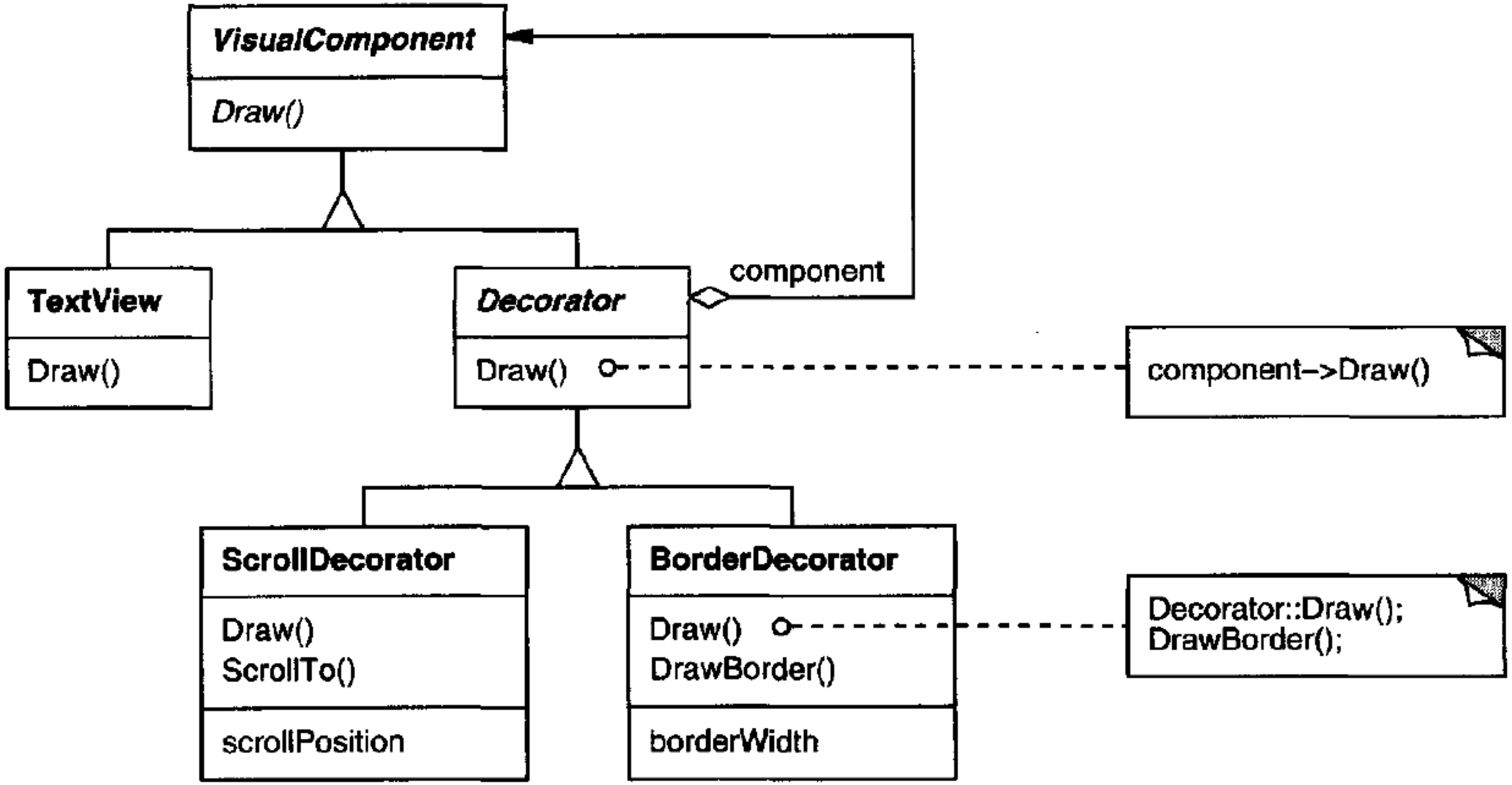
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Decorator Pattern – Text Viewer Example

– Compose the decorators with the TextView to produce both the border and the scroll behaviours for the TextView



Decorator Pattern – Text Viewer Example



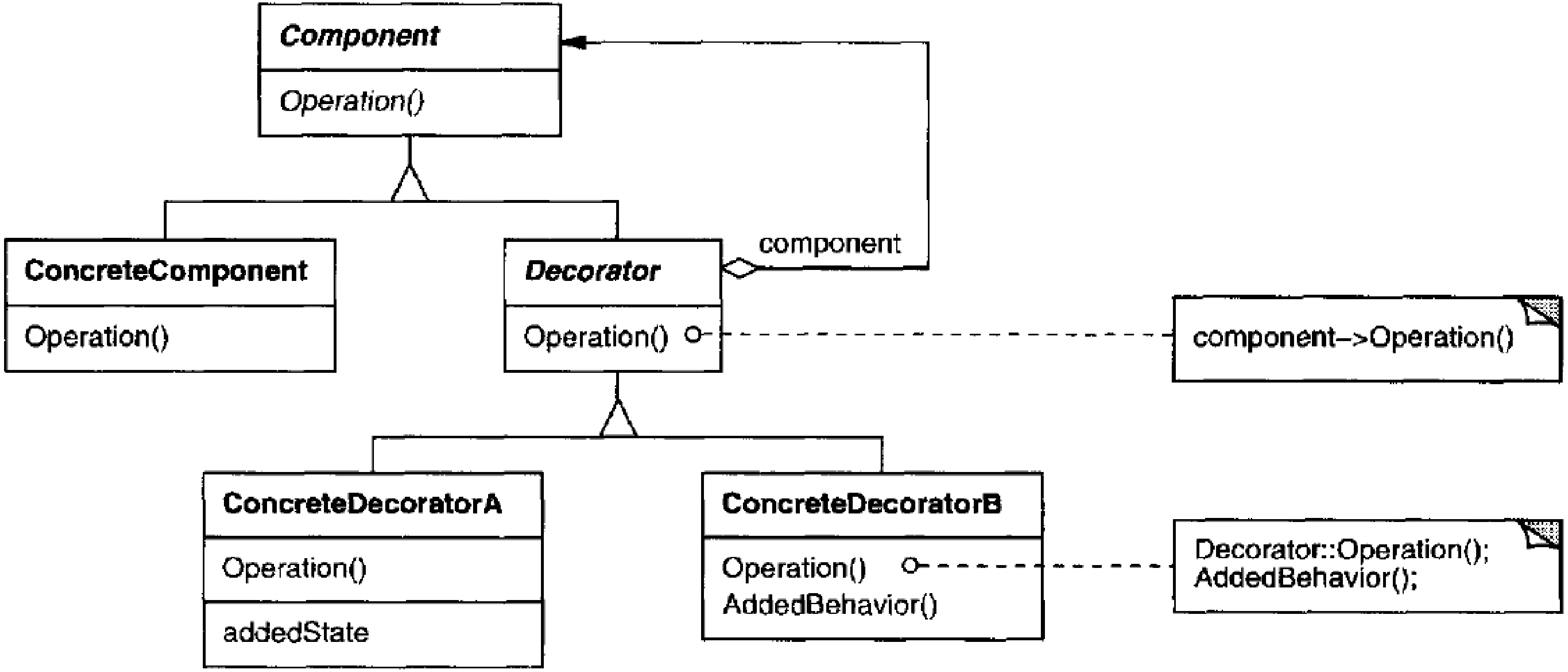
Decorator – Text Viewer Example

- VisualComponent is the abstract class for visual objects
  - It defines their drawing and event handling interface

- Decorator is an abstract class for visual components that decorate the other visual components
  - It simply forwards draw requests to its component; Decorator subclasses can extend this operation

- The ScrollDecorator and BorderDecorator classes are subclasses of Decorator
  - Can add operations for specific functionality (e.g., ScrollTo)

Decorator – Structure



Decorator – Participants

- **Component** (*VisulaComponent*)
  - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent** *(TextView)*
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a Component object and defines an interface that
  - Conforms to Component's interface.
- **ConcreteDecorator** *(BorderDecorator, ScrollDecorator)*
  - Adds responsibilities to the component

Decorator – Collaborations

- **Collaborations**
  - Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences (1)

- More flexibility and less complexity than static inheritance
  - Can add and remove responsibilities to objects at run-time
  - Inheritance requires adding new class for each responsibility (increase complexity)

- Avoids feature-laden (heavily loaded) classes high up in the hierarchy
  - Defines a simple class and add functionality incrementally with Decorator objects – applications do not need to have un-needed features
  - You can define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions

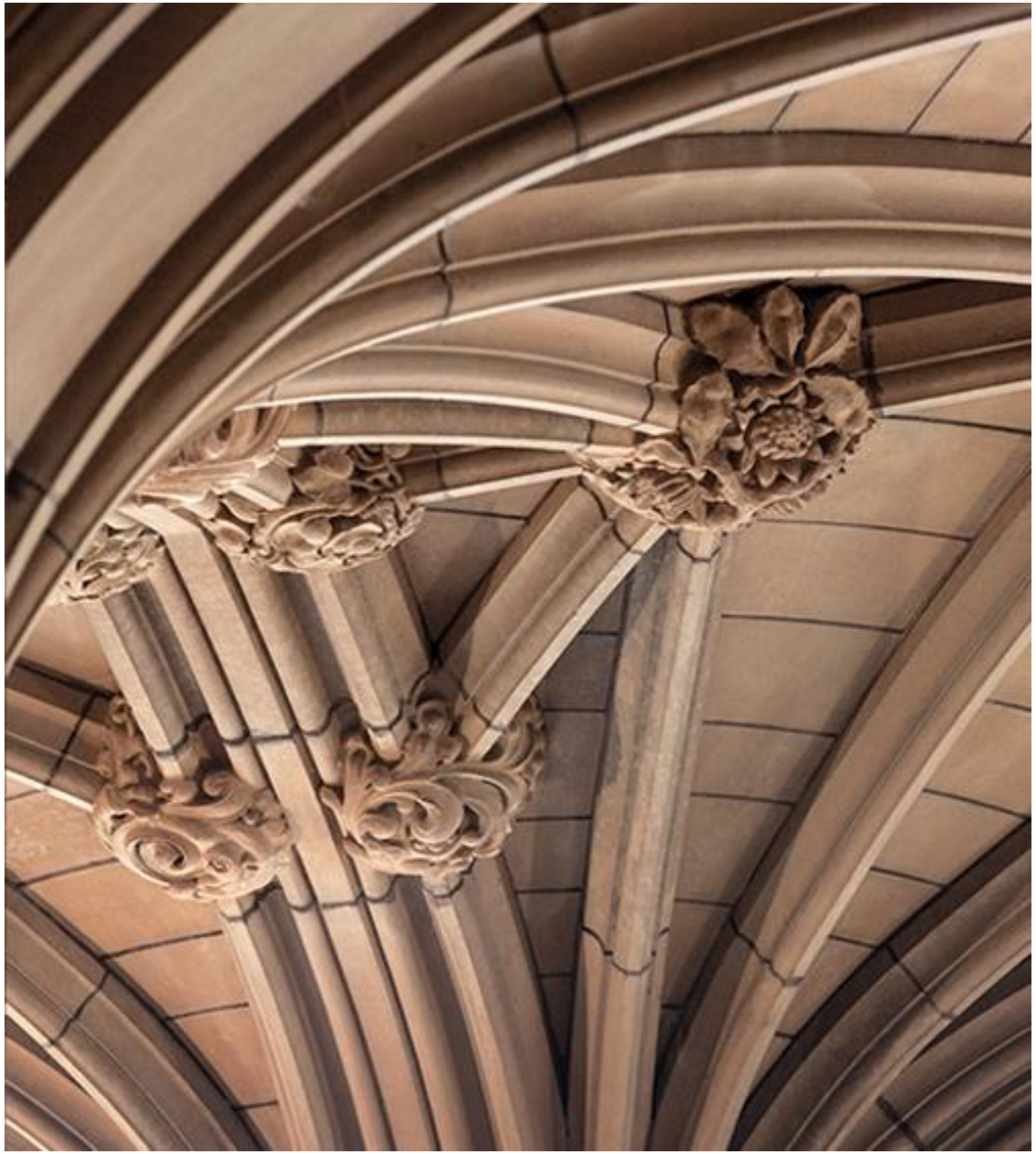
Consequences (2)

