Software Engineering I (02161)

Week 10

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

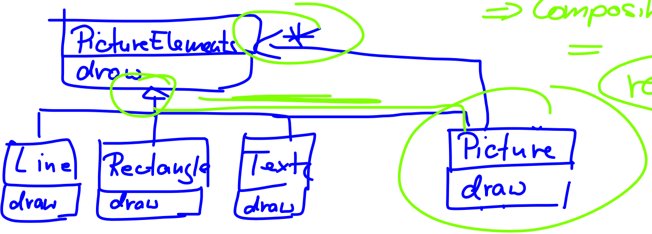
Spring 2013

Contents

Composite Pattern

Problem: Graphics Editor

- Line, Rectangle, Text
  - can be drawn
- Picture: can contain Line, Rectangle, Text, and Picture
  - can be drawn



⇒ Composite Pattern

=

recursive Datastructure

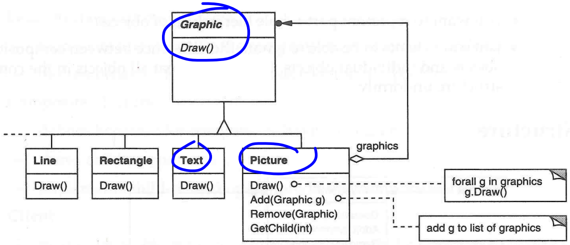
≈

⇒ in principle "infinite" datatypes

Composite Pattern

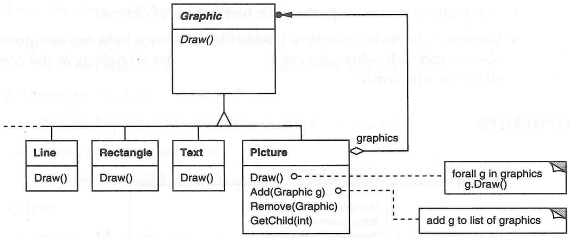
Composite Pattern

Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

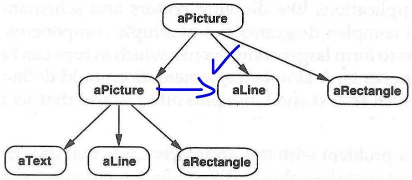


Composite Pattern: Graphics

- Class Diagram



- Instance diagram



Template Method Problem

Overdue message for Book:

1 compute due date for a book
   - a get the ~~current~~ borrow date
   - b add the max days for loan for the book

2 check if the current date is after the due date

DRY

Overdue message for CD:

1 compute due date for a cd
   - a get the current date
   - b add the max days for loan for the cd

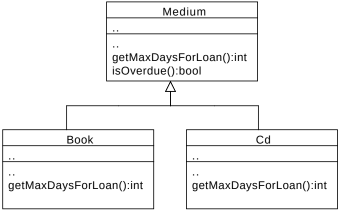
2 check if the current date is after the due date

Template method

Template alg.

- Create a template method in class Medium:
  1. compute due date for a medium
     a. get the current date
     b. add the max days for loan for that medium "Holes"
  2. check if the current date is after the due date
- In book method for getMaxDaysForLoan returning 4 weeks
- In CD getMaxDaysForLoan returns 2 weeks

Template Method




```
public abstract class Medium {
```

template method

```
  public boolean isOverdue() {
    if (!isBorrowed()) {
      return false;
    }
    Calendar date = libApp.getDate();
    Calendar latestReturnDate = new GregorianCalendar();
    latestReturnDate.setTime(borrowDate.getTime());
    latestReturnDate.add(Calendar.DAY_OF_YEAR, getMaxDaysForLoan());
    return latestReturnDate.before(date);
```

this.

plug in for subclass methods

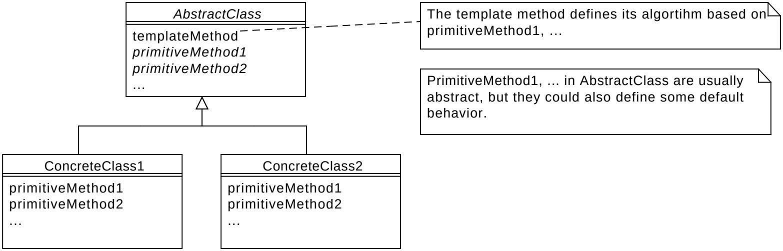
```
  }

  public abstract int getMaxDaysForLoan();
}
```

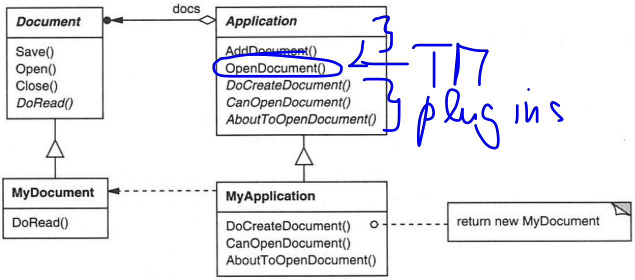
Template Method

Template Method

Define the skeleton of an algortihm in an operation, deferring some steps to subclasses. Template Method lets sublcasses redefine certain steps of an algorithm without changing the algorithm's structure.



