CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 19
November 11, 2010

Exceptions (cont.)

Design Patterns

Exceptions (cont.)

C++ exception mechanism (recall)

C++ exception mechanism is a means for a low-level routine to report an exception directly to a higher-level routine.

This separates exception-handling code from normal processing code.

An exception is reported using the keyword `throw`.

An exception is handled in a `catch` block.

Each routine in the chain between the reporter and the handler is exited cleanly, with all destructors called as expected.

Throwing an exception

`throw` is followed by an *exception* value.

Exceptions are usually objects of a user-defined exception type.

Example:

```
throw AgeError("Age can't be negative");
```

Exception class definition:

```
class AgeError {
  string msg;
public:
  AgeError(string s) : msg(s) {}
  ostream& printError(ostream& out) const { return out<< msg; }
};
```

Catching an exception

A try region defines a section of code to be monitored for exceptions.
Immediately following are catch blocks for handling the exceptions.

```
try {
   ...  //run some code
}
catch (AgeError aerr) {
    // report error
    cout<< "Age error: ";
    aerr.printError( cout )<< cout;
    // ... recover or abort
}
```

What kind of object should an exception throw?

`catch` filters the kinds of exceptions it will catch according to the type of object thrown.

For this reason, each kind of exception should throw it's own type of object.

That way, an exception handler appropriate to that kind of exception can catch it and process it appropriately.

While it may be tempting to throw a string that describes the error condition, it is difficult to process such an object except by printing it out and aborting (like `fatal()`).

Properly used, exceptions are much more powerful than that.

Standard exception class

The standard C++ library provides a polymorphic base class `std::exception` from which all exceptions thrown by components of the C++ Standard library are derived.

These are:

exception	description
`bad_alloc`	thrown by `new` on allocation failure
`bad_cast`	thrown by a failed `dynamic_cast`
`bad_exception`	thrown when an exception type doesn't match any catch
`bad_typeid`	thrown by `typeid`
`ios_base::failure`	thrown by functions in the `iostream` library

(from `http://www.cplusplus.com/doc/tutorial/exceptions/`)

Catching standard exceptions

Class `std::exception` contains a virtual function

```
const char* what() const;
```

that is overridden in each derived exception class to provide a meaningful error message.

Because the base class is polymorphic, it is possible to write a single `catch` handler that will catch all derived exception objects.

Example:

```
catch (exception& e)
  {
    cerr << "exception caught: " << e.what() << endl;
  }
```

Deriving your own exception classes from std::exception

```
#include <iostream>
#include <exception>
using namespace std;
class myexception: public exception {
  virtual const char* what() const throw()
    { return "My exception happened"; }
} myex;  // declares class and instantiates it
int main () {
  try {
    throw myex;
  }
  catch (exception& e) {
    cout << e.what() << endl;
  }
  return 0;
}
```

Multiple catch blocks

- Can have multiple `catch` blocks to catch different classes of exceptions.
- They are tried in order, so the more specific should come before the more general.
- Can have a "catch-all" block `catch (...)` that catches all exceptions. (This should be placed last.)

Rethrow

A `catch` block can do some processing and then optionally rethrow the exception or throw a new exception.

- One exception can cause multiple `catch` blocks to execute.
- To rethrow the same exception, use `throw;` with no argument.
- To throw a new exception, use `throw` as usual with an argument.

Note: Rethrowing the current exception is not the same as a new `throw` with the same exception object.

Throw restrictions

It is possible to specify that a function can only throw certain kinds of exceptions (or none at all).

This "feature" is regarded as a bad idea because the current semantics are not what one would expect.

It does not prevent the exceptions from being thrown; rather, it causes a run-time test to be inserted which calls `unexpected_exception()` when an exception is thrown that is not listed in the function's throw specifier.

Uncaught exceptions: Ariane 5

Uncaught exceptions have led to spectacular disasters.

The European Space Agency's Ariane 5 Flight 501 was destroyed 40 seconds after takeoff (June 4, 1996). The US$1 billion prototype rocket self-destructed due to a bug in the on-board guidance software. [Wikipedia]

This is not about a programming error.
It is about system-engineering and design failures.
The software did what it was designed to do and what it was agreed that it should do.

Uncaught exceptions: Ariane 5 (cont.)

