Unit 2

Component Libraries and Design Heuristics

Topics

In this unit we will address some concepts that are used in the design of object-oriented applications:

- *Coding idioms*—forms of expression that are characteristic and peculiar to a particular programming language, such as C++
- *Class libraries*—sets of components that can be (re)used in the design of a system
- *Design patterns*—good solutions to recurring problems in object-oriented design
- *Architecture*—the structure of an application in terms of the components, their interfaces, and their interconnections. Within this context, we will consider *object-oriented frameworks*.

Topics

- Object-Oriented Design Concepts
  - Coding idioms,
  - Libraries,
  - Design patterns, and
  - Architectures (including frameworks)
- Interaction Diagrams

Coding Idioms

A *coding idiom* comprises a form of expression that is characteristic and peculiar to a particular programming language, such as C++. You might be familiar with some idioms in C—for example, looping over all the elements of an array using pointer arithmetic, or the use of getchar() and EOF to read all the characters in a text file. Idioms play a role in design and implementation because they provide effective algorithms for performing certain operations in a program.

C++ has some commonly used idioms. The ones cited here have to do with program organization—putting a specification for class C in header file C.h, with suitable #define's to guard against multiple inclusion, and putting the member functions and class data members in file C.cpp—and with the fact that in C++, it's usually good to examine every class C with respect to default constructors, copy constructors, assignment operators, and destructors, which should ordinarily be defined as "virtual."

Some idioms apply across programming languages, such as the use of reference counting to effect garbage collection.

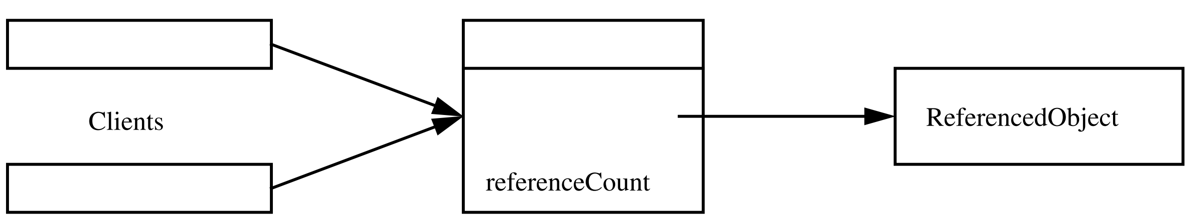
Coding Idioms

An *idiom* expresses how to implement a specific functionality within a particular programming language.

Often the same idiom looks different in different programming languages. Sometimes an idiom that is useful in one language is not needed or does not make sense in another.

Examples of C++ coding idioms are:

- Program organization:
  - .h and .cpp files
- Orthodox canonical form:
  - default constructor,
  - copy constructor,
  - assignment operator,
  - virtual destructor
- Reference counting is an idiom used in languages that do not support garbage collection:



- See James Coplien's book, *Advanced C++: Programming Styles and Idioms,* for a discussion of C++ coding idioms.

Object-Oriented Libraries

Class libraries comprise of components that can be (re)used in the design of a system. The use of components from a library can significantly reduce the design, coding, and testing efforts required in developing an application. A variety of class libraries are available commercially. Similarly, many companies put extra effort in developing a proprietary class library whose elements represent the focus of their business.

Sometimes requirements specify libraries to be used (or even developed), but frequently a library is used in the design of a system because it is “standard” in some environment. For example, we are developing *Brickles* as an application to be run under Microsoft Windows. Consequently, we would be wise to consider the use of the Microsoft Foundation Classes (MFC) since they provide a set of components that will not only make the development in the Windows environment easier, but will ensure some level of forward compatibility with updates to Windows. Some of the classes from MFC that you will see used in the design and implementation of *Brickles* ar listed at the bottom of the page. You can find more information about each of them in the Visual C++ “Help” menu.