- Decorator and its component are not identical
  - Decorated component is not identical to the component itself - you shouldn't rely on object identity when you use decorator

- Many little objects
  - Can become hard to learn and debug when lots of little objects that look alike
  - Still not difficult to customize by those who understand them

Façade Pattern

Object Structural

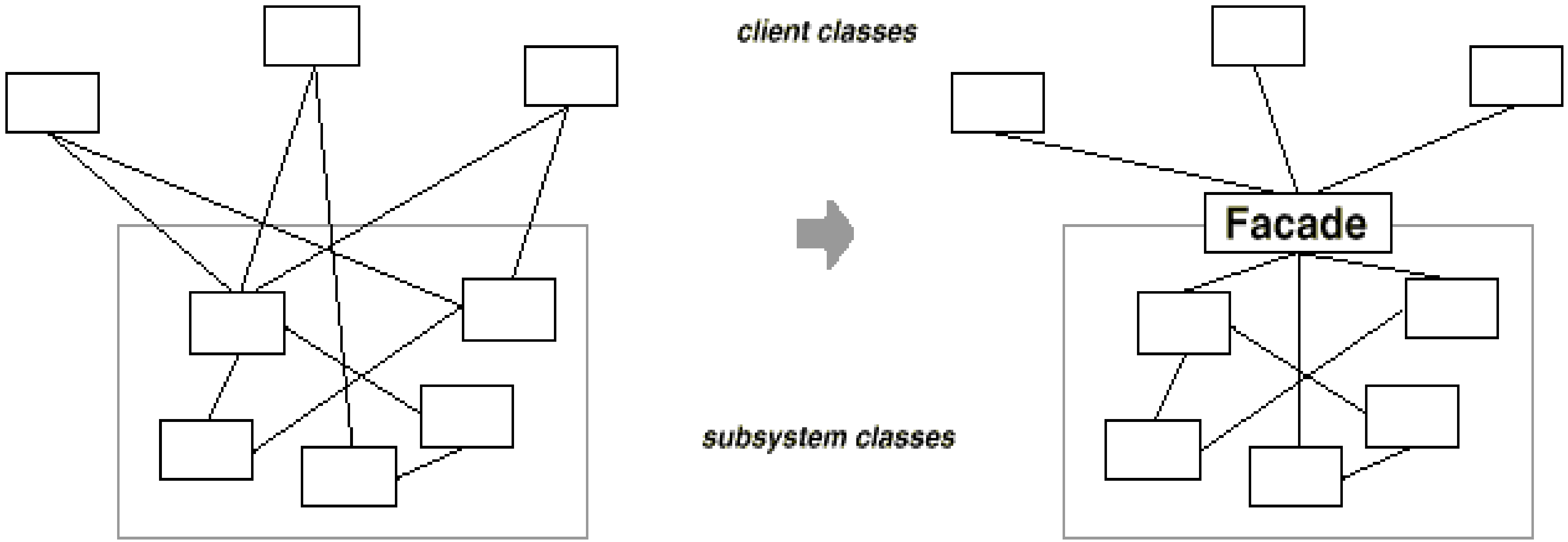


Façade Pattern

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use

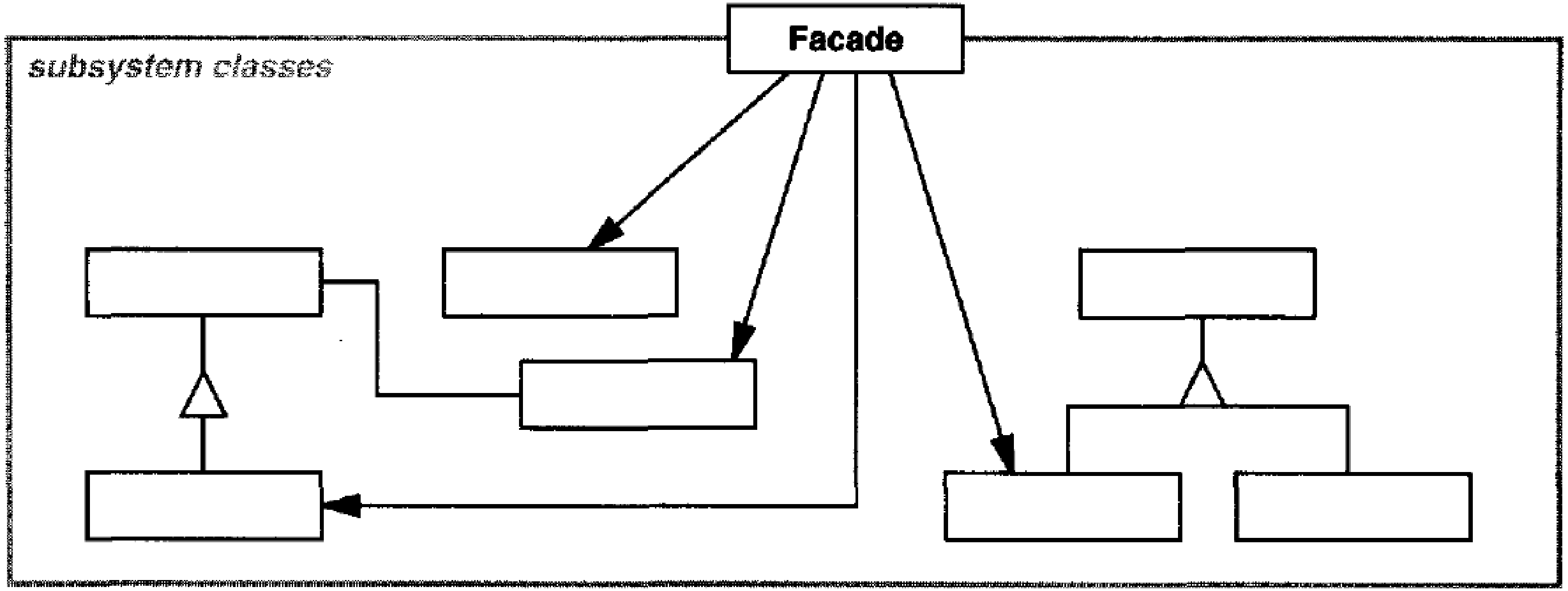
- Applicability
  - You want to provide a simple interface to a complex subsystem
  - There are many dependencies between clients and the implementation classes of an abstraction
  - You want to layer your subsystem. Façade would define an entry point to each subsystem level

Façade Motivation



A **facade** object provides a single, simplified interface to the more general facilities of a subsystem

Façade – Structure



Façade – Participants

- **Facade**
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
  - Implement subsystem functionality.
  - Handle work assigned by the Façade object
  - Have no knowledge of the facade; they keep no references to it.
- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).
    - Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces
  - Clients that use the facade don't have to access its subsystem objects directly

Consequences

- Simplify the usage of an existing subsystem by defining your own interface

- Shields clients from subsystem components, reduce the number of objects that clients deal with and make the subsystem easier to use.

- Promote weak coupling between the subsystem and the clients
  - Vary the components of the subsystem without affecting its clients
  - Reduce compilation dependencies (esp. large systems) – when subsystem classes change

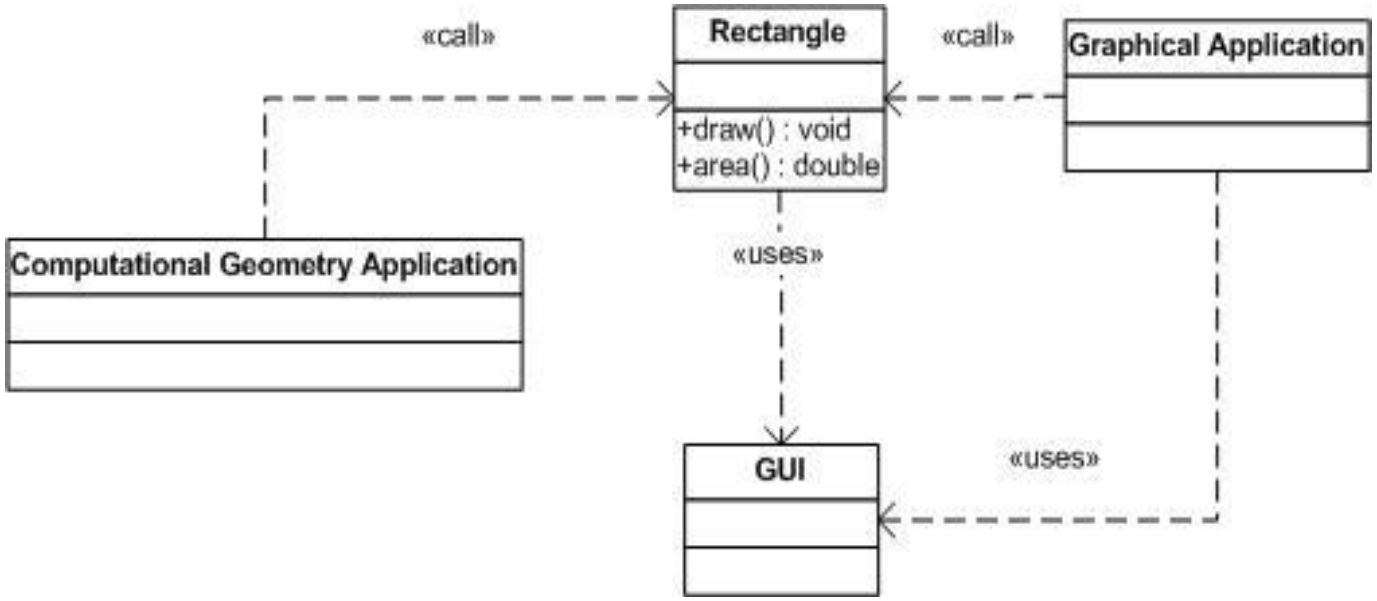
- Does not prevent applications from using subsystem classes if they need to. Choice between ease of use and flexibility.