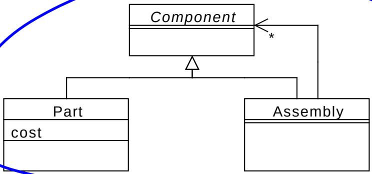
Template Method




```
public abstract class Application {
  public void openDocument(String name) {
    if (canOpenDocument(name)) {
      Document doc = createDocument(name);
      if (doc != null) {
        docs.add(doc);
        aboutToOpenDocument(doc);
        doc.open();
        doc.read();
      }
    }
  }
}
```

Visitor Pattern: Problem

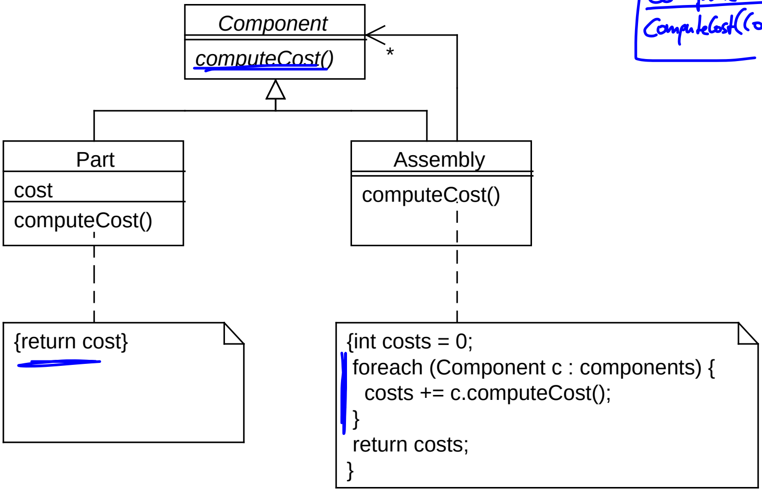
- Define a mechanism to define algorithms on complex datastructures without modifying the class, e.g. if the class is a library class
- For example, add a computeCost algorithm without adding the method to the class



Bike (Ass)

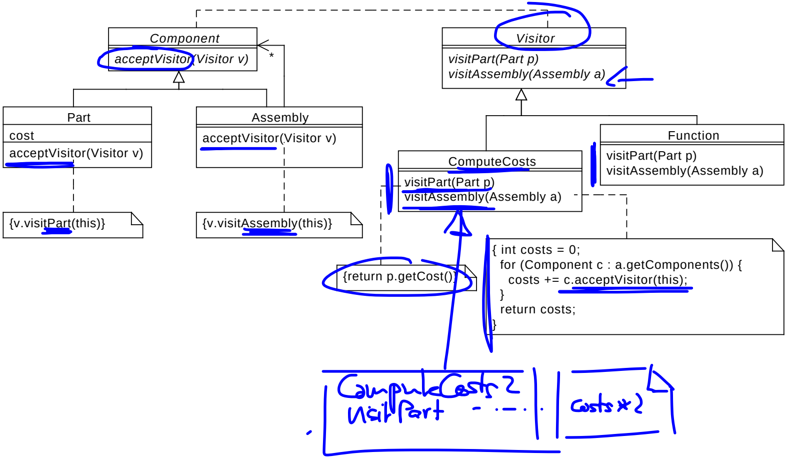
- Frame (1000 kr)
- Wheel: 28 spokes (1 kr), rim (100 kr), tire (100 kr) (Ass)
- Wheel: 28 spokes (1 kr), rim (100 kr), tire (100 kr) (Ass)

Example: compute costs for components

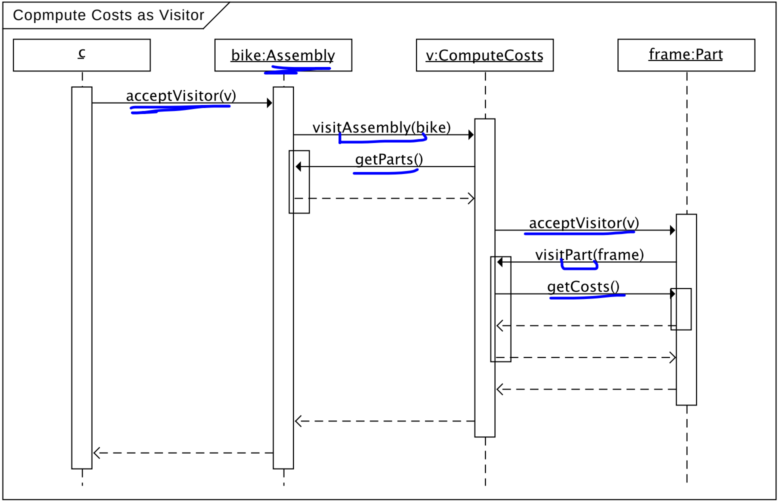


ComputeCosts
ComputeCost(Comp. c)