Object-Oriented Libraries

- A *component* is a reusable class or cluster of classes designed to provide some general purpose functionality.
- Component libraries are the object-oriented equivalent of subroutine libraries; they emphasize *code* reuse.
- Components do not impose a particular design on your application—they just provide functionality such as:
  - Lists, stacks, queues, arrays, …
  - Windows, dialog boxes, buttons, menus, …
  - Vectors, matrices, …
  - Rectangles, Circles, Arcs, Lines, …
  - …
- Examples of component libraries:
  - C++ components from RogueWave
  - Object Windows Library (OWL) from Borland
  - Microsoft Foundation Classes (MFC) from Microsoft. Some MFC classes used in the design of *Brickles*:

CPoint	CDocument	CObject	CBitmap
CSize	CView	CException	CRect
CObjectList	CWinApp	CFrameWnd	

Design Patterns

Design patterns represent good solutions to recurring problems in object-oriented design. They capture these solutions in a “succinct and easily applied form.” You probably already have catalogued and applied your own patterns in your work. Experience working in some problem domain gives you an understanding of problems that occur and what solutions exist. You might be given a problem to solve that you are able to liken to something you’ve solved before. The solution might have to adapt a bit, but it follows the same pattern.

Design Patterns

- In 1977, Christopher Alexander, an architect, published *A Pattern Language*. Alexander was the first to study patterns in buildings and communities. He developed a "pattern language" for describing and generating them. Alexander believed these patterns captured the "aliveness" that certain buildings, places, and human activities have.
- Alexander emphasized that in a good design there must be an underlying correspondence between the structure of the problem and the structure of the solution.
- Alexander's work has inspired a number of computer scientists to consider whether "patterns" exist in software systems:
  - Erich Gamma's doctoral thesis described the patterns incorporated in ET++, a GUI application framework.
  - Since 1991, workshops have been held each year in conjunction with OOPSLA to discover and document design patterns.
  - The first conference on Pattern Languages of Programs was held in August 1994.
  - The book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides (the "gang of four") was published in 1995. It describes twenty-three design patterns.

Design Patterns

Design patterns comprise a set of simple solutions to problems. A typical object-oriented design will apply many different patterns, and a single pattern might be applied many times.

Each pattern is given a name by which it can be referred—“We can use *Observer* here.” We are used to having names for algorithms—“heap sort” and “Newton’s method”—but not names for parts of designs. Each pattern also describes a problem and the solution, as well as the consequences of applying the solution in terms of trade-offs with respect to performance, extensibility, flexibility, and reusability. A pattern is usually documented with a solution (or set of alternative solutions) in some programming language.

The identification and documentation of patterns by the gang of four and by others has made a big contribution to the design of systems by providing new designers with a way to learn good object-oriented design based on the experience of others.

Note that these design patterns are derived from many years of experience that refined the solutions expressed in the patterns and proved them out.

Design Patterns

- Design patterns are descriptions of communicating objects and classes that occur in many systems and that are customized to solve a general design problem in a particular context.
- In general, a pattern has four essential elements:
  - A *pattern name* that provides a handle and lets us design at a higher level of abstraction
  - A *problem* that describes when to apply a problem, including a context.
  - The *solution* that describes the elements—classes and/or objects—that make up the design, their relationships, their responsibilities, and their collaborations.
  - The *consequences* of applying the pattern—space/time trade-offs, flexibility, extensibility, and so on.
- Design patterns are “reusable micro-architectures that contribute to an overall system architecture” that provide a common vocabulary for design.
- Advantages:
  - A common design vocabulary
  - A documentation and learning aid
  - Capture the experience of expert designers
  - Helps avoid *refactoring*—the need to change code in order to reuse it

A Few Patterns

These are summary descriptions of some of the patterns that have been applied in the design of *Brickles* used in this course. We will look at one in detail. The rest are given in Appendix B. We will refer to them as the design is explained.

A Few Patterns

Some patterns considered and applied in the design of *Brickles* are:

- *Singleton*. Ensure a class has only one instance, and provide a global point of access to it.
- *Observer*. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- *Composite*. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- *Visitor*. 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.

