Design Patterns

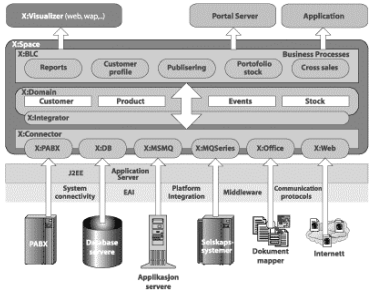
Chapter 15

In this Lecture you will Learn:

- What types of patterns have been identified in software development
- How to apply design patterns during software development
- The benefits and difficulties that may arise when using patterns

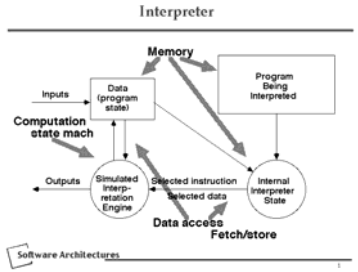
Patterns vs. Frameworks

- Frameworks are *partially completed software systems* that may be targeted at a specified type of application.
- Frameworks is a *reusable mini-architecture* that provides structure and behaviour common to all applications of this type.



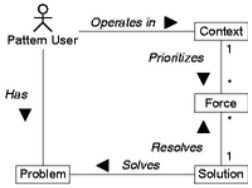
Patterns vs. Frameworks

- Patterns are *more abstract and general* than frameworks
- Patterns *cannot be directly implemented* in a particular software environment
- Patterns are *more primitive* than frameworks



Pattern Catalogues and Languages

- Pattern are group into Catalogues and Language:
  - A **Pattern Catalogue** is a *group of patterns that are related to some extent and may be used together or independently of each other*
  - The patterns in a **Pattern Language** are more *closely related*, and *work together* to solve problems in a specific domain



Software Development Principles and Pattern

- Buschmann et al. (1996) suggest the key principles that underlie pattern.
  - Abstraction
  - Encapsulation
  - Information Hiding
  - Modularization
  - Separation of Concerns
  - Coupling and Cohesion
  - Sufficiency
  - Completeness and Primitiveness
  - Separation of Policy and Implementation
  - Separation of Interface and Implementation
  - Single Point of Reference
  - Divide and Conquer

Patterns and Non-functional Requirements

- Buschmann et al. (1996) identify the most important non-functional properties of a software architecture
  - Changeability
  - Interoperability
  - Efficiency
  - Reliability
  - Testability
  - Reusability

Documenting Pattern – Pattern Template

- The **Pattern Template** *determines the style and structure of the pattern description*, and these vary in the emphasis they place on different aspects of pattern.
- The differences between pattern templates may mirror variations in the problem domain but there is no consensus as to the most appropriate template even within a particular problem domain.

Template Content

- A pattern description should include the following elements:
  - **Name** – Meaningful that reflects the knowledge embodied by the pattern
  - **Problem** – Description of the problem that the pattern addresses (the intent of the pattern).
  - **Context** – Represents the circumstances or preconditions under which it can occur.
  - **Forces** – Embodied in a pattern are the constraints or issues that must be addressed by the solution
  - **Solution** – Description of the static and dynamic relationships among the components of the pattern

Other Aspects of Templates

- A Pattern template may more extensive. Some other features that have figured in pattern template are:
  - An example of the use of a pattern that serves as a guide to its application
  - The context that results from the use of the pattern
  - The rationale that justifies the chosen solution
  - Related patterns
  - Known uses of the pattern that validate it
  - A list of aliases for the pattern
  - Sample program code and implementation details

Type of Design Patterns

- In the catalogue of 23 design patterns presented by Gamma et al. (1995) patterns are classified according to their scope and purpose.

  The 3 main categories of purpose that a pattern can have:
  - Creational
  - Structural
  - Behavioural

Type of Design Patterns

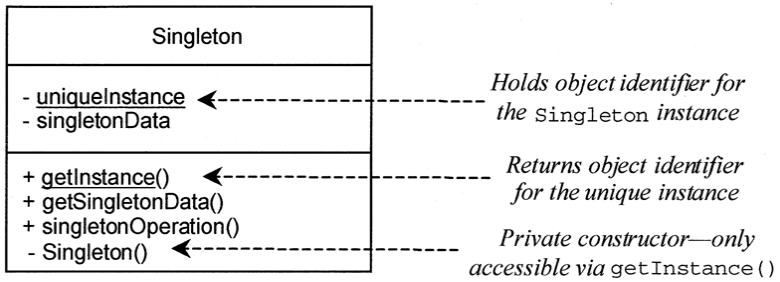
- Typically address issues concerning **Changeability**, and involve several different aspects:
  - Maintainability
  - Extensibility
  - Restructuring
  - Portability