Single-Responsibility Principle

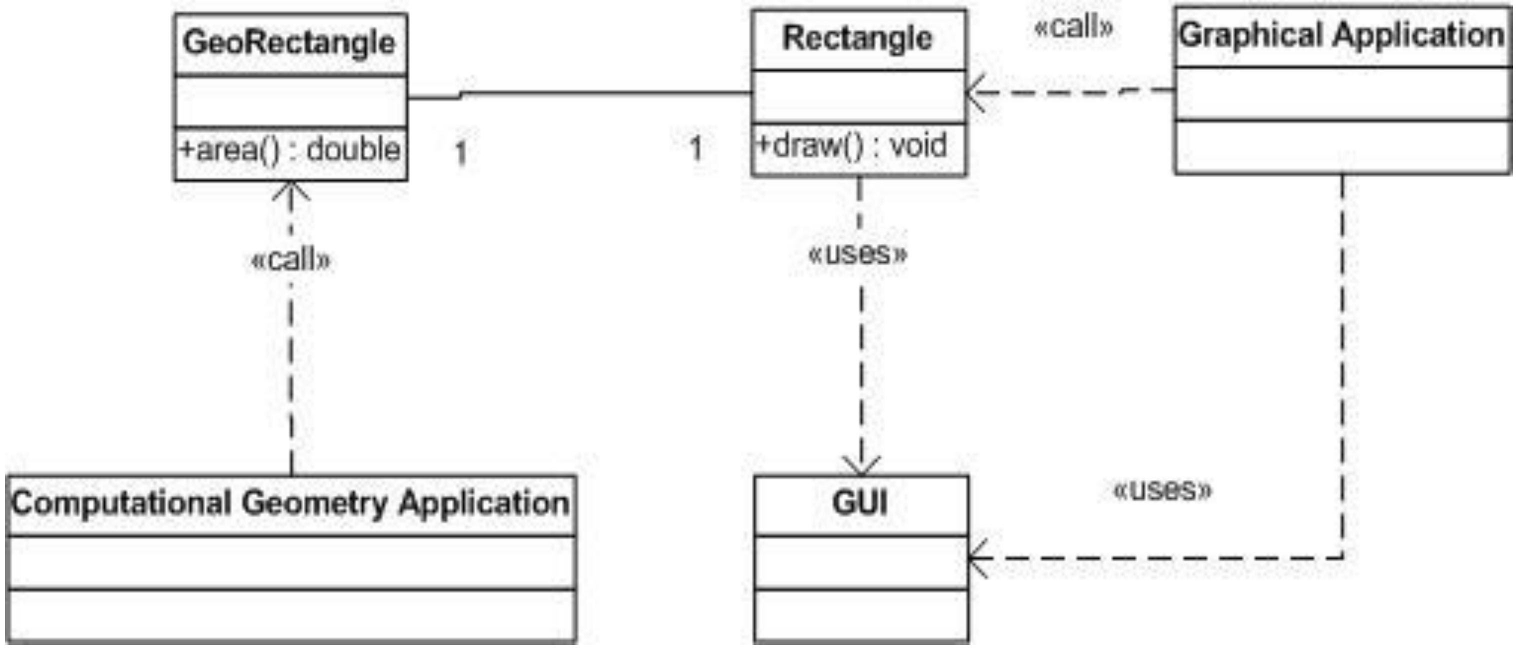
- A class should have only one reason to change
  - Each responsibility of a class is an area of potential change
  - More than one responsibility means more than one area of change
  - This increases the probability the class will change in the future
  - And when it does, it’ll affect more than one aspects of your design

Single Responsibility: Example

- *Rectangle* class has two sets of responsibilities
- **Problems**
  - We must include the GUI in the *Computational Geometry application*
  - If we change the way of rendering the *rectangle* class, the change will force us to rebuild, retest and redeploy the *Computational Geometry Application*



Separate the two Sets of Responsibilities



Façade – NextGen POS

- *Pluggable business rules* in POS (iteration 2 requirements)

- Consider rules that might invalidate an action at certain point
  - <u>When a new sales is created</u>:
    - Business rule 1: if it will be paid by a gift card, only one item is allowed to be purchased. Invalidate all requests of entering another item.

  - <u>When a payment is made by gift certificate:</u>
    - Business rule 2: balance should due back in another gift certificate. Invalidate all requests of giving customer change either in cash or credit card.

Façade - NextGen POS Business Rules

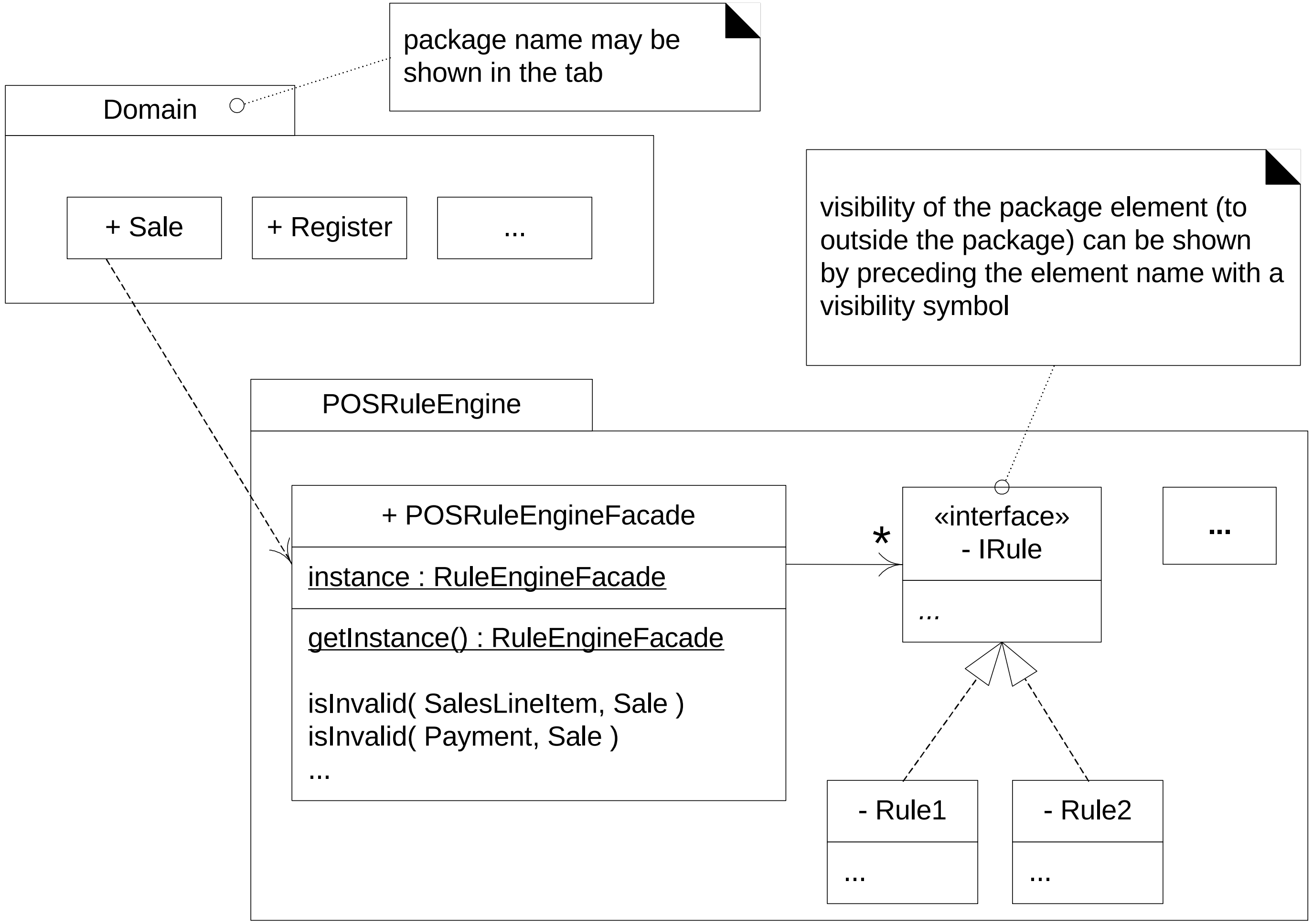
- Each store that would deploy PoS system will have different business rules implemented differently
  - Not desirable to scatter the implementation of business rules all over the system
    - Consequence: for each installation of POS system, need to modify lots of classes
  - We want a design that has low impact on the existing software components.
    - Factor out all rules in a separate subsystem to localize the change.

Façade in POS system

- Use an façade object **POSRuleEngineFacade** to communicate with the business rule subsystem
- Façade object is usually implemented as Singleton.

```
public class sale{
   public void makeLineItem(ProductDescription desc, int quantity){
    SalesLineItem sli = new SalesLineItem (desc, quantity);
    // call to the Façade
    if (POSRuleEngineFacade.getInstance().isInvalid(sli, this))
        return;
    lineItems.add(sli);
   }
   //..
}.. End of Sale
```

Façade – NextGen POS


package name may be shown in the tab
Domain
+ Sale
+ Register
...
visibility of the package element (to outside the package) can be shown by preceding the element name with a visibility symbol
POSRuleEngine
+ POSRuleEngineFacade
instance : RuleEngineFacade
getInstance() : RuleEngineFacade
isInvalid(SalesLineItem, Sale)
isInvalid(Payment, Sale)
...
*
«interface»
- IRule
...
...
- Rule1
...
- Rule2
...

Façade – NextGen POS

- Façade is always used to separate different tiers of a system
  - The Façade controller acts as the single point of entry from UI (presentation) layer to Domain layer
  - We also use Façade to control the communication between domain layer and data layer.