Example: compute costs as a visitor



Compute costs as a visitor



Visitor Pattern

Visitor Pattern

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Double Dispatch: Problem I

Define a plus method for MyFloat and MyInteger classes (both subclasses of MyNumber) without using conditionals

- In MyFloat

```
public MyFloat plus(MyNumber n) {
    if (n instanceof MyFloat) {
        return this.plusFloat(n);
    }
    if (n instanceof MyInteger) {
        return this.plusFloat(n.convertToFloat());
    }
}
```

Double Dispatch: Problem II

- In MyInteger

```
public MyFloat plus(MyNumber n) {
    if (n instanceof MyFloat) {
        return this.conv.toFloat().plusFloat(n);

      }
    if (n instanceof MyInteger) {

        return n.plusInt(this);

    }
}
```

Double Dispatch: Solution

→ Smalltalk Ruby

Using double dispatch:

- In MyFloat

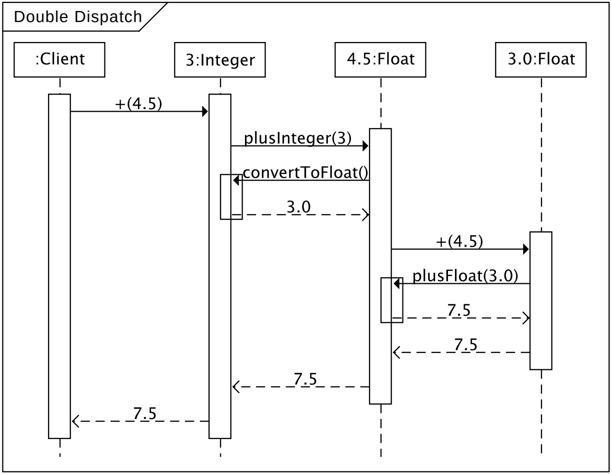
```
public MyNumber plus(MyNumber n) {
   return n.plusFloat(this);
}

public MyNumber plusFloat(MyFloat n) {
   return ... // Do the computation on floats
}
public MyNumber plusInteger(MyInteger n) {
   return this.plus(n.convertToFloat());
}
```

- In MyInteger

```
public MyNumber plus(MyNumber n) {
   return n.plusInteger(this);
}
public MyNumber plusFloat(MyFloat n) {
   ret. this.convToFloat().plus(n);
}
public MyNumber plusInteger(MyInteger n) {
   return .... // primitive int. addition
}
```