Singleton

The Singleton pattern is used to ensure a class has only one instance and provide a global point of access to it. This situation can occur, for example, in systems in which a single component controls access to another part of a system—for example, a storage manager coordinates all accesses to a database. If we have a StorageManager class to define that component, then we must ensure that only a single instance is ever created.

The solution involves a class method, Instance(), that returns the sole instance of the class (creating it, if necessary). The class maintains a pointer, uniqueInstance, to the single instance. The operations associated with the single object, as well as the data, are defined as members of the class.

A C++ implementation is given next.

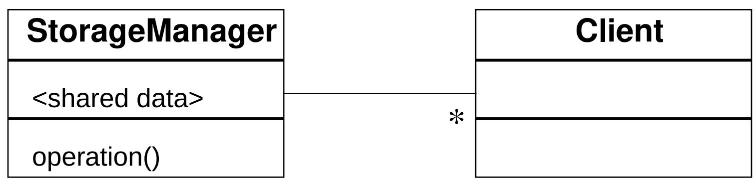
Singleton

- ## Intent

Ensure a class has only one instance and provide access to it.

- ## Motivation

It is sometimes critical that there only be one instance of a class. Certain managers, such as storage managers, that coordinate access to shared resources must be the exclusive agent in the system.

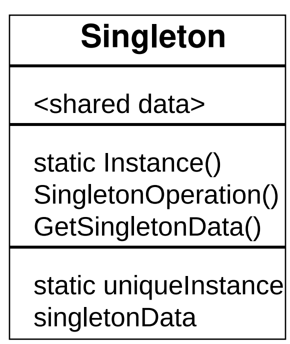


- ## Applicability

Use the Singleton pattern when there must be exactly one instance of a class, and it must be accessible from a single point of access.

Use the Singleton pattern when the instance must be extensible by subclassing.

- ## Structure



- ## Participants

The Singleton class that provides a method that returns the single instance.

The Singleton class produces the instance.

- ## Consequences

Benefits: Guarantee of single instance or any specific number of instances; avoids global variables

Risks: An additional small class that must be managed

Implementing Singleton

This is only one possible implementation for Singleton in C++ (what is another?). It makes use of static data members and static member functions.

Note that the default constructor is protected to force the use of the "class method" Instance to get a "new" instance of the singleton.

Implementing Singleton

```
// Singleton.h
#ifndef NOWHERE_H
#define NOWHERE_H
class Singleton {
  public:
    static Singleton* Instance();
  protected:
    Singleton();
  private:
    static Singleton* _instance;
};
#endif



// Singleton.cpp
#include "Singleton.h"

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance(){
  if(_instance == 0){
    _instance = new Singleton;
  }
  return _instance;
}

Singleton::Singleton() {}
```

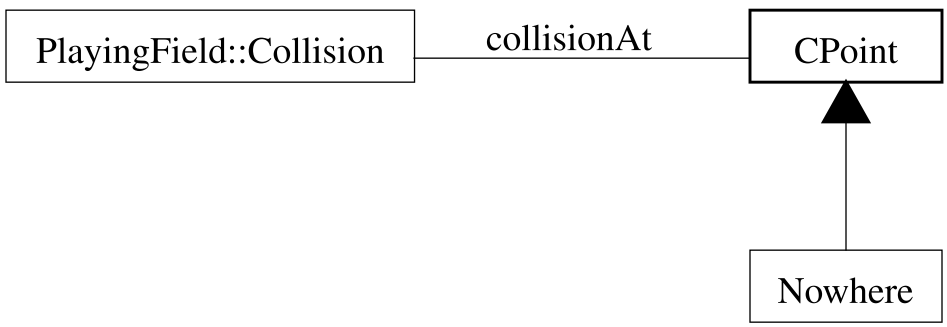
Applying Singleton

In Brickles, we use a mouse to control the paddle. We can use only a single instance of a Mouse class, and therefore could apply the Singleton pattern to get the desired behavior.

We also will want a distinguished point (instance of MFC class CPoint) to designate “nowhere” on a playing field—for example, we might want to compute collision point between two objects, but a collision might not have occurred. We could use “0”—a.k.a. “NULL”—but that would require the use of pointers to CPoint’s everywhere. Singleton can help us define such a point.