Façade – JDBC Example

- JDBC involves many classes
- Lots of client codes use those classes in a similar way
- Provide a façade to encapsulate some popular JDBC objects
- Save some programming work when dealing with common database tasks

--Example from Thomas E. Davis “Clever Façade makes JDBC look easy”, *JavaWorld*, May 1999
Ryan Daigle, “Eliminate JDBC overhead”, *JavaWorld*, May 2002

General JDBC programming

- Steps:
  - Load Driver
  - Get a connection (`java.sql.Connection`)
  - Form a SQL string
  - Get a prepared statement (`java.sql.PreparedStatement`)
  - Populate the prepared statement with values
  - Execute the statement
  - Iterate through the result set and form the result objects (`java.sql.ResultSet`)

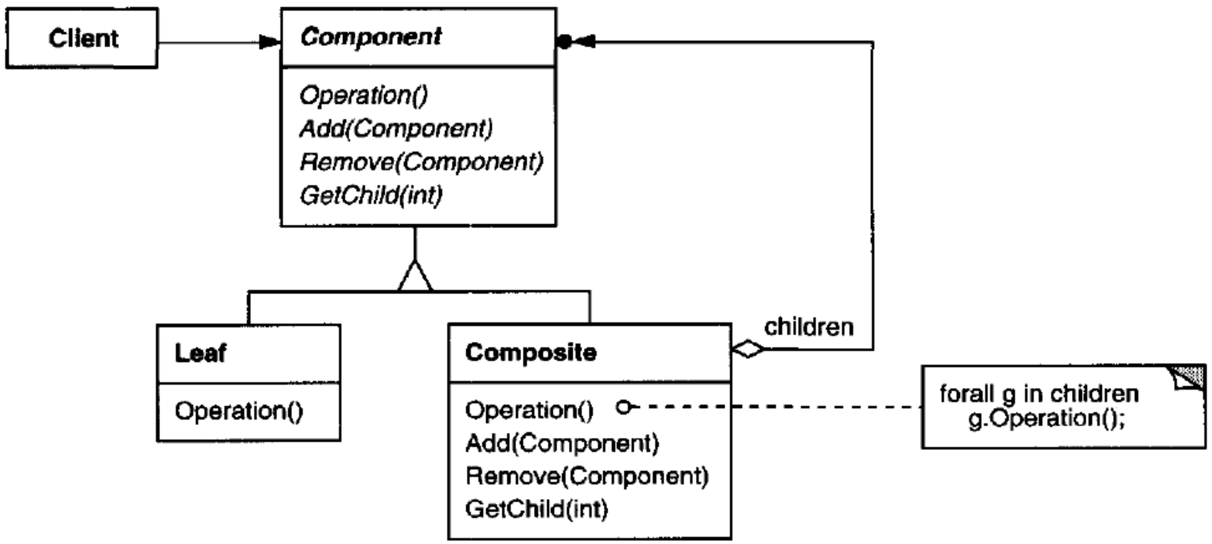
Problems & solution

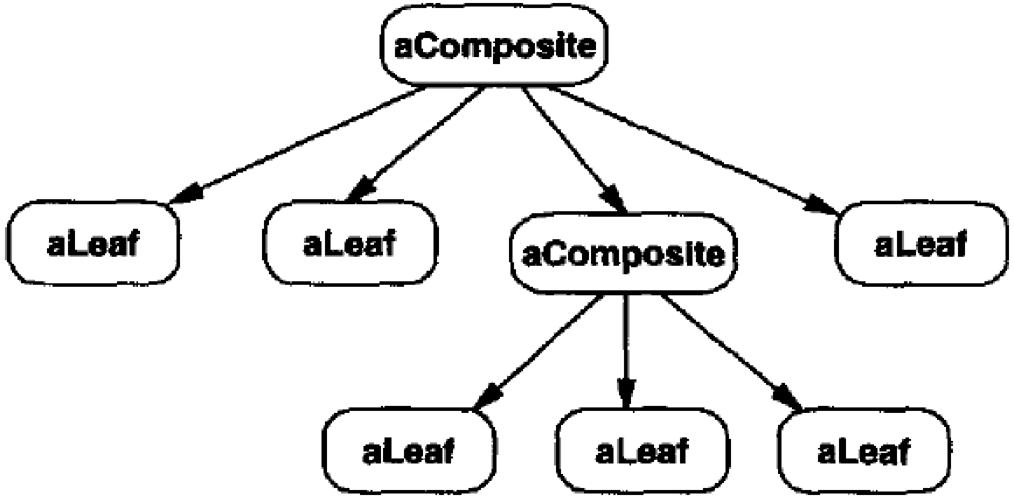
- May end up with lots of repetitive code if many objects need to communicate with database
  - Set up connection, exception handling,…
- Solution
  - Encapsulate common tasks in a set of utility classes – façade
  - Façade can be a class or a set of classes

Composite

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies.
  - Lets clients treat individual objects and compositions of objects uniformly
- Applicability
  - To represent part-whole hierarchies of objects
  - Clients want to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

Composite – Structure





A typical composite object structure

Adapter – Participants (1)

- **Component**
  - Declares the interface for objects in the composition
  - Implements default behavior for the interface common to all classes, as appropriate
  - Declares an interface for accessing and managing its child components
  - Defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate (optional)
- **Leaf**
  - Represents leaf objects in the composition A. leaf has no children
  - Defines behavior for primitive objects in the composition.

Adapter – Participants (2)

- **Composite**
  - Defines behavior for components having children
  - Stores child components.
- **Client**
  - Manipulates objects in the composite on through the Component interface
- **Collaborations**
  - Clients use the component class interface to interact with objects in the composite structure
  - If the recipient is a Leaf, then the request is handled directly
  - If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding

Composite – Consequences (1)

- Allows recursive composition
  - Clients code can expect a primitive object as well as a composite one
- Simplifies the client code
  - Clients do not need to write case-statement-style functions over the classes that define the composition
- Allow adding new kind of components
  - Clients do not have to change for new component classes
- Makes it harder to restrict composite's components
  - Type system cannot enforce certain components types, use run-time checks instead

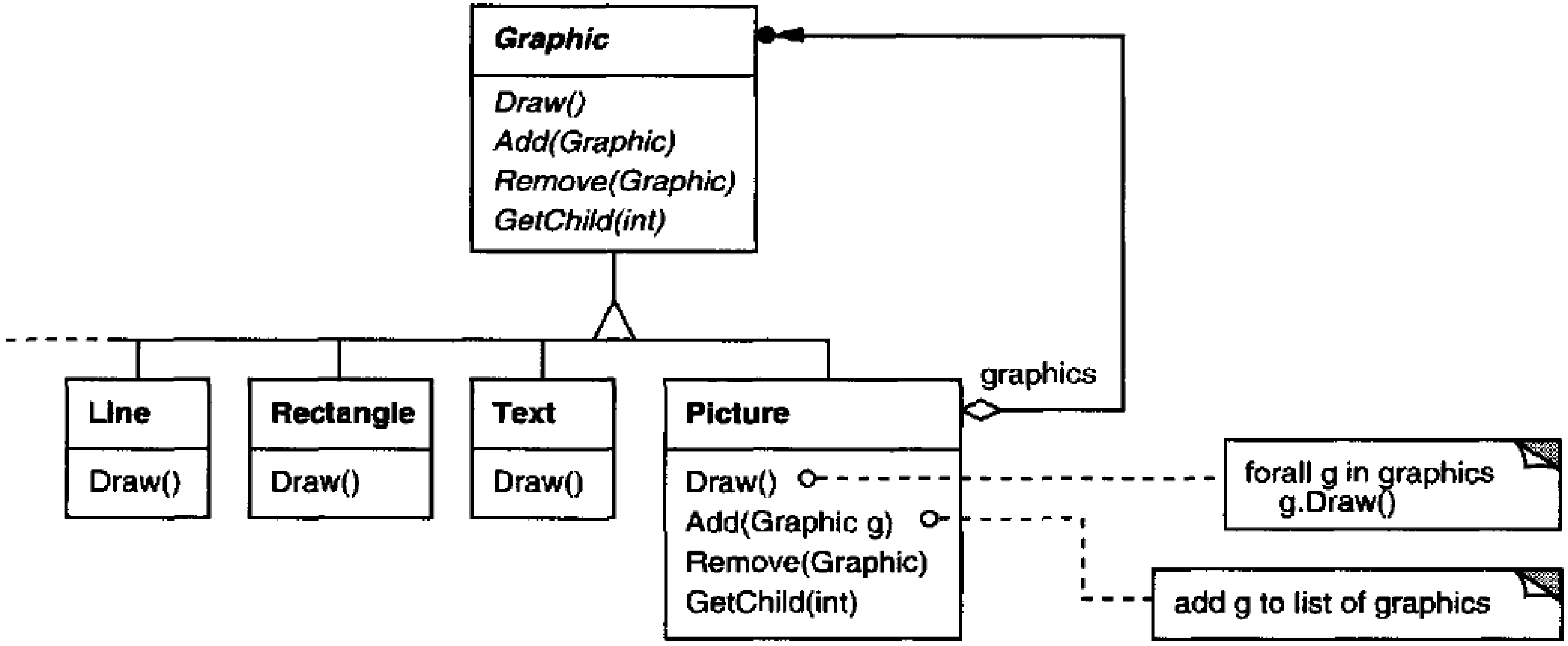
Composite – Graphic Editor Example

- Graphics applications let users build complex diagrams out of simple components

- This can be implemented as primitive classes for graphical primitives (e.g., Text and Lines and container classes for these primitives

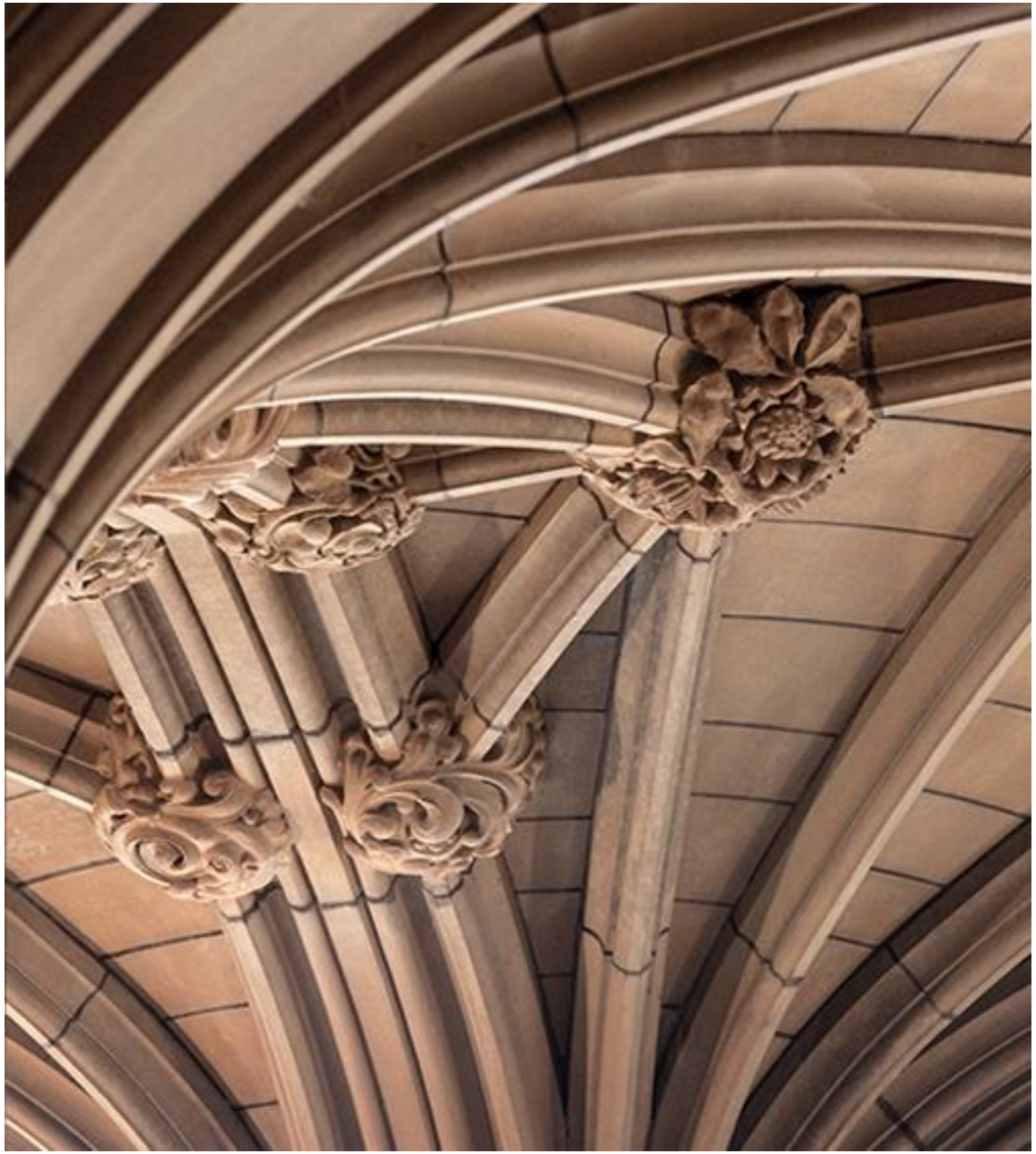
- Problem with this solution:
  - The client's code must distinguish between primitive and container objects (treat it differently) which will lead to a more complex application (client)