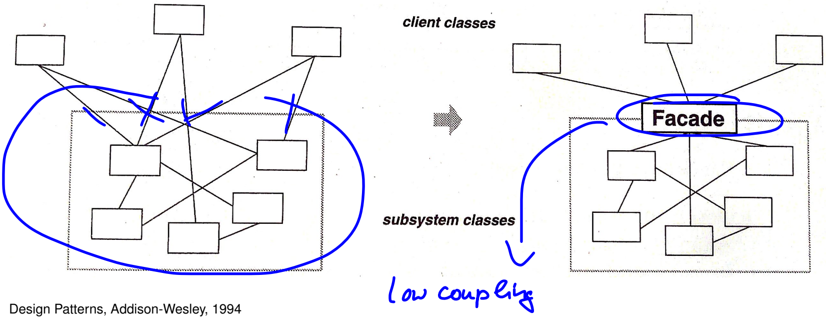
Double Dispatch: Compute $3 + 4.5$



Facade

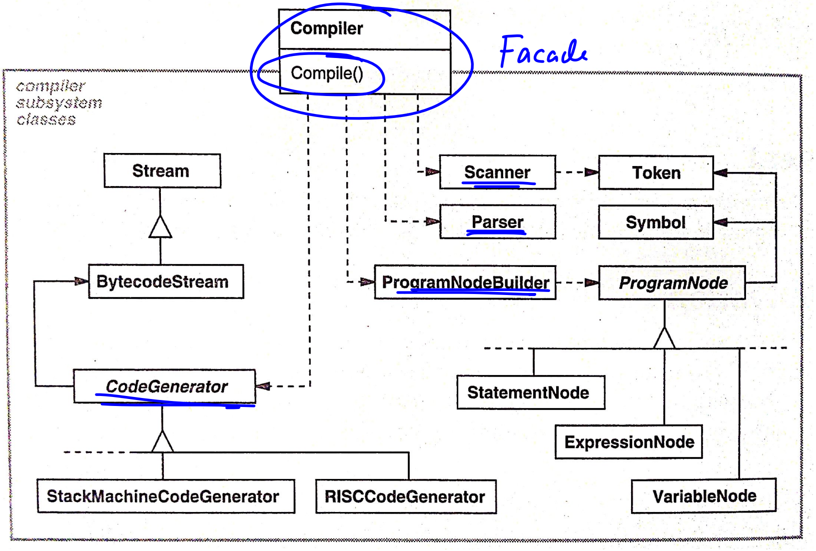
Facade

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



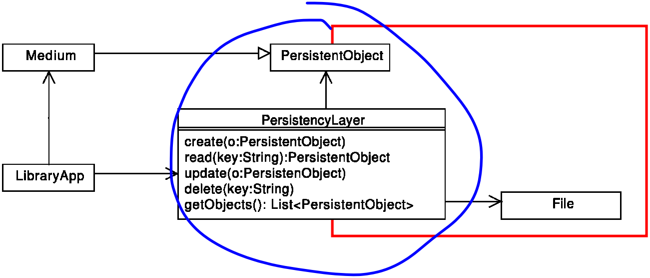
Design Patterns, Addison-Wesley, 1994

Example Compiler



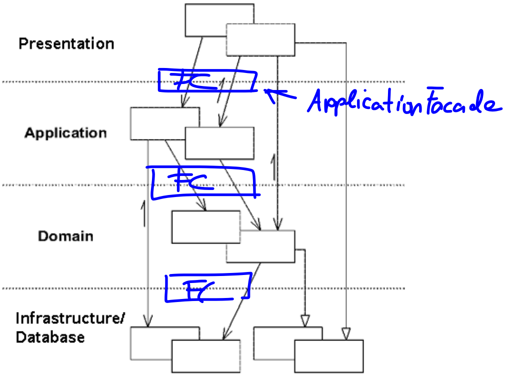
Design Patterns, Addison-Wesley, 1994

Example Persistency Layer



Layered Architecture

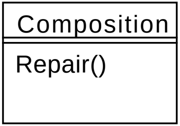
Use Facades to provide simple interfaces to the different layers



Eric Evans, Domain Driven Design, Addison-Wesley, 2004

Strategy / Policy: Problem

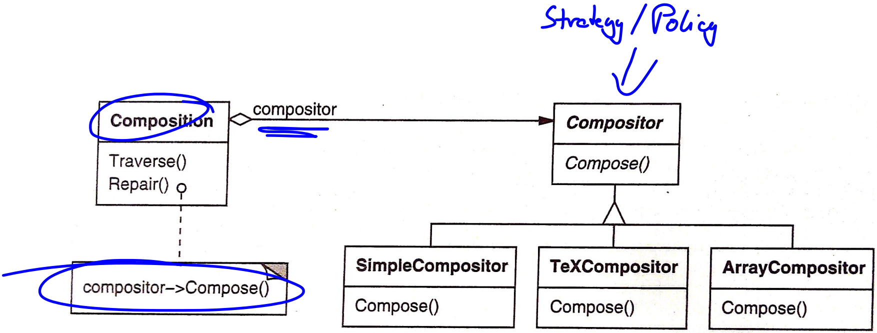
Different strategies for layouting text: simple, TEX, array, . . . Example: Text formatting




```
public void repair() {
    ...
    if (strategy = "simple")
          // Do simple linebreak
    else if (strategy = "tex")
        // Use TeX's algorithm
    else if (strategy = "array")
        // Do array style linebreak
    ..
}
```

Compose method into a strategy obj.

Strategy Pattern: Solution

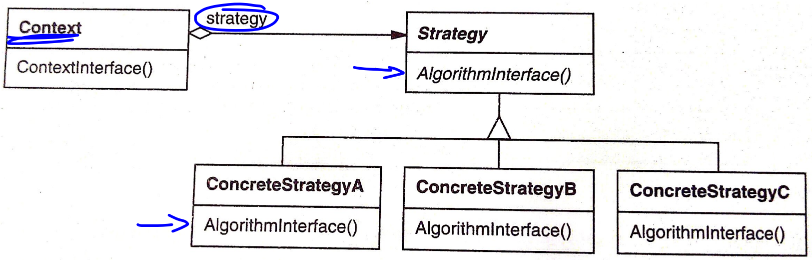


Design Patterns, Addison-Wesley, 1994

Strategy / Policy

Strategy / Policy

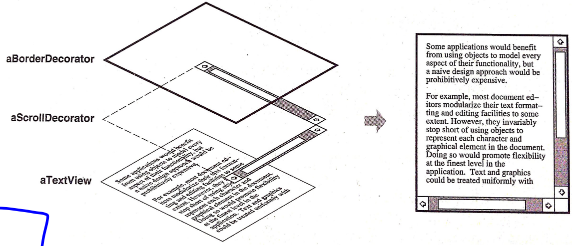
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Design Patterns, Addison-Wesley, 1994