Applying Singleton

- In checking for a collision, the ArcadeGamePiece::isAlong method returns either an instance of CPoint, which could be the single instance of class Nowhere.
- NoWhere is a Singleton class. Only a single instance of NoWhere exists.



Architecture

Architecture refers to the structure of a software system in terms of the components, their interfaces, and their interconnections. There are many kinds of architectures, and a system might use combinations of several kinds. Some examples of software architectures are:

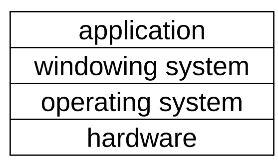
- Layered architecture—a hierarchal architecture where each layer provides services to the layer above. Each layer has knowledge of the one(s) below it, but has no knowledge about any layer above it.
- Partitioned architecture—a division of a system into a set of vertical components that are either independent or weakly coupled. Each component provides one kind of service. The partitions might know about one another, but this knowledge is not deep.
- Client-Server architecture. An architecture used to partition an application among a number of processors such that a server works independently of the clients that use it.

Architectures

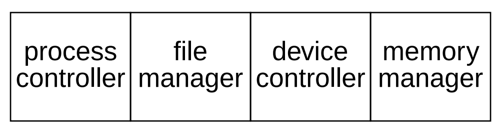
- A dictionary definition of *architecture* is:
  - "The science, art, or profession of designing and constructing"
  - "A style of design"
- In software development, architecture expresses a fundamental paradigm for structuring a system. It provides a set of predefined subsystems, as well as rules and guidelines for organizing the relationships between them.

  Examples of software architectures are:

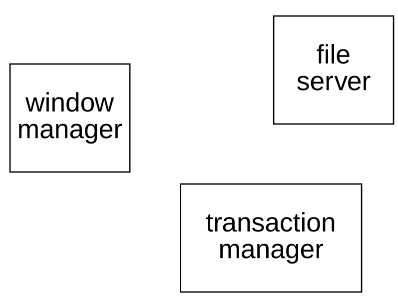
  - Layered architecture



  - Partitioned architecture



  - Client-server architecture



Frameworks

Some architectures are reusable in the design of many applications in the sense that these applications—or portions of them—are based on the same underlying structure. In an object-oriented design, we could recognize some set of objects that have the same general set of responsibilities and interactions.

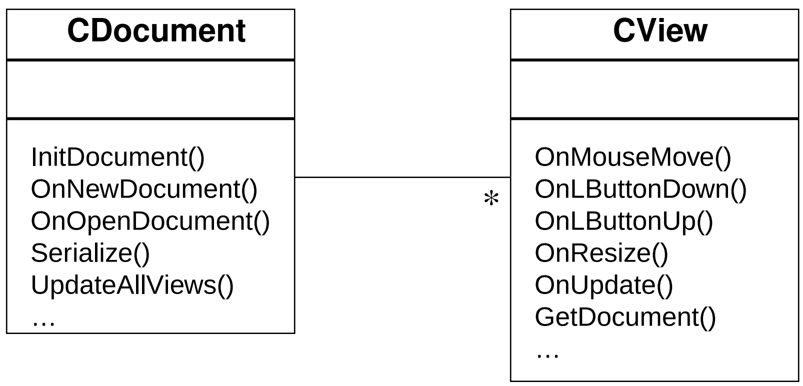
A "reusable design" is referred to as a *framework*. An object-oriented framework comprises a set of abstract classes that establish a set of relationships and responsibilities used in the design of similar application. Such a framework is analogous to a application generator in the procedural paradigm. [An application generator is a program that generates programs, and all the programs it generates have the same underlying structure (design) although each does something different.]