Composite – Graphic Editor Example



Builder

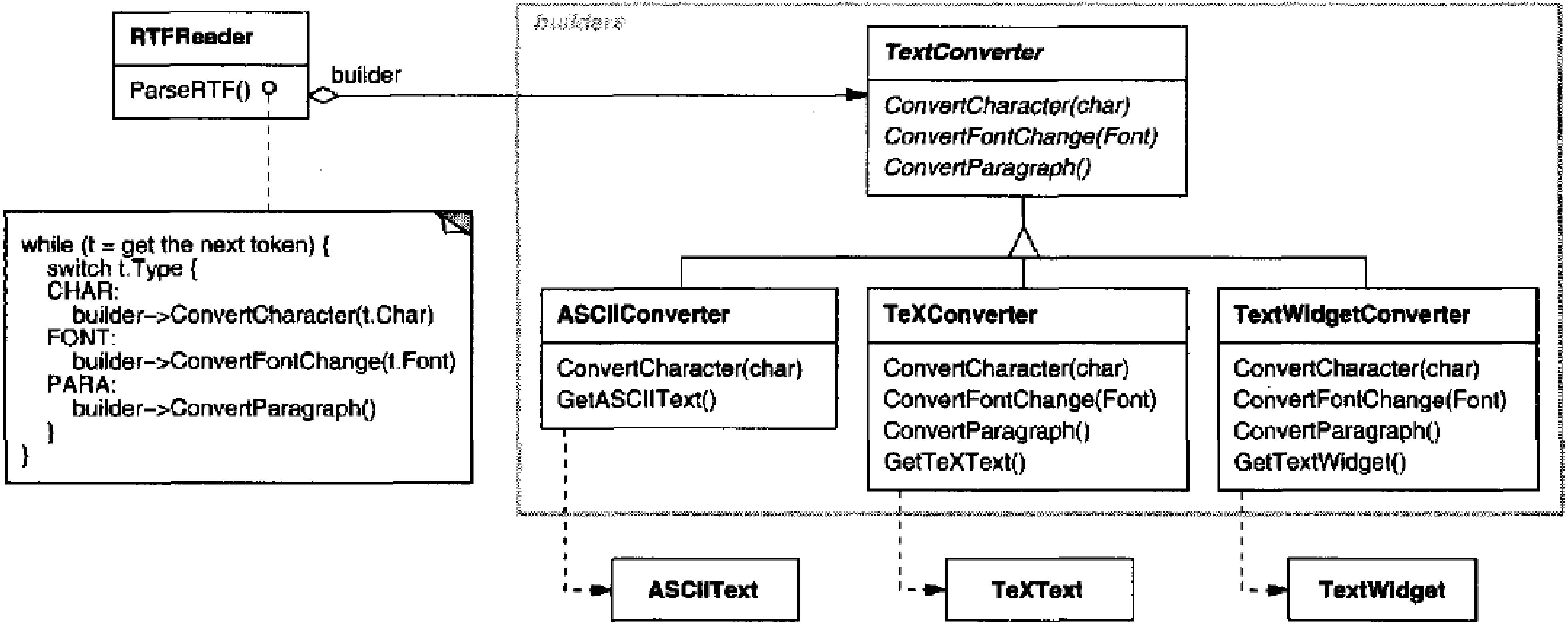
Object Creational



Builder

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Applicability
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - The construction process must allow different representations for the object that's constructed

Builder – Rich Text Format Example



```
while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(t.Char)
    FONT:
        builder->ConvertFontChange(t.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
```

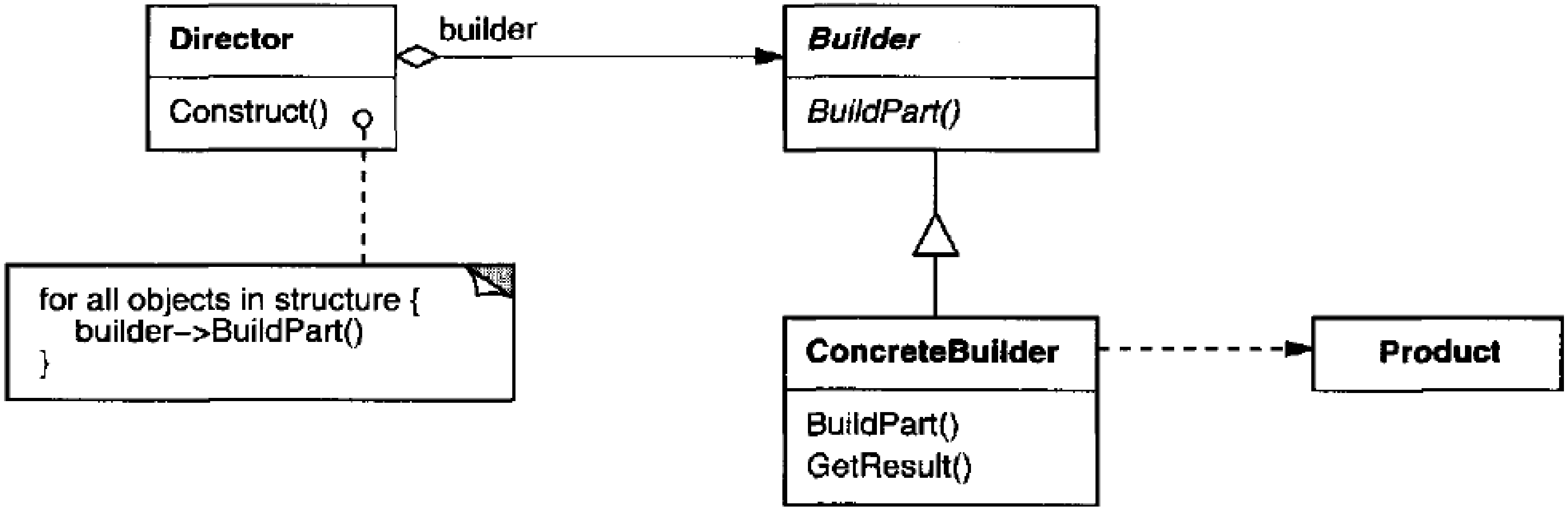
Builder – Rich Text Format Example

- A reader for the RTF (Rich Text Form at) document exchange e format should be able to convert RTF to many text formats
- Problem: the number of possible conversions is open-ended, so it should be easy to add a new conversion without modifying the reader
- A solution is to configure the *RTFReader* class with a *TextConverter* object that converts RTF to another textual representation
- Subclasses of TextConverter specialize in different conversions and formats
- Each kind of converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interfac

Builder – Rich Text Format Example

- Builder Pattern captures all the relationships
  - Each converter class is called a builder in the pattern, and the reader is called the director
  - It separates the algorithm for interpreting a textual format from how a converted format gets created and represented
  - The RTFReader's parsing algorithm can be reused to create different text representations from RTFdocuments – just configure the RTFReader with different subclasses of TextConverter

Builder – Structure



Builder – Participants (1)

- **Builder** (TextConverter)
  - Specifies an abstract interface for creating parts of a Product object

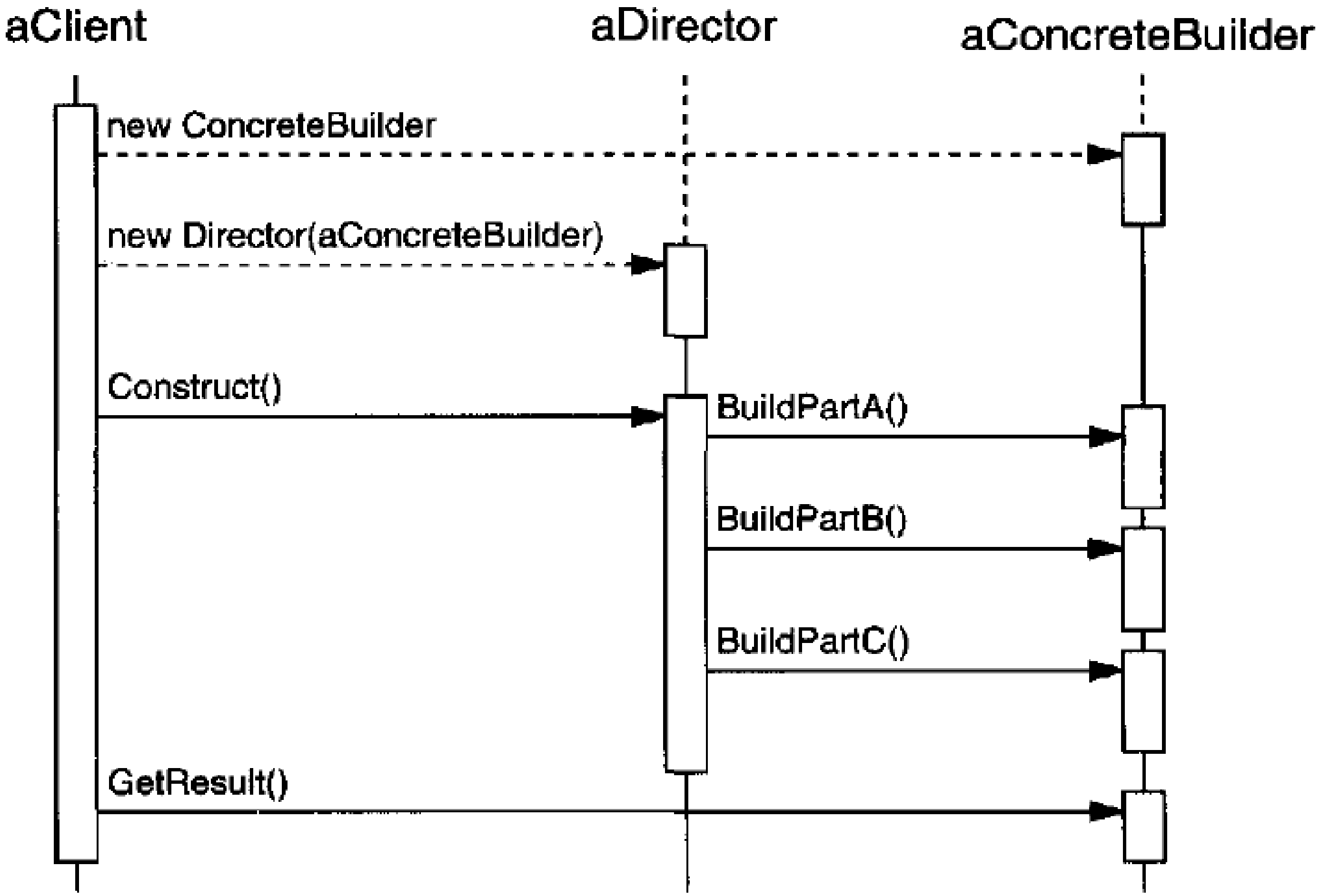
- **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter)
  - Constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget)