Decorator: Problem

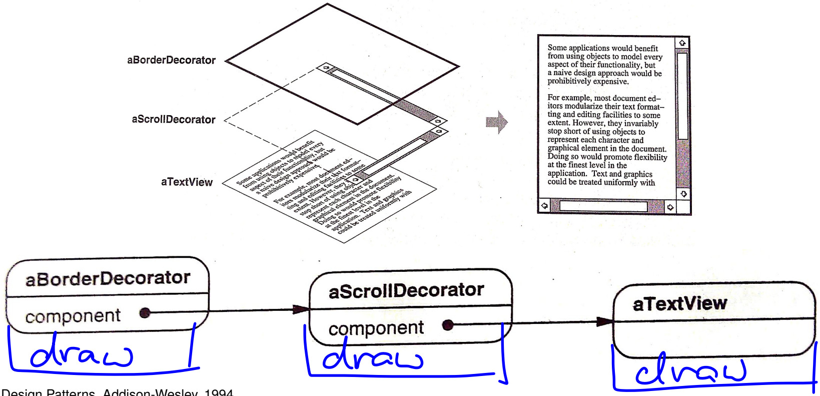
- Composing a set of functionalities
- Eg. Basic windows, windows with borders, windows with scrollbars



Window

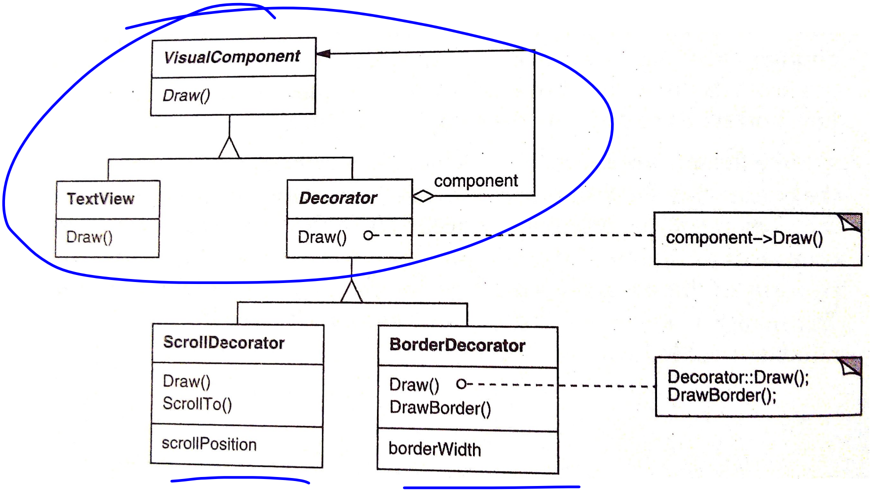
drawContents
drawBorder
drawScrollbar
draw