Creational Patterns

- Concerned with the construction of object instances
- Separate the operation of an application from how its objects are created
- Gives the designer considerable flexibility in configuring all aspects of object creation
- This configuration might be dynamic (at run-time) or static (at compile-time)

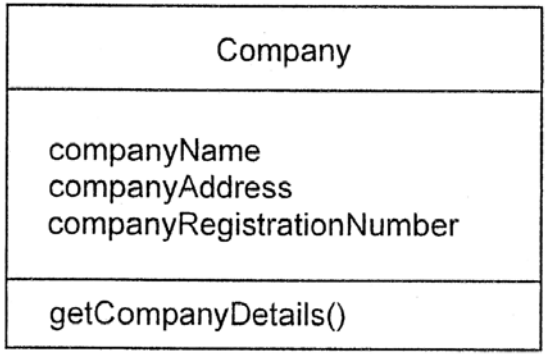
Creational Patterns: Singleton

- Singleton can be used to ensure that only one instance of a class is created.
- The class constructor in the Singleton pattern is private so that it can only be accessed the class-scope instance() method. This ensures that the Singleton class has total control over its own instantiation



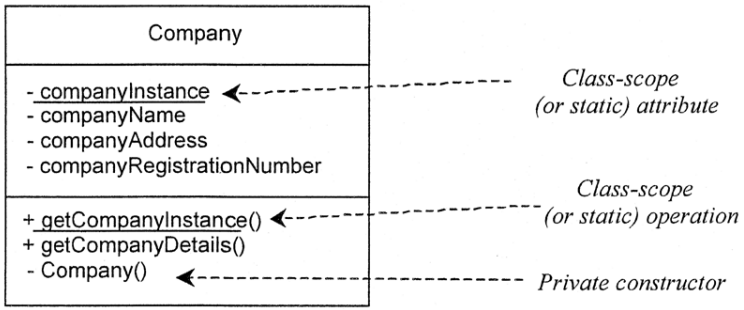
Singleton Example

- How does one ensure that only one instance of the company class is created for following class?



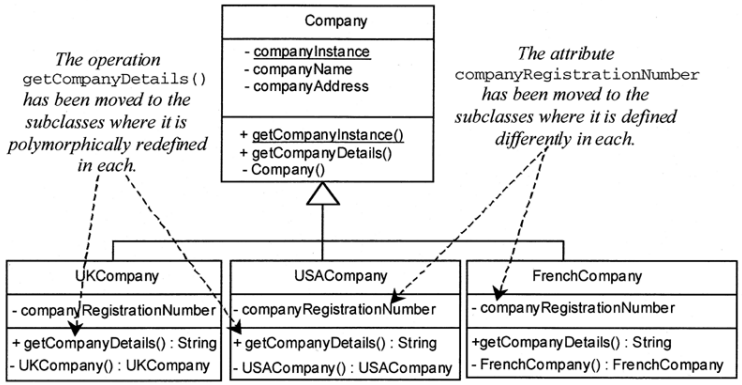
Singleton Example Solution

- Restrict access to the constructor
- The use of class-scope operations allows global access.



Singleton Example

- Different subclasses of Company can be instantiated as needed, depending on run-time circumstances



Advantages of Singleton

- It provide controlled access to the sole object instance as the Singleton class encapsulates the instance.
- The namespace is not unnecessarily extended with global variables.
- The Singleton class may be sub-classed. At system start-up user-selected options may determine which of the subclass is instantiated when the Singleton class is first accessed
- A variation of this pattern can be used to create a specified number of instances if required.

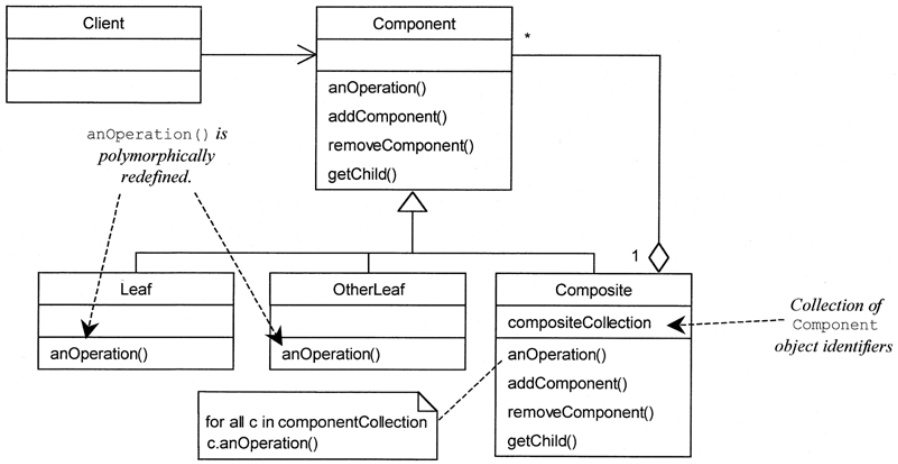
Disadvantages of Singleton

- Using the pattern introduces some additional message passing. To access the singleton instance the class scope method has to be accessed first rather than accessing the instance directly
- The pattern limits the flexibility of the application. If requirements change and as a result the singleton class may have many instances then accommodating this new requirement necessitates signature modification to the system.
- The singleton pattern is quite well known and developers are tempted to use it in circumstances that are inappropriate. Patterns must be used with care.