Builder – Participants (2)

- **Director** (RTFReader)
  - Constructs an object using the Builder interface

- **Product** (ASCIIText, TeXText, TextWidget)
  - Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Builder – Collaboration



Builder – Consequences (1)

- Varying product's internal representation
  - Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder

- Isolation of code construction and representation
  - Each ConcreteBuilder contains all the code to create and assemble a particular kind of product.
  - Different Directors can reuse it to build Product variants from the same set of parts
    - E.g., SGMLReader uses the same TextConverters to generate different formats (ASCIIText, TextWidget and TexXText)

Builder – Consequences (2)

- Finer control over the construction process
  - The builder pattern constructs the product step by step under the directors control
  - Only when the product finished does the director retrieve it from the builder
  - The builder interface reflects the process of constructing the product more than other creational patterns

Builder – Implementation (1)

- An abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default.

- A ConcreteBuilder class overrides operations for components it's interested in creating

Builder – Implementation Issues (1)

- Assembly and construction interface
  - In the RTFexample, the builder converts and appends the next token to the text it has converted so far, BUT sometimes you might need access to parts of the product constructed earlier
  - Example: Tree structures (e.g., parse tress) are built bottom-up. How this should be implemented?

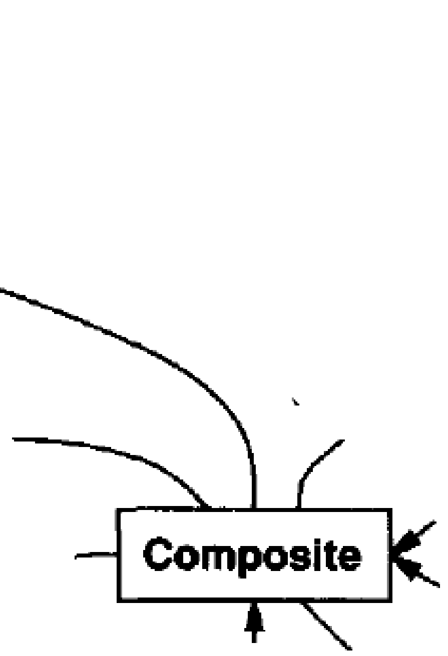
Builder – Other Implementation (2)

- Why no abstract class for the products?

- That there is little to gain from giving different products a common parent class
  - The products produced by the concrete builders differ so greatly in their representation
  - The ASCIIText and the TextWidget objects are unlikely to have a common interface, nor do they need one
  - The client usually configures the director with the proper concrete builder, the client is in a position to know which concrete subclass of Builder is in use and can handle its products accordingly

Builder – Related Patterns

- A composite what the builder often builds



Group Discussion

What are similarities and/or differences between Abstract Factory and Builder?

THE UNIVERSITY OF SYDNEY

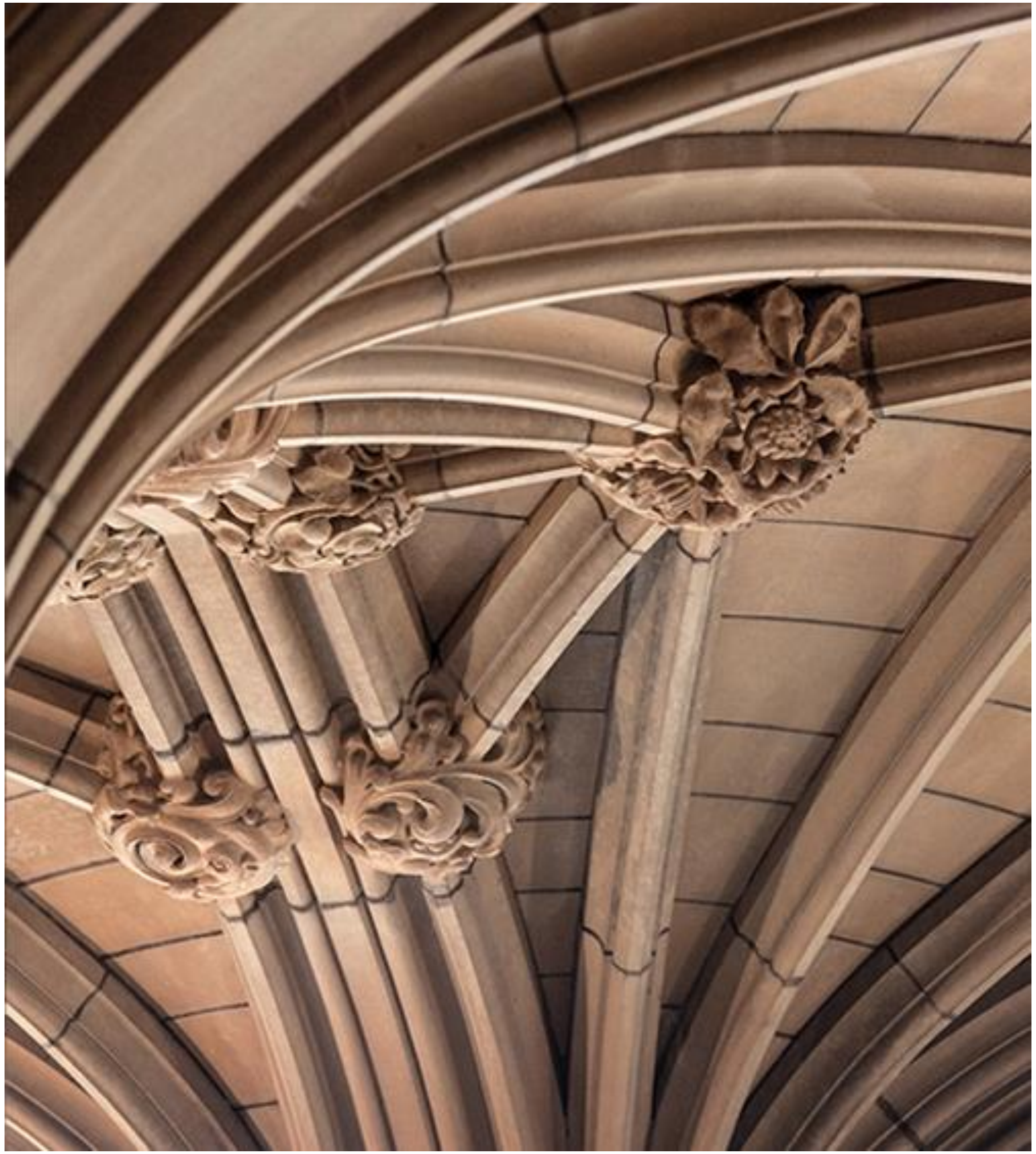
Builder – Related Patterns

- Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step.

- Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the AbstractFactory pattern is concerned, the product gets returned immediately.

Prototype

Object Creational

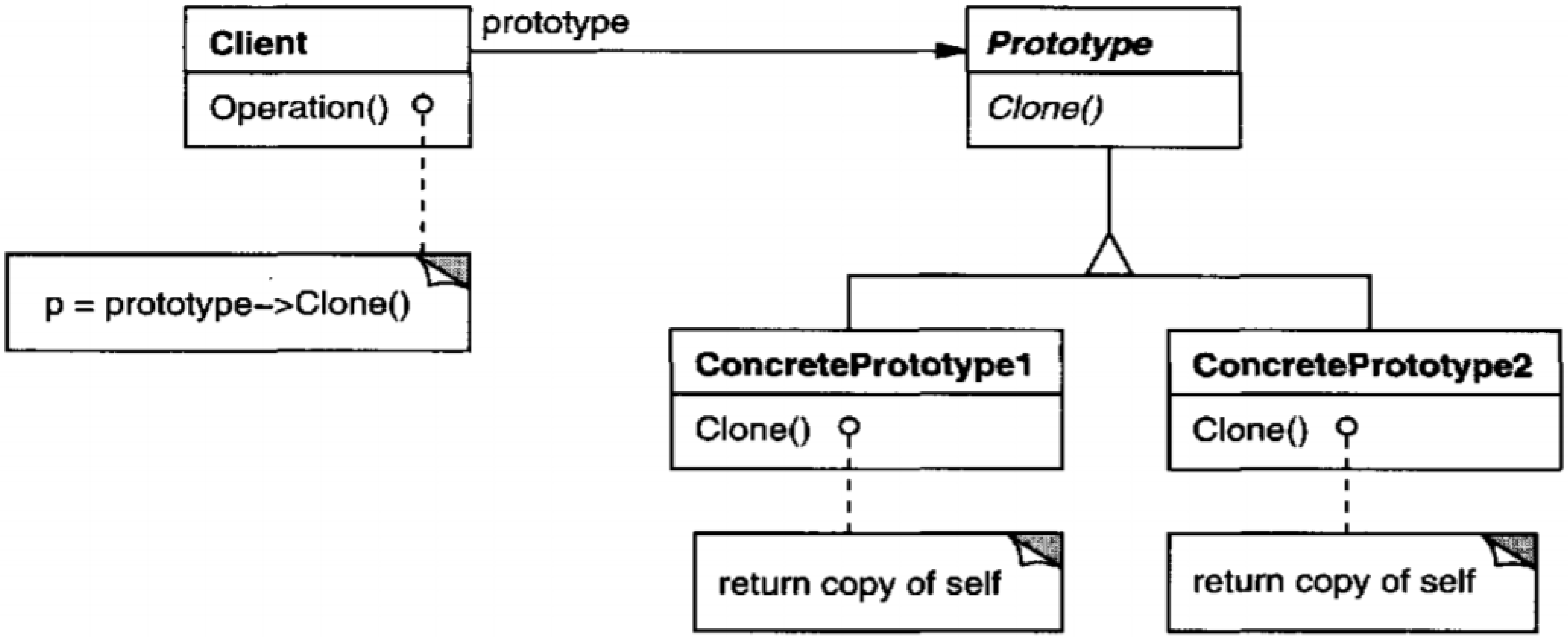


THE UNIVERSITY OF SYDNEY

Prototype

- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- Applicability
  - When a system should be independent of how its products are created, composed, and represented
  - When the classes to instantiate are specified at run-time
  - To avoid building a class hierarchy of factories that parallels the class hierarchy of products
  - When instances of a class can have one of only a few different combinations of state

