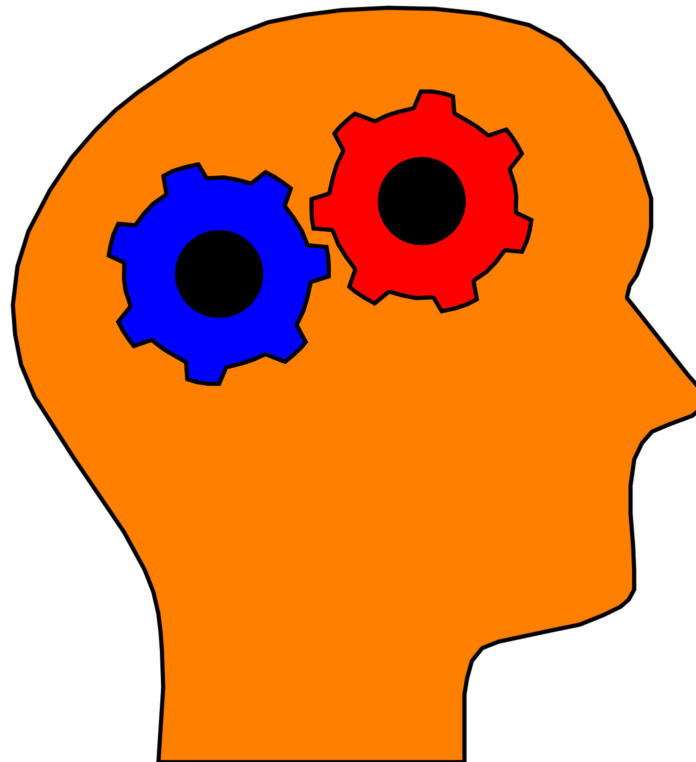
A Design Perspective



Maurizio Martignano
Copyright © 2007

Design Perspective - Introduction

- This section contains some considerations about common (Object Oriented) Design Principles.
- Once the features of a programming language are known, the main point is to see when and how it is convenient to use them.
- For each principle some indications about possible implementation in C/C++/Java are given.

Maurizio Martignano
Copyright © 2007

Design Perspective - Programming to an Interface (C++/JAVA/C#)

- When inheritance is used carefully, all classes derived from an abstract (pure virtual) class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class.
- All subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

  (See example c2cpp4.cxx)

Maurizio Martignano
Copyright © 2007

Design Perspective - Programming to an Interface (C++/Java/C#) (cont)

- There are two benefits to manipulating objects solely in terms of the inheritance defined by abstract classes:
  - clients remain unaware of the specific types of the objects they use, as long as the objects adhere to the interface that clients expect;
  - clients remain unaware of the classes that implement these objects; clients only know about the abstract class(es) defining the interface.

Maurizio Martignano
Copyright © 2007

Design Perspective - Programming to an Interface (Java/C#) (cont)

- In Java and C# it is possible to use “interfaces” instead of “abstract classes”.
- The advantage of using interfaces is that they hide completely the implementation (the same is not true for abstract classes -- why?), see example MGCalc.

Maurizio Martignano
Copyright © 2007

Design Perspective - Inheritance vrs. Aggregation/Composition

- The two most common ways of achieving functionality reuse are:
  - **class inheritance**, a.k.a. white-box reuse;
  - **object aggregation/composition** (I.e. embedding /linking an object to another one), a.k.a. black-box reuse.

(See example c2cpp4.cxx and C++ exercise 4)

Maurizio Martignano
Copyright © 2007

Design Perspective - Inheritance vrs. Aggregation/Composition (cont)

- The advantages of inheritance are:
  - it does not require a lot of re-writing;
  - it is easier to modify the implementation being reused.
- The disadvantages are:
  - parent classes often define at least part of their subclasses' physical representation;
  - inheritance exposes a subclass to details of it's parent implementation (I.e. it breaks encapsulation);
  - the implementation of a subclass may become so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclasses to change;
  - testing is difficult.

Maurizio Martignano
Copyright © 2007

Design Perspective - Inheritance vrs. Aggregation/Composition (cont)

- The advantages of object aggregation/composition are:
  - it requires objects to respect each others' interfaces (encapsulation is not broken);
  - it "produces" fewer implementation dependencies;
  - testing is easier.
- The disadvantages are:
  - it is not possible to define interfaces like the abstract classes;
  - when objects are embedded/linked into others the memory handling inside constructors (and destructors for C++) has to be designed properly;
  - it requires more re-writing.

Maurizio Martignano
Copyright © 2007

Design Perspective - Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its delegate (See C++ exercise 4).
- Delegation requires re-writing and may create long chains of function calls.
- Delegation is the normal way of using Java interfaces, see example Adventure.

Maurizio Martignano
Copyright © 2007

Design Perspective - Inheritance vrs. Templates

- Inheritance lets you provide default implementations for operations and lets subclasses override them.
- Templates let you change the types that a class (a function) can use.

