Refining the Requirements Model

Chapter 8

In this Lecture you will Learn:

- What is meant by a component
- How generalization and aggregation help to develop reusable components
- How to identify generalization and composition
- How to model generalization and composition
- What is meant by the term pattern
- What types of patterns can be used in software development

Component-based Development

- Component-based development means either:
  - Assembling software from pre-existing components, or
  - Building components for others to use

Class Exercise

- What are the advantages of components?

Problems of Software Components

- Why are components hard?
  - The NIH (Not-Invented-Here) Syndrome
  - Model Organization

The NIH Syndrome

- In spite of all the library resources available today, some professional programmers still fall preys to the NIH syndrome.
- This is the attitude of one who thinks:
  - I don’t trust other people’s widgets – even those that appear to work, suit my purpose and are affordable – I want to invent my own widgets anyway.

Model Organization

- It is difficult enough to create a model that is useful at other stages of one single project
- It is even harder to create a structured model that is useful on a completely different project.

Class Exercise

- Explain why the NIH (Not-Invented-Here) syndrome occurs.

Component-based Development

- The contribution of object-orientation to reuse:
  - Encapsulation of internal details makes it easier to use components in systems for which they were not designed
  - Generalization hierarchies make it easier to create new specialized classes when they are needed
  - Composition and aggregation structures can be used to encapsulate components

Important of Encapsulation

- In building a software system from ready-made components, it is important that the interfaces between components should be completely described, reasonably simple and preferably standardized.
- If this applies to a particular component, there is no need to know any of the details of its internal construction in order to understand how to use it.
- If it is necessary to know about its internal construction, then the interface is probably neither simple nor standardized (it will be unique to that component) and the component will be tightly coupled to any other components with which it interacts.
- This greatly increases the difficulty of making any change to the system as a whole, whether this is intended to bring an improvement or to repair a fault.

Important of Generalization

- Generalized components have the capacity to be reused in many different situations.
- As a component becomes more specialized, it can be used in fewer and fewer different situations.

Class Exercise

- What does object-orientation offer that helps to create reusable components?

Composition in UML

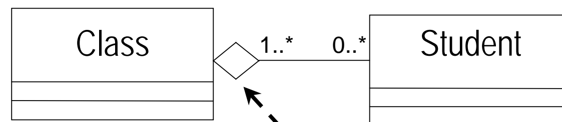
- Composition is a type of abstraction that encapsulates groups of classes that collectively have the capacity to be a reusable sub-assembly.
- Special types of association, both sometimes called whole-part

Composition and Aggregation

- Aggregation is essentially any whole-part relationship
- Semantics can be very imprecise
- Composition is 'stronger':
  - Each part may belong to only one whole at a time
  - When the whole is destroyed, so are all its parts

Aggregation

- An everyday example

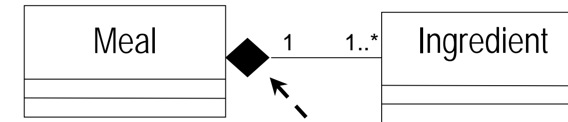


Unfilled diamond signifies aggregation

- Clearly not composition
  - Students could be in several classes
  - If class is cancelled, students are not destroyed!

Composition

- Another everyday example

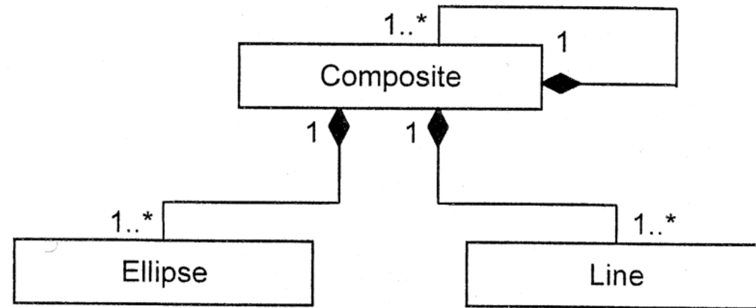


Filled diamond signifies composition

- This is (probably) composition
  - Ingredient is in only one meal at a time
  - If you drop your dinner on the floor, you probably lose the ingredients too

Example

- Composition used in class diagram to represent composite objects



Class Exercise

- Distinguish composition from aggregation.