Heres a summary of the events and its import for system engineering:

- A decision was made to leave a program running after launch, even though its results were not needed after launch.
- An overflow error happened in that calculation,
- An exception was thrown and, by design, was not caught.
- This caused the vehicle's active and backup inertial reference systems to shut down automatically.

As the result of the unanticipated failure mode and a diagnostic message erroneously treated as data, the guidance system ordered violent attitude correction. The ensuing disintegration of the over-stressed vehicle triggered the pyrotechnic destruction of the launcher and its payload.

Termination

There are various conditions under which the exception-handling mechanism can fail. Two such examples are:

- Exception not caught by any `catch` block.
- A destructor issues a `throw` during the stack-unwinding process.

When this happens, the function `terminate()` is called, which aborts the process.

This is a bad thing in production code.

Conclusion: *All exceptions should be caught and dealt with explicitly.*

Design Patterns

General OO principles

1. **Encapsulation** Data members should be private. Public accessing functions should be defined only when absolutely necessary. This minimizes the ways in which one class can depend on the representation of another.
2. **Narrow interface** Keep the interface (set of public functions) as simple as possible; include only those functions that are of direct interest to client classes. Utility functions that are used only to implement the interface should be kept private. This minimizes the chance for information to leak out of the class or for a function to be used inappropriately.
3. **Delegation** A class that is called upon to perform a task often delegates that task (or part of it) to one of its members who is an expert.

What is a design pattern?

A pattern has four essential elements.[1]

1. A *pattern name*.
2. The *problem*, which describes when to apply the pattern.
3. The *solution*, which describes the elements, relations, and responsibilities.
4. The *consequences*, which are the results and tradeoffs.

[1] Erich Gamma et al., *Design Patterns*, Addison-Wesley, 1995.)

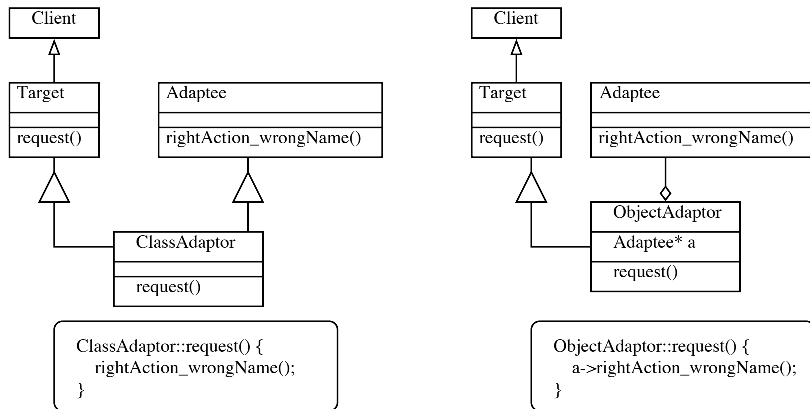
Adaptor pattern

Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires.

Solution: Define an adapter class that can add, subtract, or override functionality, where necessary.

Adaptor diagram

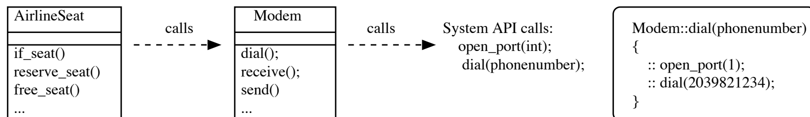
There are two ways to do this; on the left is a class adapter, on the right an object adapter.



Indirection

This pattern is used to decouple the application from the implementation where an implementation depends on the interface of some low-level device.

Goal is to make the application stable, even if the device changes.



Proxy pattern

This pattern is like Indirection, and is used when direct access to a component is not desired or possible.

Solution: Provide a placeholder that represents the inaccessible component to control access to it and interact with it. The placeholder is a local software class. Give it responsibility for communicating with the real component.

Special cases: Device proxy, remote proxy. In Remote Proxy, the system must communicate with an object in another address space.

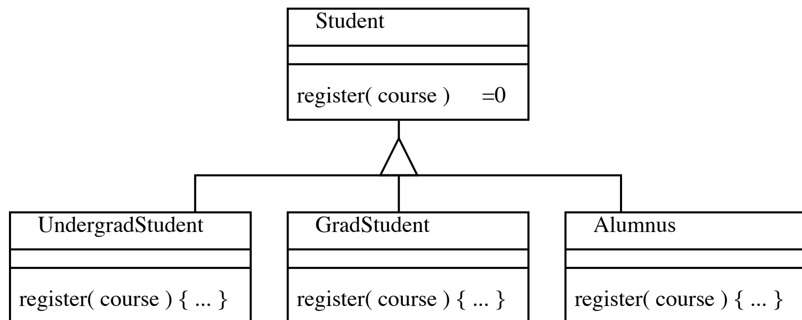
Polymorphism pattern

In an application where the abstraction has more than one implementation, define an abstract base class and one or more subclasses.