Maurizio Martignano
Copyright © 2007

Design Perspective - Toolkits and Frameworks

- A **toolkit** is a set of related and reusable classes designed to provide useful, general-purpose functionality (e.g. the STL).

- **Toolkits don't impose a particular design** on your application; they just provide functionality that can help your application do its job.

Maurizio Martignano
Copyright © 2007

Design Perspective - Toolkits and Frameworks (cont)

- A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software (e.g. building graphic editors).

- **The framework dictates the architecture** of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibility thereof, how the classes and objects collaborate, and the thread of control.

Maurizio Martignano
Copyright © 2007

Design Perspective – Patterns Definitions

- A pattern is a generalised solution to a particular type of problem.
- Everywhere that we face that same type of problem, we can use the same solution.
- In addition, a pattern typically provides a flexible and robust solution.
- Using patterns enables designers to arrive at a solution much quicker than if they had to devise the solution themselves.
- Design patterns also enable inexpert designers to develop solutions that are just as flexible and powerful as those from expert designers with many years of experience.

Maurizio Martignano
Copyright © 2007

Design Perspective – Patterns History

- Patterns are a recent software engineering problem-solving discipline that emerged from the object-oriented community. Patterns have roots in many disciplines, including literate programming, and most notably in Alexander's work on urban planning and building architecture (Alexander, 1977).
- The goal of the pattern community is to build a body of literature to support design and development in general. There is less focus on technology than on a culture to document and support sound design. Software patterns first became popular with the object-oriented Design Patterns book (Gamma et al., 1995).

Maurizio Martignano
Copyright © 2007

Design Perspective – Patterns History (cont)

- But patterns have been used for domains as diverse as development organization and process, exposition and teaching, and software architecture. At this writing, the software community is using patterns largely for software architecture and design ...
- Today, the pattern discipline is supported by several small conferences, by a broad spectrum of activities at established software engineering conferences, and by a rapidly growing body of literature.

Maurizio Martignano
Copyright © 2007

Design Perspective – Patterns: The GoF Book

- Strongly suggested reading:

  Gamma, Helm, Johnson and Vlissides “Design Patterns (Elements of Reusable Object-Oriented Software), Addison Wesley, 1995, ISBN 0-201-63361-2.

Maurizio Martignano
Copyright © 2007

Design Perspective – Describing Design Patterns

- Pattern Name – the name of the pattern
- Intent – what does the pattern do
- A.k.a – also known as
- Motivation – an illustrating scenario
- Applicability – when, where the pattern can be applied
- Structure – a graphic representations of the patterns
- Participants – the classes / objects participating in the pattern
- Collaborations – how the participants collaborate to carry out their responsibility

Maurizio Martignano
Copyright © 2007

Design Perspective - Describing Design Patterns (cont)

- Consequences – how does the pattern supports its objectives? What are the trade-offs and result
- Implementation – what pitfalls, hints, techniques, should you be aware of
- Sample code
- Known uses
- Related Patterns

Maurizio Martignano
Copyright © 2007

Design Perspective - Patterns Classification

- Creational Patterns
  - Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - Builder – Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Factory method – Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
  - Prototype – Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - Singleton – Ensure a class only has one instance, and provide a global point of access to it.

Maurizio Martignano
Copyright © 2007

Design Perspective - Patterns Classification (cont)

- Structural patterns
  - Adapter – Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Bridge – Decouple an abstraction from its implementation so that the two can vary independently.
  - Composite – Compose objects into three structures to represent the part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.
  - Decorator – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Maurizio Martignano
Copyright © 2007

Design Perspective - Patterns Classification (cont)

- Facade – Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight – Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy - Provide a surrogate or a placeholder for another object to control access to it.

Maurizio Martignano
Copyright © 2007

Design Perspective - Patterns Classification (cont)

- Behavioural Patterns
  - Chain of Responsibility – Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - Command – Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.
  - Interpreter – Given a language, define a representation for its grammar along with an interpreter that uses that representation to interpret sentences in the language.
  - Iterator – Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Maurizio Martignano
Copyright © 2007

Design Perspective - Patterns Classification (cont)

- Mediator – Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento – Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later.
- Observer – Define a one-to-many dependency between objects so that when one object changes state, all its dependent are notified and updated automatically.
- State – Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Maurizio Martignano
Copyright © 2007

Design Perspective - Patterns Classification (cont)

- Strategy – Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently form the clients that use it.
- Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.
- Visitor – Represent the 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.

Maurizio Martignano
Copyright © 2007

Design Perspective - Summary

- Programming to an interface
- Inheritance vrs. Aggregation/Composition
- Delegation
- Inheritance vrs. Templates
- Toolkit and Frameworks
- Design Patterns

Maurizio Martignano
Copyright © 2007

Design Perspective - Further Reading

- Geroge Wilkie, “Object-Oriented Software Engineering”, Addison Wesley, 1993, ISBN 0-201-62767-1.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software”, Addison Wesley, 1994, ISBN 0-201-63361-2.

Maurizio Martignano
Copyright © 2007