Prototype – Structure



Prototype – Participants

- **Prototype**
  - Declares an interface for cloning itself
- **ConcretePrototype**
  - Implements an operation for cloning itself.
- **Client**
  - Creates a new object by asking a prototype to clone itself

- **Collaborations**
  - A client asks a prototype to clone itself

Prototype – Consequences (1)

- It hides the concrete product classes from the client, thereby reducing the number of names clients know about

- These patterns let a client work with application-specific classes without modification

- Each subclass of prototype must implement the clone operation

Prototype – Consequences (2)

1- Adding and removing products at run-time

- New concrete product class can be incorporated into a system
  - The client can install and remove prototypes at run-time

2- Specifying new objects by varying values

- New kinds of objects can be defined by instantiating existing classes and registering the instances as prototypes of client objects
- A client can exhibit new behavior by delegating responsibility to the prototype – this kind of design lets users define new "classes" without programming (cloning a prototype is similar to instantiating a class)

Prototype – Consequences (3)

1- Specifying new objects by varying structure

- Many applications build objects from parts and subparts and for convenience let you instantiate complex, user-defined structures
  - Editors for circuit designs build circuits out of sub-circuits; specific sub-circuits can be used

- Similarly, sub-circuit can be added as a prototype to the palette of available circuit elements
  - When the composite circuit object implements clone as a deep copy, circuits with different structures can be prototype

Prototype – Consequences (4)

4- Reduced sub-classing

- By cloning prototype instead of asking factory method to make a new object
  - Factory method produces a hierarchy of Creator classes that parallel the class hierarchy

Prototype – Consequences (5)

5- Configure an application with classes dynamically

An application that wants to create instances of a dynamically loaded class won't be able to reference its constructor statically – how to do this?

- The run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager

- Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally

Prototype – Implementation (1)

Using a prototype manager

- Keep a **registry** (**prototype manager**) of available prototypes when prototypes in a system can be created and destroyed dynamically
  - Clients will store and retrieve prototypes from the register, but will not manage them
  - Before cloning, a client will ask the register for a prototype
  - Prototype manager has operations for registering a prototype under a key and unregistering it
  - Clients can change or even browse the registry at run-time

Prototype – Implementation (2)

Implementing the clone operation

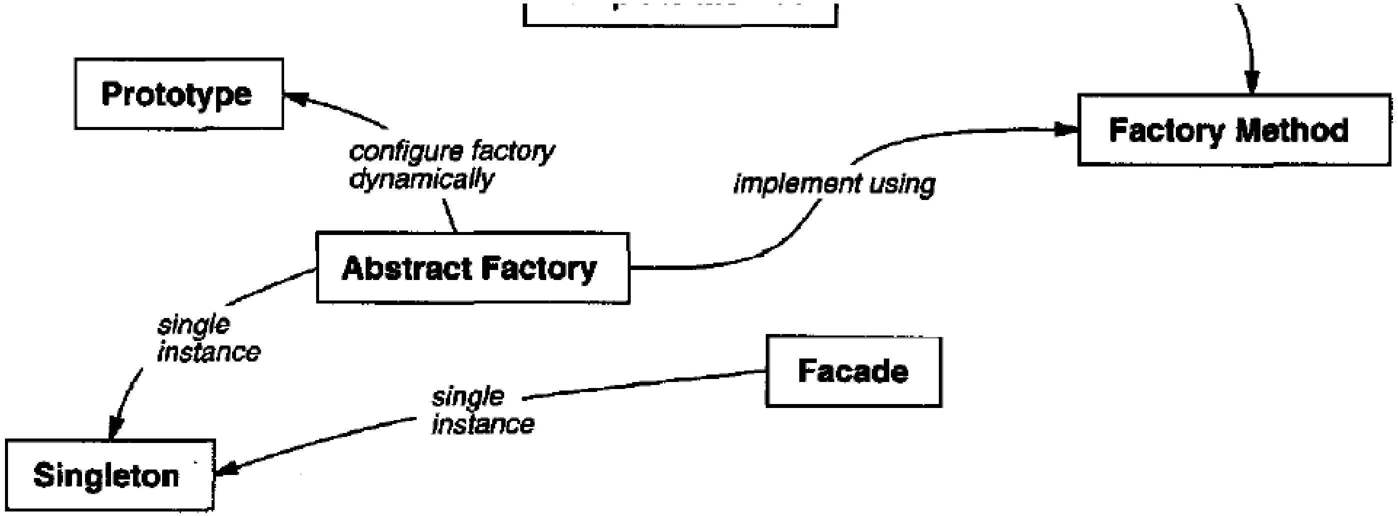
- Shallow copy vs deep copy
  - Does cloning an object in turn clone its instance variables or do the clone and original just share the variables
- Cloning with complex structures usually requires a deep copy because the clone and the original must be independent
  - Ensure the clone's component are clones of the prototype's components

Prototype – Implementation (3)

Initializing clones

- Some clients may want to initialize some or all of its internal state to values of their choice
  - Can't pass these values in the Clone operation, as their number will vary between classes of prototypes
  - Clients may use already defined operations in your prototype classes for re-setting key values/states
  - You may need to introduce such initialize operations that set clone's internal state accordingly
    - Be aware of the deep copying clone operations

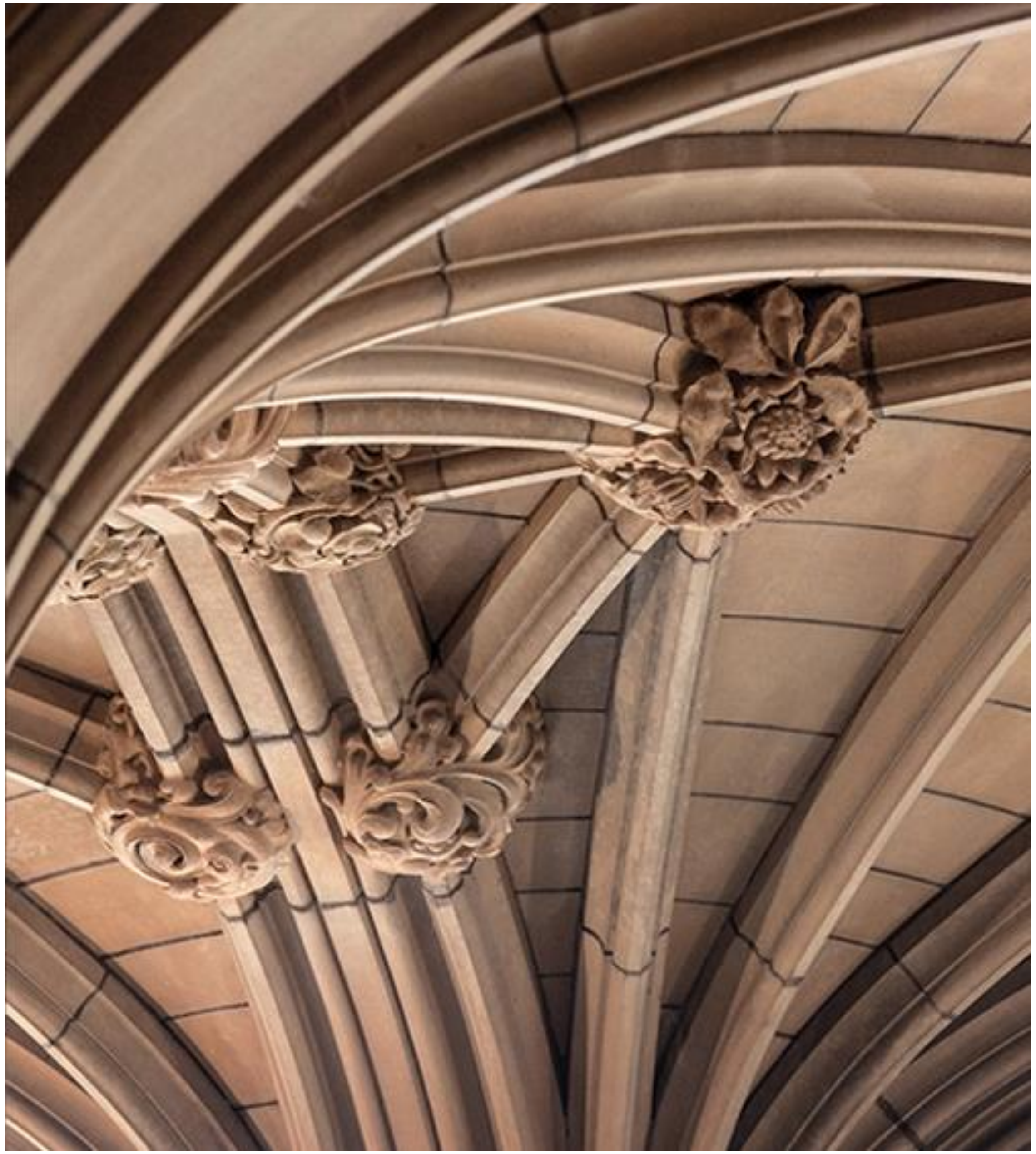
Prototype – Related Patterns



- Prototype and Abstract Factory are competing patterns in some ways
  - They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.

Memento Design Pattern

Object Behavioural



Memento Pattern

- Intent
  - Capture and externalize an object's internal state so that the object can be restored to this state later, without violating encapsulation
- Applicability
  - A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
  - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Memento – Implementing Undo

- Consider implementing checkpoints and undo mechanism that let users revert back of uncertain operations

- How such behavior could be design ?
  - What information should be captured?
  - How and where to store such information?
  - Is the way you proposed to store the information a good design? Why? Why not?

