Object-Oriented Principles and Practice / C++

Alice E. Fischer

June 24, 2013

STL

iterators (Chapter 13.7)
pair and map (Chapter 13.8)
algorithms (Chapter 13.7)

Iterators

- Every stl template class has its own kind of iterator.

```
#include <vector>
#include vector<Edge> edges;
#include vector<Edge>::iterator scan =
edges(begin);
```

- An iterator can start at the beginning of a collection, be incremented like a pointer in a loop, and go to the end of the collection.
- An iterator can also be used to point at something you took OUT of a collection. It works very much like a pointer but is not type-compatible with a pointer to the same base type.

Pairs

- `#include <utility>`
- A pair is a predefined STL template class. Pairs are used by various other stl classes for many diverse purposes.
- The template instantiation requires two type parameters, one for each member of the pair.
- The name of the type has the form `pair< string, Node>`
- A pair has two public members named `first` and `second`.
- To construct a pair, call `make_pair()` with two objects of the right types.

What is a Map?

We use a map to organize a data set that must be searched often.

- `#include <map>`
- A map is an associative container data structure.
- You store pairs in the map, where each pair consists of a data object and an indexing key.
- To retrieve a data object later, you execute find(key).
- The programmer can locate an object without writing a search loop.
- In C++, the map template is implemented efficiently, with O(log(n)) complexity for insertion and search. (Probably a red-black tree).

Imagine implementing Google Maps, with hundreds of thousands of cities (nodes). We want the city-nodes stored in an O(log(n)) container like a map.

Making a Map of Nodes

We use a `map<string, Node>` to store the Nodes in our Graph.

- To use the map, we need an iterator:

```
map<string, Node>::iterator p;
```

- We make pairs of string and a Node: `pair<string, Node>`
- Use one field of the Node as an index value.

```
string name = "Nan";
make_pair( name, Node(name) );
```

- `make_pair()` makes a pair, then returns a second pair consisting of the pair it made and a boolean success/failure code. To get an iterator pointing to the pair we want, select `.first`:

```
nodeMap.insert(item).first;
```

- For simplicity, in this program we ignore the success/failure code. Good practice would be to check it and call fatal() if it is false.

Using a Map of Nodes

We made the map so that we could easily and efficiently find any node in the graph, given its name.

- To locate a node, use map::find(), which returns an iterator to a pair:

```
map<string, Node>::iterator p;
p = nodeMap.find(name);
```

- If the node was not in the map, find() returns `nodeMap.end()`.

```
if (p == nodeMap.end()) { // name not found in map
```

- "Not found" is normal inside the Graph constructor but is a fatal error when it happens in the middle of an algorithm.

STL algorithms

Refer to cplusplus website: references, other.

- `#include <algorithm>`
  `#include <vector>`
- std::sort sorts any array, given an appropriate comparison function.
- The heap algorithms are defined and work with a specified comparison function on an object that has been declared as a `vector`. See PQueue.hpp

Fundamental Design Patterns

Encapsulation
Expert
Delegation
Narrow Interface
Creator
Low Coupling
High Cohesion
Don't Talk to Strangers
Polymorphism
Chain of Responsibility

Basic Principles for OO Design

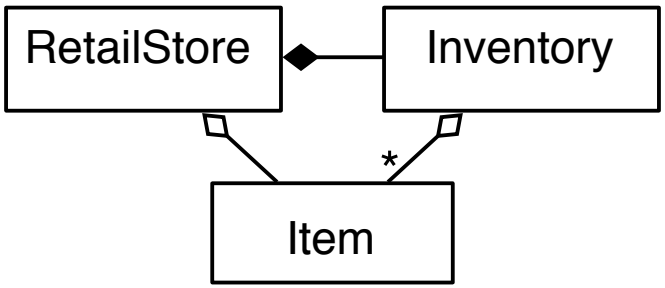
These are recognized as important fundamental design principles.

- Encapsulation: data members should be private. Accessors should be defined only when necessary.
- Expert: Each class should do for itself all actions that involve its data members.
- Delegation: Delegate all actions to the class that is the expert on the data.
- Narrow interface: Keep the set of public functions as simple as possible. Functions that are not needed by client classes should be private.
- Creator: Allocate and initialize an object in the class that composes, aggregates, or contains it.

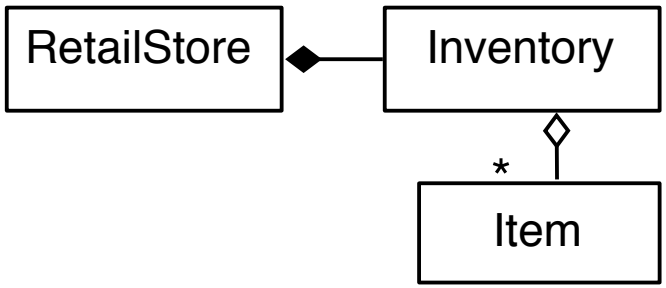
Low coupling

A UML diagram contains links between classes. The number of links should be minimized.

When assigning responsibility for a task to a class, assign it so that the placement does not increase coupling.



Bad: Unnecessary coupling



Good: Minimal coupling

High cohesion

A class should have a single, narrow purpose.