Let the subclasses implement the abstract operations.

This decouples the implementation from the abstraction and allows multiple implementations to be introduced, as needed.

Polymorphism diagram



Controller

A controller class takes responsibility for handling a system event.

The controller should coordinate the work that needs to be done and keep track of the state of the interaction. It should delegate all other work to other classes.

Three kinds of controllers

A controller class represents one of the following choices:

- The overall application, business, or organization (facade controller).
- Something in the real world that is active that might be involved in the task (role controller).
  Example: A menu handler.
- An artificial handler of all system events involved in a given use case (use-case controller).
  Example: A retail system might have separate controllers for BuyItem and ReturnItem.

Choose among these according to the number of events to be handled, cohesion and coupling, and to decide how many controllers there should be.

Bridge pattern

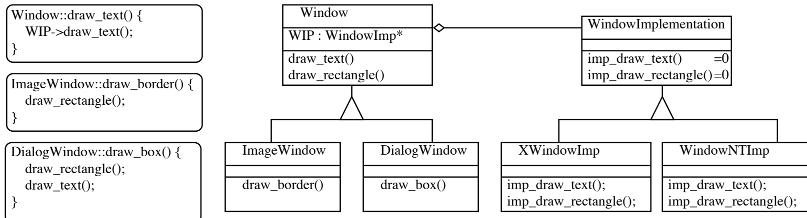
Bridge generalizes the Indirection pattern.

It isused when both the application class and the implementation class are (or might be) polymorphic.

Bridge decouples the application from the polymorphic implementation, greatly reducing the amount of code that must be written, and making the application much easier to port to different implementation environments.

Bridge diagram

In the diagram below, we show that there might be several kinds of windows, and the application might be implemented on two operating systems. The bridge provides a uniform pattern for doing the job.



Subject-Observer or Publish-Subscribe: problem

Problem: Your application program has many classes and many objects of some of those classes. You need to maintain consistency among the objects so that when the state of one changes, its dependents are automatically notified. You do not want to maintain this consistency by using tight coupling among the classes.

Example: An OO spreadsheet application contains a data object, several presentation "views" of the data, and some graphs based on the data. These are separate objects. But when the data changes, the other objects should automatically change.

Subject-Observer or Publish-Subscribe: pattern

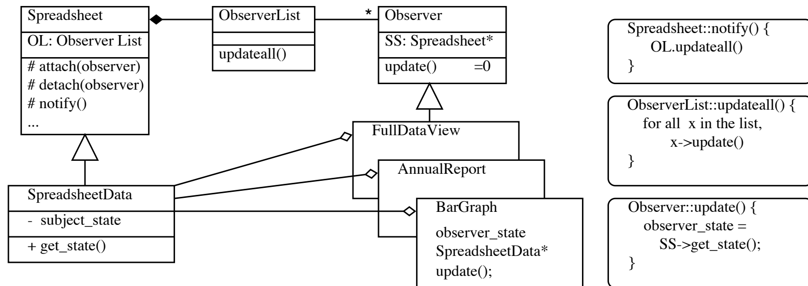
Call the SpreadsheetData class the **subject**; the views and graphs are the **observers**.

The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects.

Observer objects may be added to this list, as needed, and all will be notified when the subject (SpreadsheetData) changes.

We derive a concrete subject class (SpreadsheetData) from the Spreadsheet class. It will communicate with the observers through a `get_state()` function, that returns a copy of its state.

Subject-Observer or Publish-Subscribe: diagram



See textbook for more details.

Singleton pattern

Suppose you need exactly one instance of a class, and objects in all parts of the application need a single point of access to that instance.

Solution: A single object may be made available to all objects of class `C` by making the singleton a static member of class `C`.

A class method can be defined that returns a reference to the singleton if access is needed outside its defining class.

Example: Suppose several parts of a program need to use a StringStore. We might define StringStore as a singleton class.

The `StringStore::put()` function is made `static` and becomes a global access point to the class, while maintaining full protection for the class members.