Adding Structure

- Add generalization structures when
  - Two classes are similar in most details, but differ in some respects
  - May differ
    - In behaviour (operations or methods)
    - In data (attributes)
    - In associations with other classes

Example

- Analyst's note of the differences between Agate Staff Type

```
17 March - brief interview with Amarjeet Grewal (Finance Director)
Purpose - clarification of points from last Thursday's interview

Asked about staff types
        - only two types seem relevant to system -
                creative staff (C) and  admin staff (A)
How do they differ?
        - main difference is bonus payment...
                1. (C) bonus calculated on basis of campaign profits
                        (only those campaigns they worked on)
                2. (A) paid rate based on average of all campaign profits
Any other diffs?  Amarjeet says -
        - C qualifications need to be recorded
        - C can be assigned as contact for a client
        - A are not assigned to specific campaigns
No other significant differences.
                (NOTE - at next interview, get details of both algorithms)
```

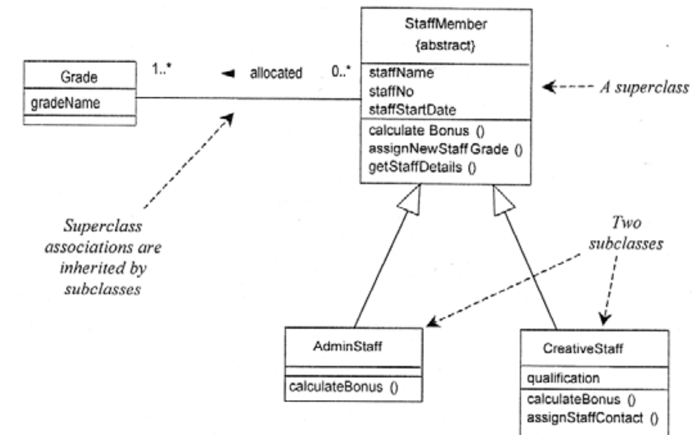
Example

- This note highlight some useful information that must be modeled appropriately
  - There are two types of staffs
  - Bonuses are calculated differently
  - Different data should be recorded for each type of staff

Creative	Have qualifications recorded Can be client contact for campaign Bonus based on campaigns they have worked on
Admin	Qualifications are not recorded Not associated with campaigns Bonus not based on campaign profits

Example

- A generalization hierarchy describes a relationship among Agate staff types.

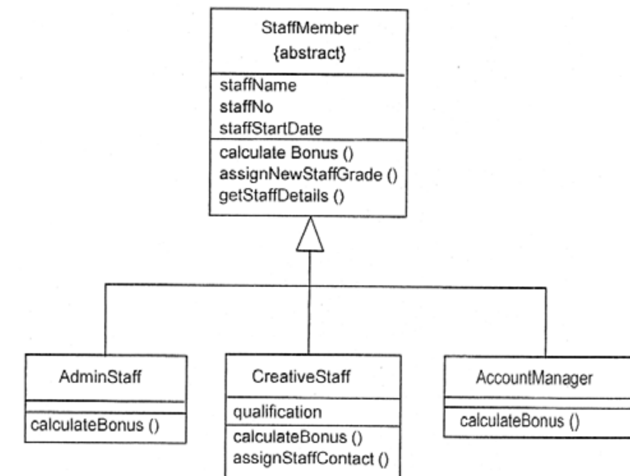


Usefulness of Generalization

- Imagine that the Agate system is completed and in regular use.
- Some time after installation, the Directors decide that they want to reorganize the company, and one of the results is that Account Managers are to be paid bonuses related to campaign profits.
- Their bonus is to be calculated in a different way from both Administrative and other Creative Staff, and is to include an element from campaigns that they supervise, and an element from the general profitability of the company.