Memento – Implementing Undo

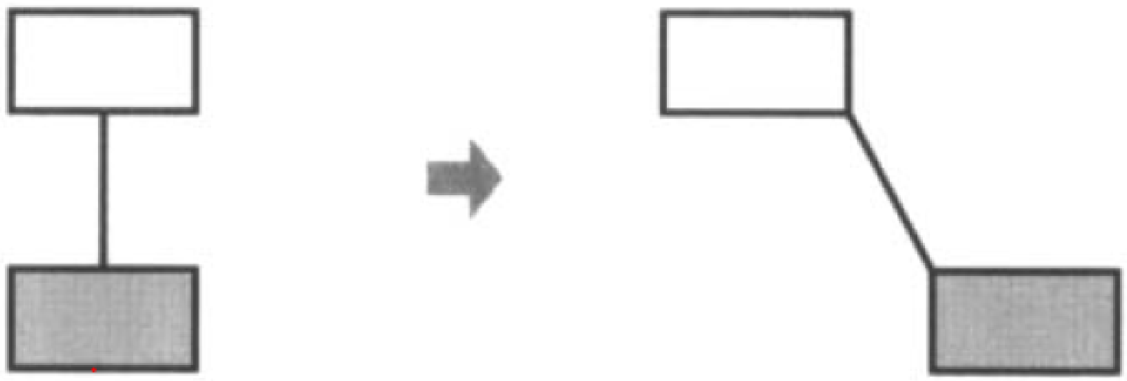
- State information must be stored so that objects can be restored to their previous state

- Encapsulation – objects normally encapsulate some or all of their state – which makes it inaccessible to other objects and impossible to save externally

- Exposing this state would violate encapsulation; and can compromise application's reliability and extensibility

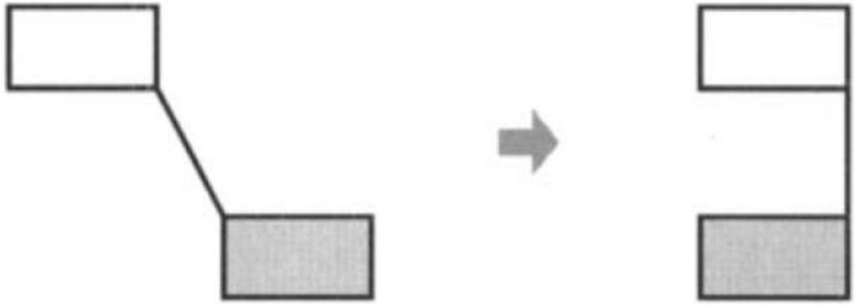
Memento – Graphics Editor (Undo)

- A graphics editor allow users to connect and move shapes
  - E.g., connect two rectangles with a line
  - Rectangles stay connected when the user moves either of the rectangles
  - The editor ensures that the line stretches to maintain the connection



Memento – Graphics Editor (Undo) – Problem

- ConstraintSolver object records connections as they are made and generates mathematical equations that describe them
- ConstraintSolver uses the results of its calculation to rearrange the graphics so that they maintain the proper connections
- How to "undo" a move operation?
  - Store the original distance moved and move the object back to an equivalent distance



Memento – Graphics Editor (Undo) – Problem

- Does the ConstraintSolver's public interface suffice to allow precise reversal of its effect on other objects?
  - The undo mechanism must work closely with the ConstraintSolver to re-establish the previous state
  - Should avoid exposing the ContraintSolver's internals to the undo mechanism

- How to solve this problem?

Memento – Graphics Editor (Undo) – Solution

- The undo mechanism requests a memento from the originator when it needs to checkpoint the originator’s state

- The originator initialize the memento with current state information

- Only the originator can store and retrieve information from memento – the memento is not transparent to other objects

Memento – Graphics Editor (Undo) – Solution

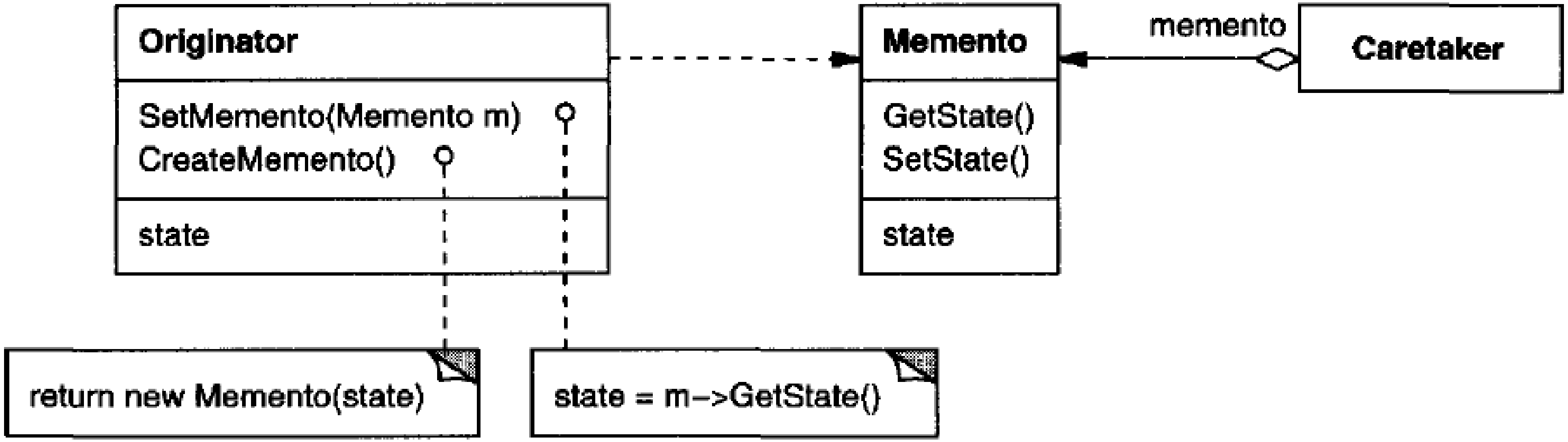
1. When a user makes a move operation, the editor requests a memento from the ConstraintSolver

2. The ConstraintSolver creates and returns a memento, an instance of a class (e.g., SolverState), which contains data structures that describe the current state of the ConstraintSolver's internal equations and variables

3. When the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver

4. The ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state

Memento – Structure



Memento – Participants (1)

- **Memento (SolverState)**
  - Stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator’s discretion

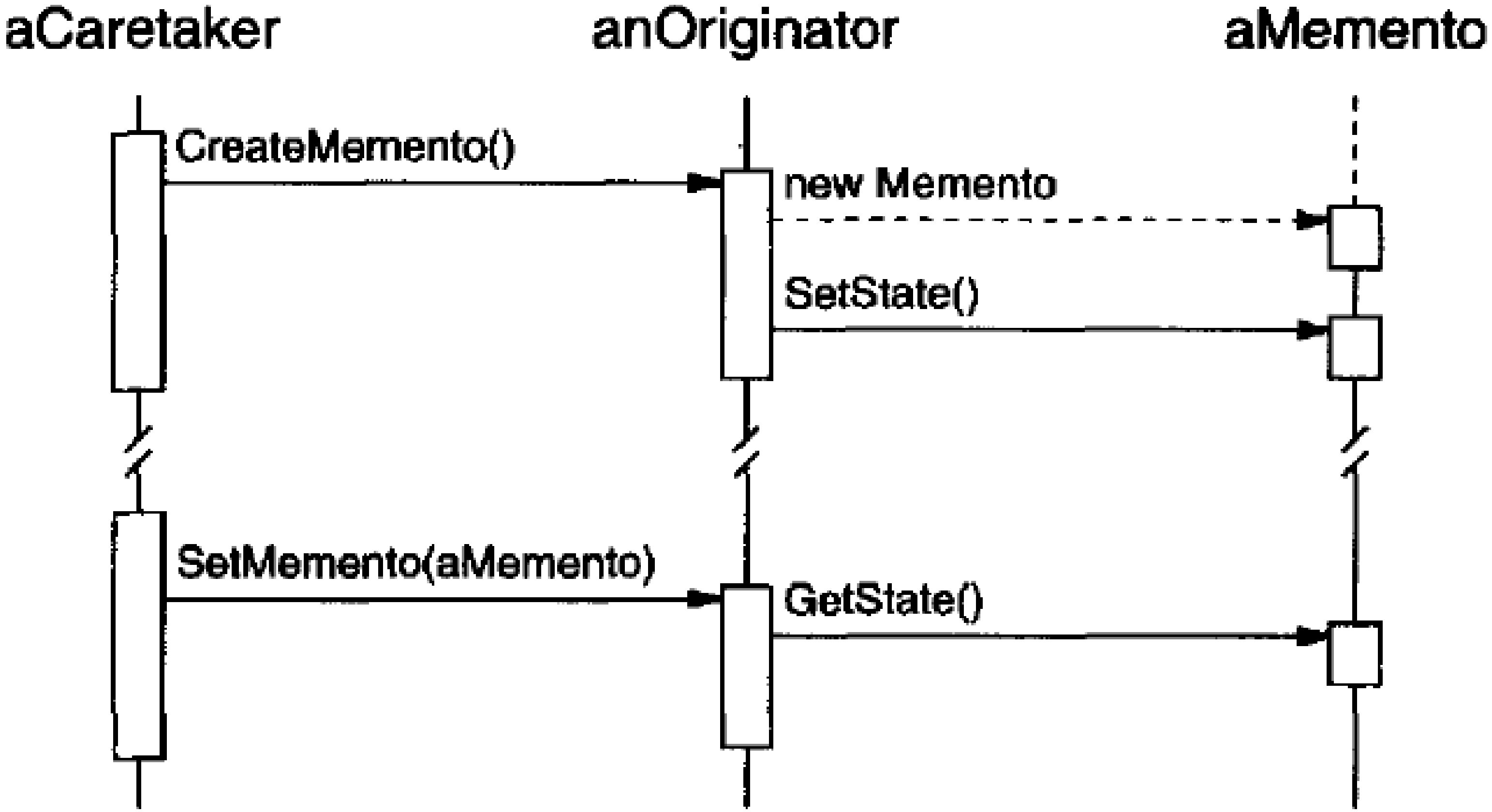
  - Protects against access by objects other than the originator. Mementos have effectively two interfaces
    - *Caretaker:* sees a narrow interface to the Memento — it can only pass the memento to other objects
    - *Originator:* sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state

Memento – Participants (2)

- **Originator (ConstraintSolver)**
  - Creates a memento containing a snapshot of its current internal state
  - Uses the memento restore its internal state

- **Caretaker (undo mechanism)**
  - Responsible for the memento's safekeeping
  - Never operates on or examines the contents of a memento

Memento – Collaborations



Memento – Consequences (1)

- Preserve encapsulation boundaries
  - By protecting other objects from potentially complex originator internals

- Simplifies Originator
  - Originator keeps the versions of internal state that clients have requested

- Might be expensive
  - In case originator must copy large amounts of information to store in the memento or
  - If clients create and return mementos to the originator often enough

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.