Example: Window decorators



Design Patterns, Addison-Wesley, 1994

Example: Window decorators

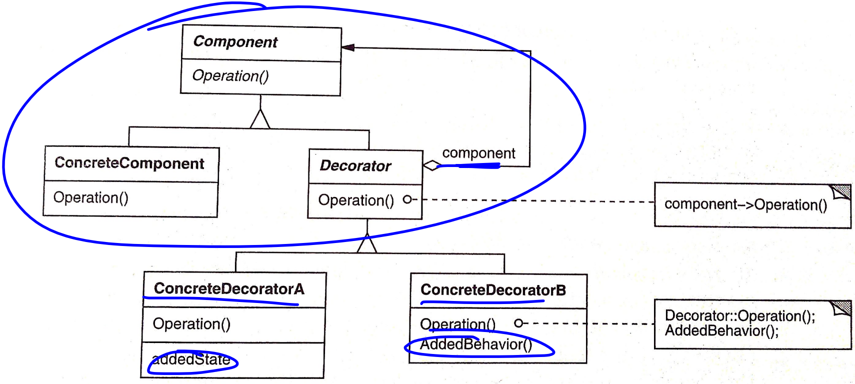


Design Patterns, Addison-Wesley, 1994

Decorator

Decorator

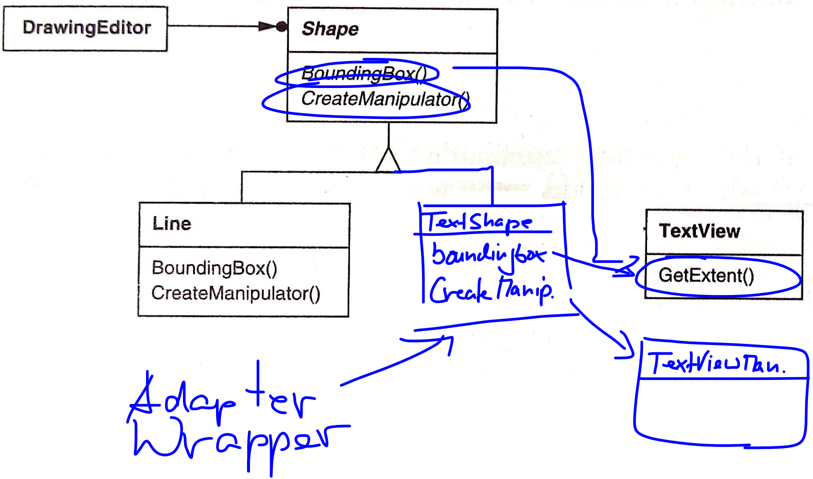
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



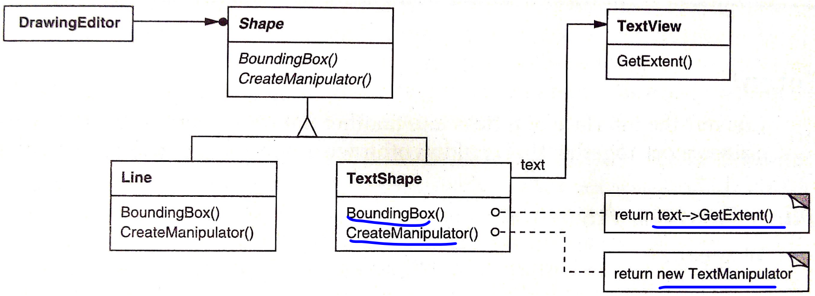
Design Patterns, Addison-Wesley, 1994

Adapter / Wrapper: Problem

- I want to include a text view as part of my graphic shapes
  - Shapes have a bounding box
  - But text views only have an method GetExtent()



Example: Using text views in a graphics editor

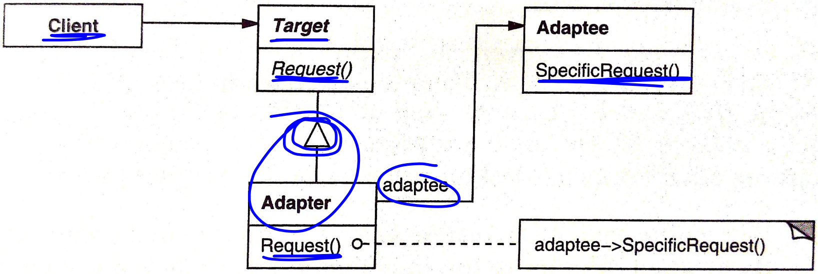


Design Patterns, Addison-Wesley, 1994

Adapter / Wrapper

Adapter / Wrapper

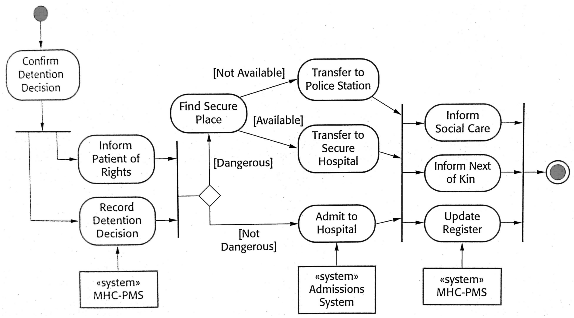
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Design Patterns, Addison-Wesley, 1994