Structural Patterns

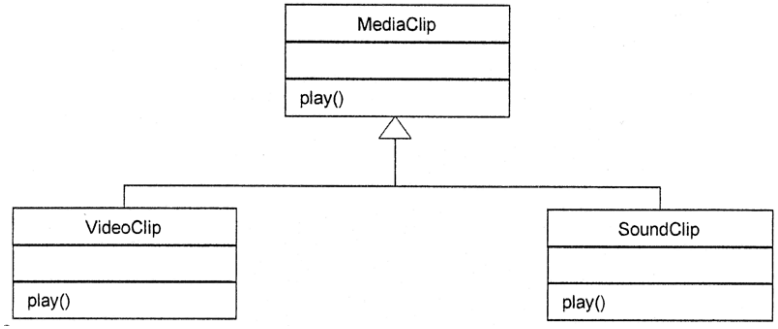
- Concerned with the way in which classes and objects are organized
- Offer effective ways of using object-oriented constructs such as inheritance, aggregation and composition to satisfy particular requirements

Composite Pattern General Form



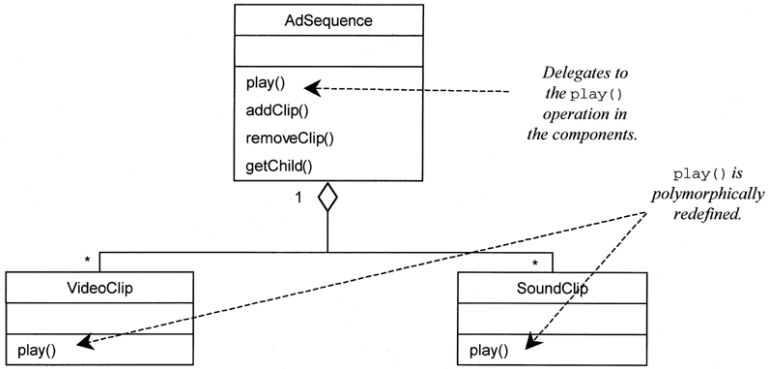
Composite Example

- How can we present the same interface for a media clip whether it is composite or not?

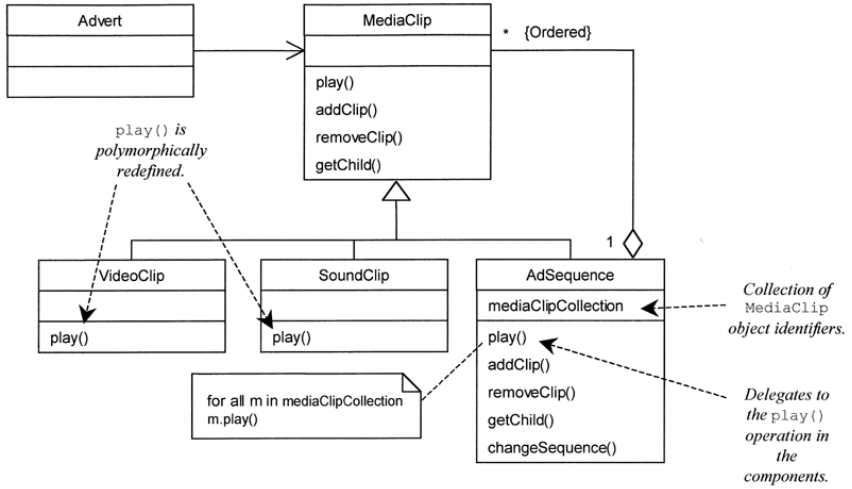


Composite Example

- How can we incorporate composite structures?



Composite applied to Agate

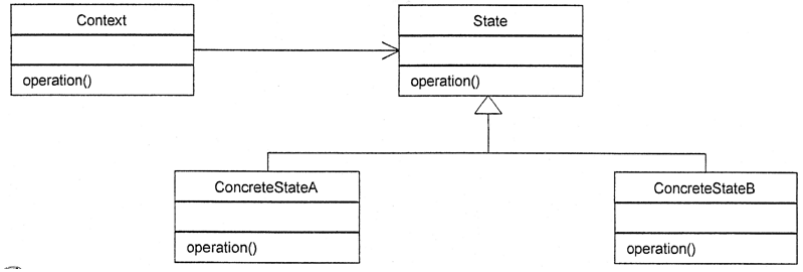


Behavioural Patterns

- Address the problems that arise when assigning responsibilities to classes and when designing algorithms
- Suggest particular static relationships between objects and classes and also describe how the objects communicate