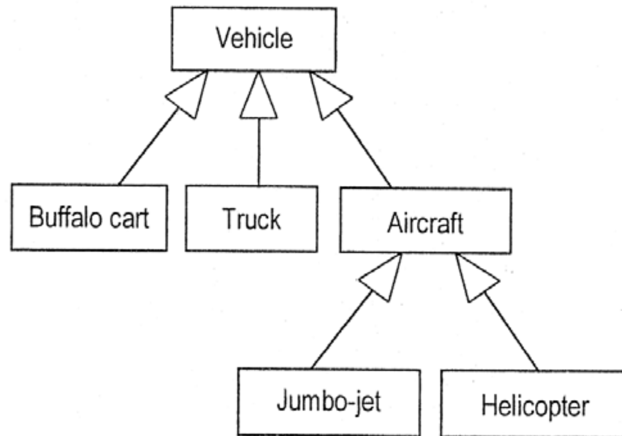
Usefulness of Generalization

- The new subclass is easy to add for the previous diagram

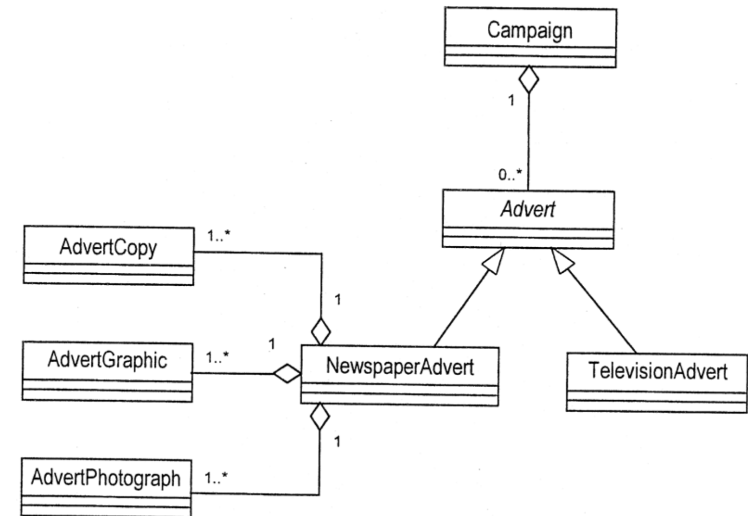


Identifying Generalization (Top-down Approach)

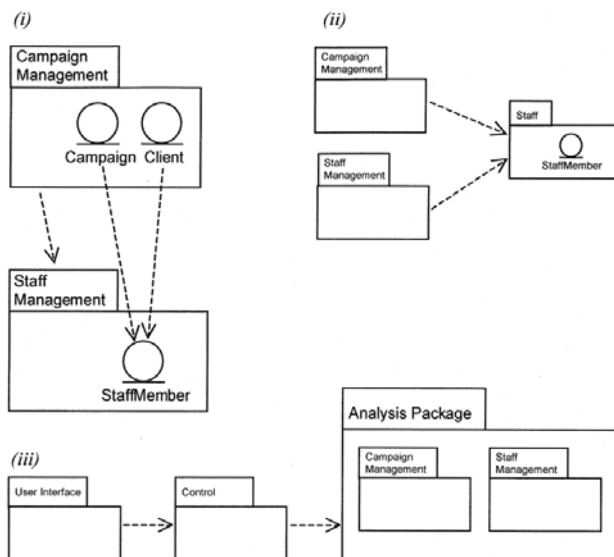
- If an association can be described by the expression is a kind of, then it can usually be modeled as generalization.



Combining Generalization with Composition or Aggregation



Analysis Packages and Dependencies



What is a Pattern?

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
  - Alexander et al. (1977)
- A pattern is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts.
  - Riehle and Zullighoven (1996)
- Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.
  - Gabriel (1996)

Software Development Patterns

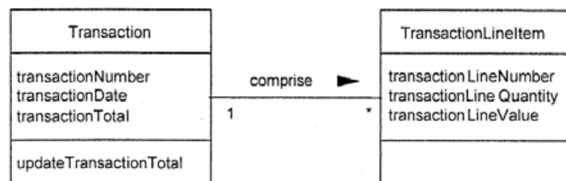
- According to one influential definition (Gabriel, 1996), a pattern comprises three elements:
  - A **Context** in which a given problem occurs repeatedly
  - A **Set of Forces** that influence or constrain the possible solutions
  - A **Software Configuration** that allows these forces to be resolved.