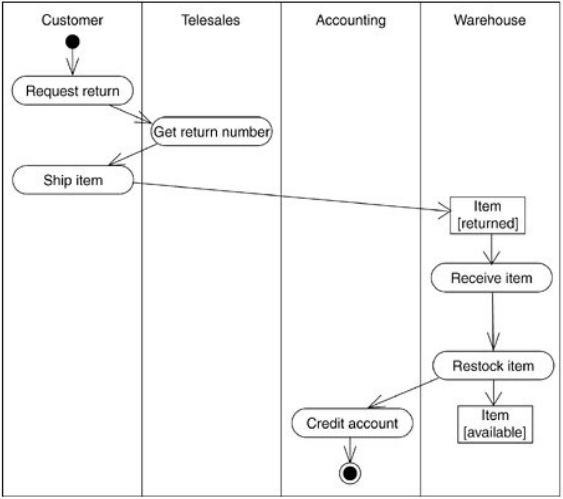
Contents

Activity Diagram: Business Processes

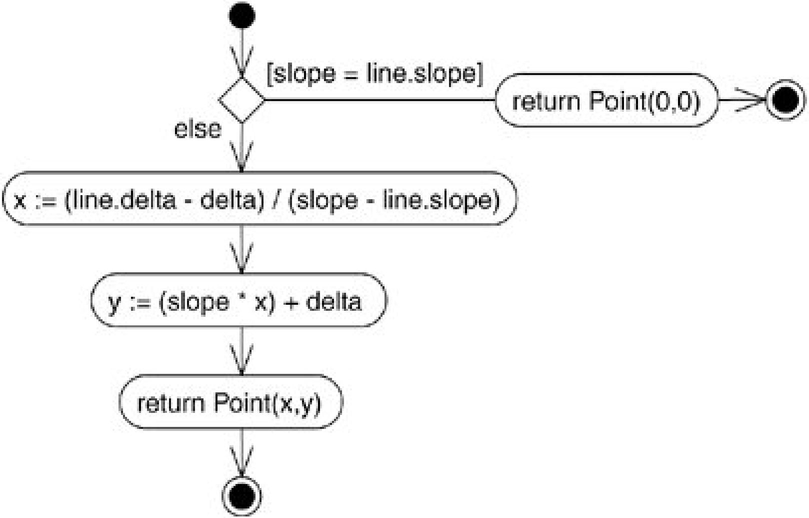


- Describe the *context* of the system
- Helps finding the requirements of a system
  - modelling business processes leads to suggestions for possible systems and ways how to interact with them
  - Software systems need to fit in into existing business processes

Activity Diagram Example Workflow



Activity Diagram Example Operation

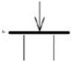
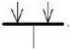
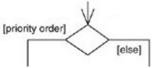


UML Activity Diagrams

- Focus is on *control flow* and *data flow*
- Good for showing *parallel/concurrent* control flow
- Purpose
  - Model business processes
  - Model workflows
  - Model single operations
- Literature: UML Distilled by Martin Fowler

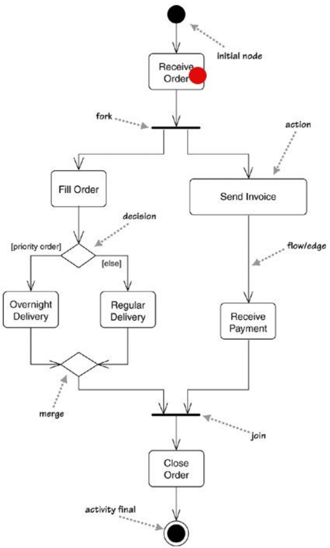
Activity Diagram Concepts

- *Actions* 

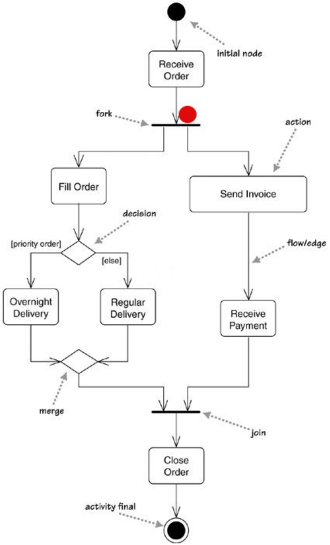
  - Are atomic
  - E.g Sending a message, doing some computation, raising an exception, . . .
    - UML has approx. 45 Action types
- *Concurrency*
  - Fork: Creates concurrent flows 
    - Can be true concurrency
    - Can be interleaving
  - Join: Synchronisation of concurrent activities 
    - Wait for all concurrent activities to finish (based on token semantics)
- *Decisions* 

  - Notation: Diamond with conditions on outgoing transitions
  - `else` denotes the transition to take if no other condition is satisfied