Examples of well-known object-oriented frameworks are Model-View-Controller (from Smalltalk) and Document-View (from MFC). These frameworks are very similar and are quite general in their applicability. They can be used for most applications and work by distributing responsibilities for input, output, and processing to various objects. In the case of Document-View, a document is responsible for holding data and processing it. A view is responsible for accepting user input events, directing them to the document to which the view is attached, and then updating the display of the document when it has changed significantly. A document is responsible for updating all views when it has changed in some significant way. The mechanisms for registering views with a document, and for updating views are provided by the CDocument class. The default mechanisms for handling input events (mouse moves, mouse button presses, etc.) and hooks for updating views are provided by CView.

A framework is used by creating subclasses of the framework classes and then overriding methods and adding other members as necessary to implement the application.

Frameworks

- An object-oriented *framework* is a reusable design and a set of components that set it up.
- The Microsoft Foundation Classes (MFC) are organized around a Document-View architecture that is an adaptation of the Model-View-Controller architecture used in the Smalltalk environment.



- “At run time, an application written with the Microsoft Foundation Class Library (MFC) is a group of cooperating objects that communicate by sending and receiving Windows messages and by calling each others’ member functions.”
  - A *document* defines, stores, and manages an application’s data. A user manipulates a document through a view attached to a document.
  - A *view* manages user interactions with a document, displaying data in a document and supporting input, selection, and editing of the document

Object-Oriented Design

Object-oriented design is "problem-centered" is the sense that we try to design the software to use the objects that appear in the problem (as defined in the analysis object model). Thus, the solution structure derives from the problem structure. This approach derives from the observation that in a given system, the problem changes less frequently than a solution, so if we design software based on the problem, the effort needed to rework it as solutions do more (or different) things will be reduced when compared to an architecture that is oriented toward a specific solution to a problem.

The solution is created from a model of a problem by giving the software entities that represent problem ("real-world") entities additional responsibilities—for example, in the design of a controller for the traffic intersection, a lane might be made responsible for stopping and starting the flow of traffic within it (by manipulating one or more signals), even though in the "real world" a lane can only hold traffic. So, in design we frequently animate objects in order to support a solution.

Since the design is derived from the problem structure, the correctness and completeness of analysis models is very important. We will consider the model for Brickles. Frequently, one or more classes in an analysis model will not be useful in design—for example, vehicles use intersections, but a vehicle object is not likely to be needed in implementing a traffic intersection controller. On the other hand, we frequently need to add new classes in designing a system. [Sometimes it turns out that such a class should have been put on the analysis model in the first place!] The challenge of object-oriented design lies in how to represent the model in the software, and in how to animate the various objects to solve the problem.

Object-Oriented Design

- An object-oriented design will be problem-centered, incorporating a model of the problem in the solution. The objects in the problem are enhanced with additional relationships—perhaps to new kinds of objects—and additional responsibilities that provide a solution. The challenges lie in:
  - Deciding which objects belong in the solution (Will we need a “Player” object in Brickles? Will we need some objects not identified in analysis?)
  - Deciding which associations (links) need to be implemented (Does a puck need to know what playing field it’s in?)
  - Defining class protocols (interfaces) (Are there any parameters needed when requesting a brick to break?)
  - Determining how inputs are processed to outputs—that is, how objects interact with one another to effect computations (How are collisions detected and how are they processed?)
- An object-oriented design must consider performance, reliability, maintainability, extensibility, and/or reusability
- An object-oriented design must also consider the application environment—for example, support for multitasking or distributed processing, and perhaps even the implementation language.