Critical Aspects of a Pattern

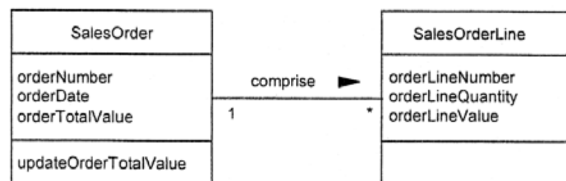
- Coplien (1996) identifies the critical aspects of a pattern as follows:
  - It solves a problem
  - It is a proven concept
  - The solution is not obvious
  - It describes a relationship
  - The pattern has a significant human components

Example

- A simple example of an analysis pattern from Coad et al (1997) is the Transaction-Transaction Line Item pattern



- This pattern might be applied to sales order processing system.



Software Development Patterns

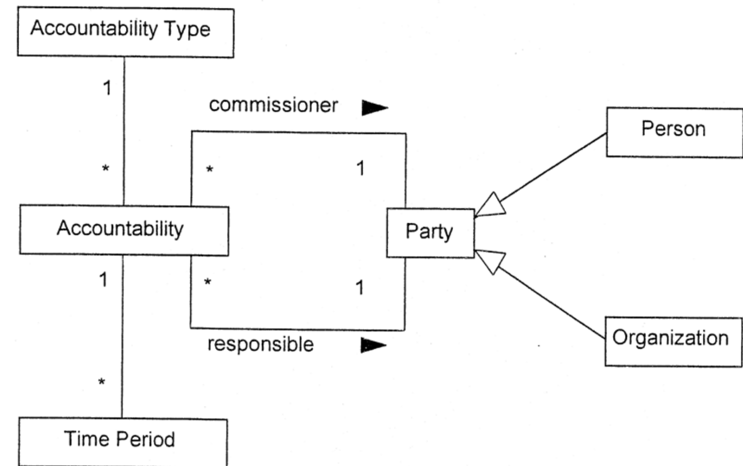
- Patterns are found at many points in the systems development lifecycle
  - **Analysis Patterns** are groups of concepts useful in modelling requirements
  - **Architectural Patterns** describe the structure of major components of a software system
  - **Design Patterns** describe the structure and interaction of smaller software components

Software Development Patterns

- Patterns have been applied widely in software development
  - **Organization Patterns** describe structures, roles and interactions in the software development organization itself

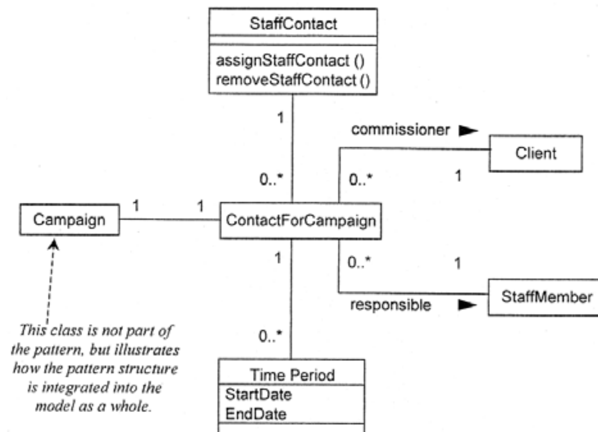
Example

- The Accountability analysis pattern adapted from Fowler.



Example

- The previous Accountability pattern can be applied to Agate's StaffContact relationship



Antipattern

- An antipattern captures practice that is demonstrably bad (by documenting an attempted solution that failed), and can also provide a reworked solution that can be applied within a specific context.
- Mushroom Management is an organization antipattern in the domain of software development organizations.

Class Exercise

- How do patterns help the software developer?