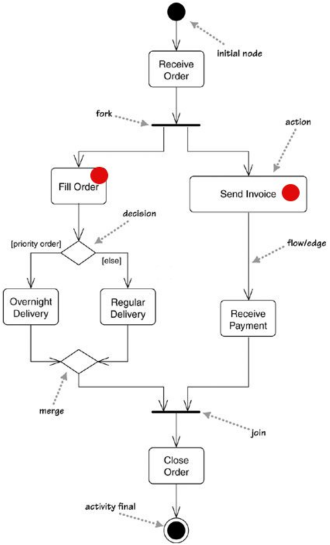
Activity Diagrams Execution



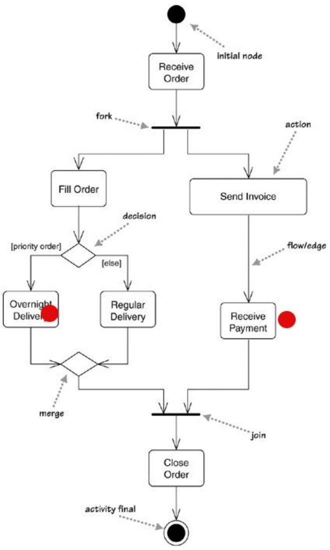
Activity Diagrams Execution



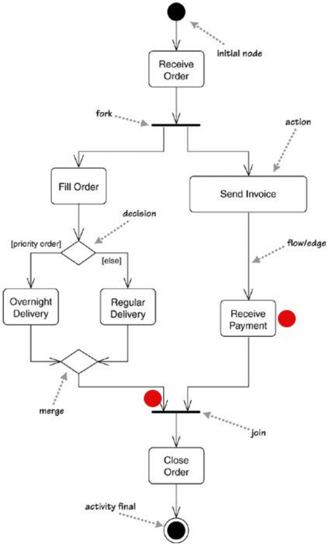
Activity Diagrams Execution



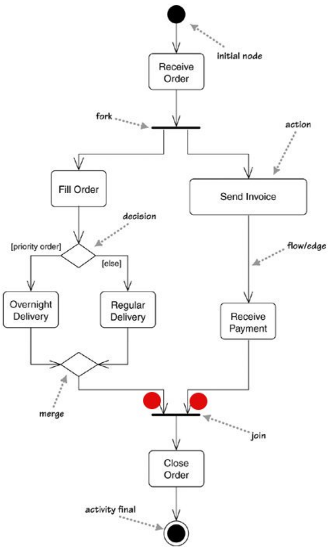
Activity Diagrams Execution



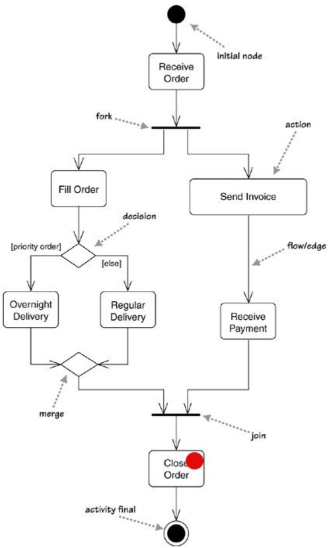
Activity Diagrams Execution



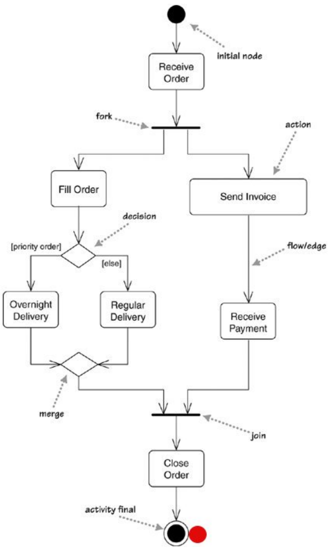
Activity Diagrams Execution



Activity Diagrams Execution

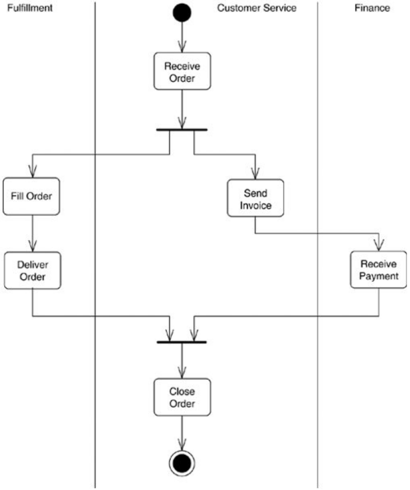


Activity Diagrams Execution

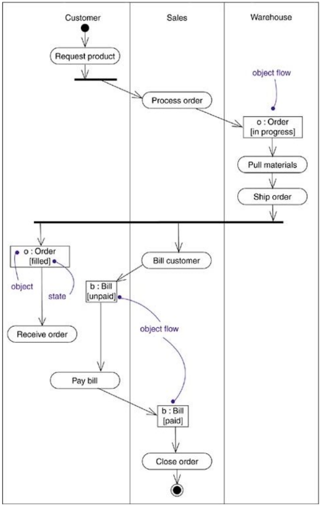


Swimlanes / Partitions

- Swimlanes show **who** is performing an activity

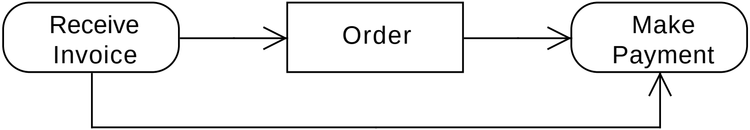


Objectflow example



Data flow and Control flow

- *Data flow* and *control flow* are shown:



- Control flow can be omitted if implied by the data flow:

Use of Activity Diagrams

- Emphasise on concurrent/parallel execution
- Requirements phase
  - To model business processes / workflows to be automated
- Design phase
  - Show the semantics of one operation
    - Close to a graphic programming language

Activity Diagram vs State Machines