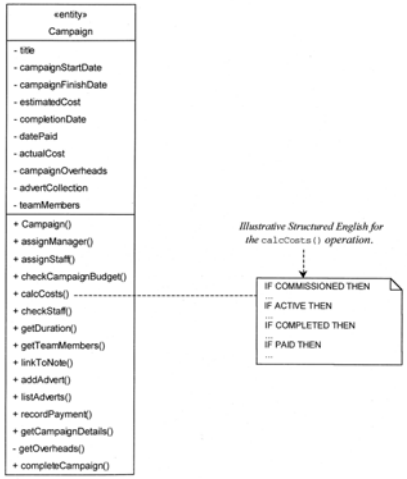
General form of State Pattern

- The State Pattern use both of these techniques:
  - Use inheritance structures to spread behavior across the subclasses
  - Use aggregation and composition to build complex behavior from simpler component



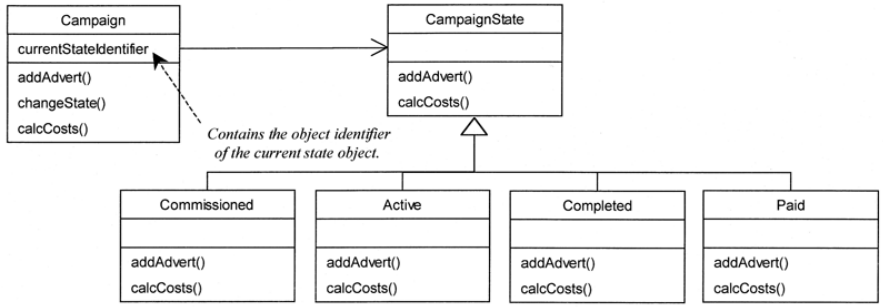
State Example

- Consider the class Campaign.



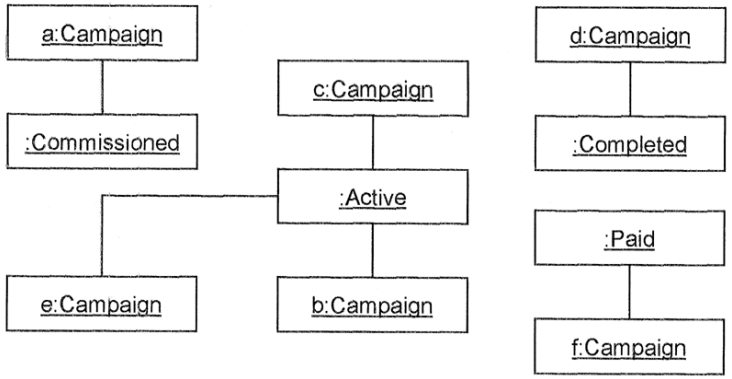
State Example

- State pattern applied to the class Campaign



State Example

- Some State pattern objects for Agate – note that there are 6 Campaign objects sharing the four State objects.



Explanation of State Example

- It has four states – Commissioned, Active, Completed and Paid
- A Campaign object has different behaviour depending upon which state it occupies
- Operations have case statements giving this alternative behaviour
- The class factored into separate components – one for each of its states

Implementation Problems of State Pattern

- The State pattern may have the following implementation problems.
  - If State objects cannot be shared amongst Context objects (i.e. are not pure state objects) then each Context object will have to have its own State object thus increasing storage requirements.
  - State objects may have to be created and deleted as the Context object changes state increasing the processing requirement.
  - The use of the State pattern introduces one additional message, which also possibly increases the processing requirement.

How to use Design Patterns?

- Before using a pattern to resolve the problem ask
  - Is there a pattern that addresses a similar problem?
  - Does the pattern trigger an alternative solution that may be more acceptable?
  - Is there a simpler solution? Patterns should not be used just for the sake of it
  - Is the context of the pattern consistent with that of the problem?
  - Are the consequences of using the pattern acceptable?
  - Are there constraints imposed by the software environment that would conflict with the use of the pattern?

Effective Use of Patterns

- The 7 steps from Gamma et al. are as follows.
  - Read the pattern to get a complete overview
  - Study the Structure, Participants and Collaborations of the pattern in detail
  - Examine the Sample Code to see an example of the pattern in use
  - Choose names for the pattern's participants (i.e. classes) that are meaningful to the application
  - Define the classes
  - Choose application specific names for the operations
  - Implement operations that perform the responsibilities and collaborations in the pattern

Dangers and Benefits of the Use of Pattern

- Two general dangers of using patterns are *the inappropriate application of a pattern* and *some limitation on creativity*.
- Two advantages of using patterns are *the introduction of a reuse culture at the design level* and *the rich source of development experience they offer*.