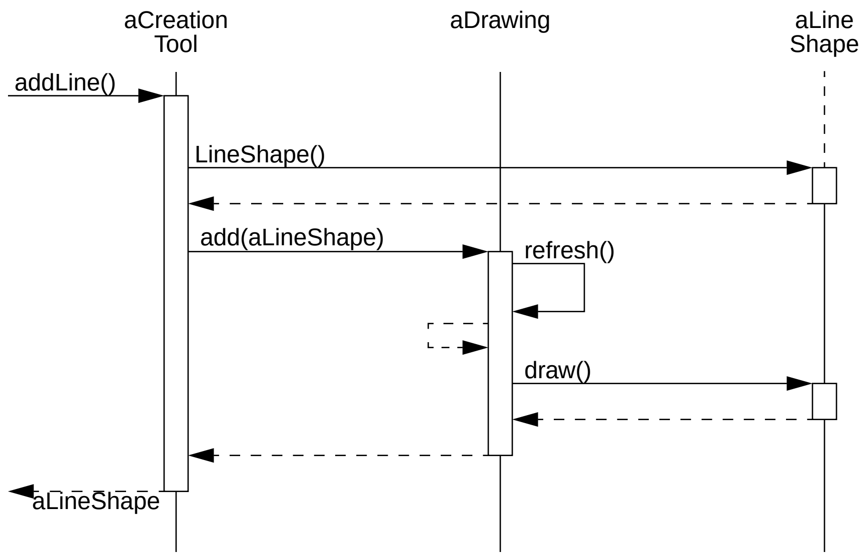
Interaction Diagrams

An *interaction diagram* shows the messages between objects that results from some stimulus. Time flows from top to bottom. A solid vertical line portrays the lifetime of an object whose label appears at the top. [A dashed vertical line is used before/after destruction.] A widened line shows that an object is active—that is, in the process of handling a message. A message is indicated by a horizontal arrow, labeled with the name of the request and (optionally) the parameters. Creation of an object is shown with a C++-style constructor notation—for example, `Rectangle(5,3)`. A message sent by an object to itself is shown by an arrow that points back to the object. A return is indicated by a dashed arrow, labeled, if applicable, with a return value. For example[1]:

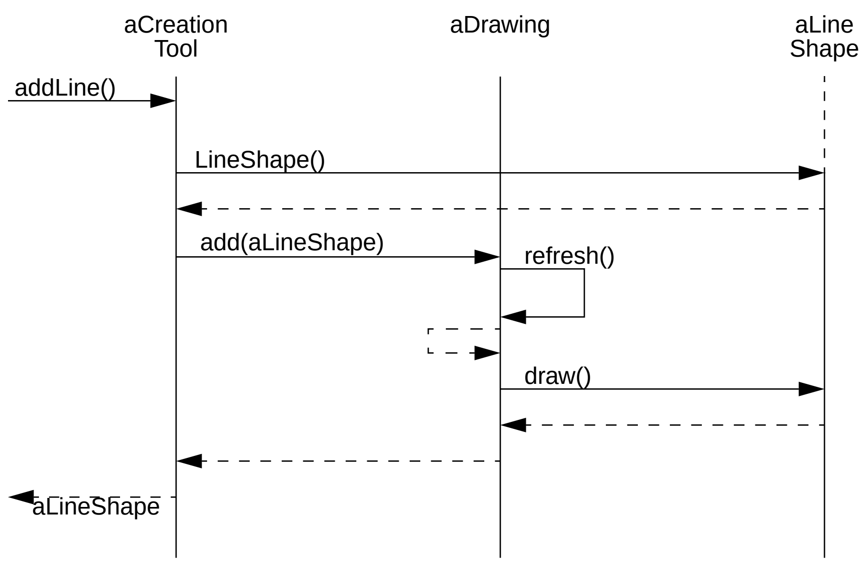
1. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995 [ISBN 0-201-63361-2].

Interaction Diagrams

- One tool used in object-oriented design are interaction diagrams that display the messaging between objects:



- NOTE: When drawing these in class, we will sometimes omit the widened line:



Summary

Take a few minutes to review the material in this unit. Note the three most important concepts from your perspective.

-
-
-

Summary

- Coding idioms are useful in design because we can make good use of the experience of others in finding effective ways of expressing certain computations (or concepts) in a given programming language.
- The availability of component (class) libraries can help to reduce the effort needed to build an application and design activities should consider the availability of such components.
- Design patterns are useful because we can build on the experience of others in finding good solutions to problems that come up again and again in object-oriented design.
- Object-oriented frameworks provide a mechanism for reusing a design (and part of the implementation of that design).
- There are many ways to structure an application. In object-oriented design, we strive to incorporate the structure of the problem to be solved.
- One useful tool in object-oriented design is the interaction diagram. These diagrams are useful both in creating the design for an application and in documenting the design of an application.