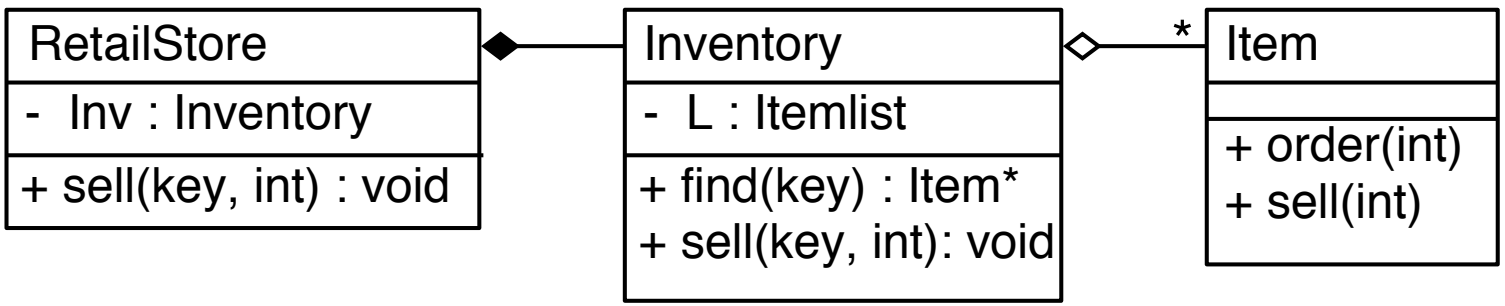
- The members of the class should be strongly related and focused on the purpose and responsibilities of the class.
- Don't let the classes "sprawl" by adding more and more specialized features
- Maintain the separation of purposes: define structural elements and semantic elements in separate classes.
- Example: If you want a linked list of books, define a book class and define a pair classes, List and Cell. Don't define the members of a book in the Cell class.

Don't talk to strangers!

Principle: The class A should only call functions in class B if you can see the class name B when looking at the header file of A.

Reason: Program maintenance is difficult when there are hidden dependencies.

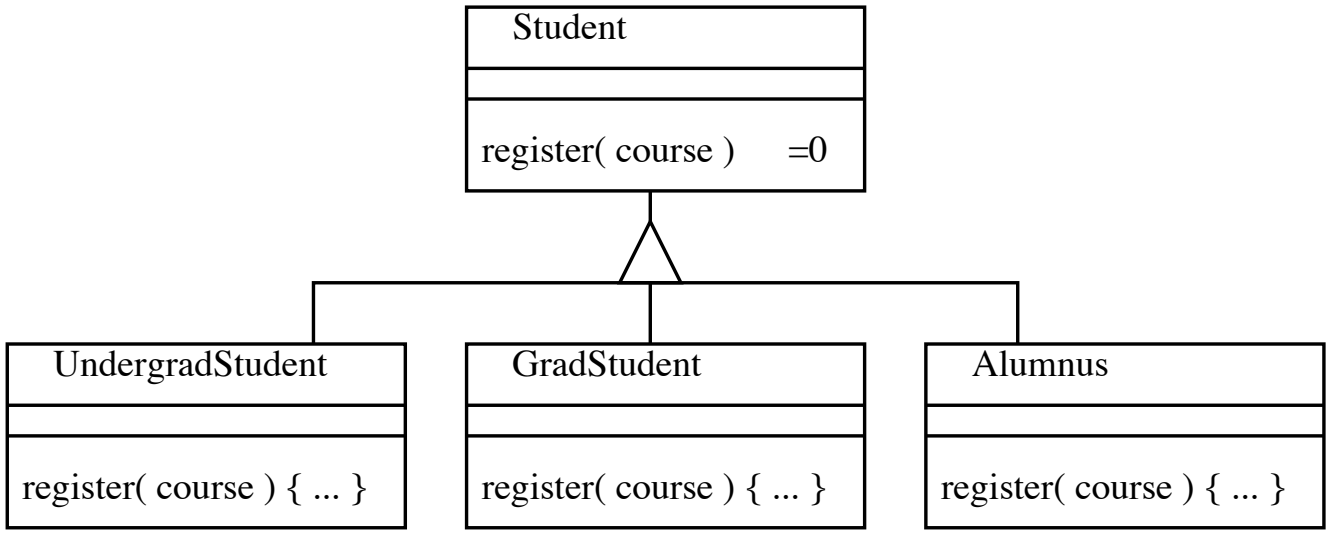
In the diagram, RetailStore should not be calling the `Item::sell` because there is no mention of the Item class in the RetailStore class definition. Instead, RetailStore should call a function in Inventory and let Inventory figure out how to handle it's Items.



Polymorphism

Principle: Use polymorphism (derivation + virtual functions) to implement a set of related but not identical classes.

Reason: This lets the programmer create a common stable interface for dealing with all variations, and also avoids duplicating blocks of code. Coding, debugging, and program maintenance all become easier.



Chain of Responsibility

Objects created by declarations are stored on the run-time stack.

- These objects are deleted automatically when control leaves the declaring block.
- When an object is put into an array or an STL container (such as vector), it is copied. The array or vector becomes the "owner" and it will manage the storage.

Objects that are created by calling new are the programmer's responsibility to manage.

- There must be a clearly defined "owner" of every dynamically created object.
- When an object pointer is put into a container (an array or vector), the "owner" is the class that declares the container.
- The owner is responsible for deleting the object at the end of its useful lifetime.

Common Design Patterns

Adapter
Indirection
Bridge, or Plug-and-Play
Singleton
Observer or MVC
Decorator
Framework
Factory method
Abstract Factory

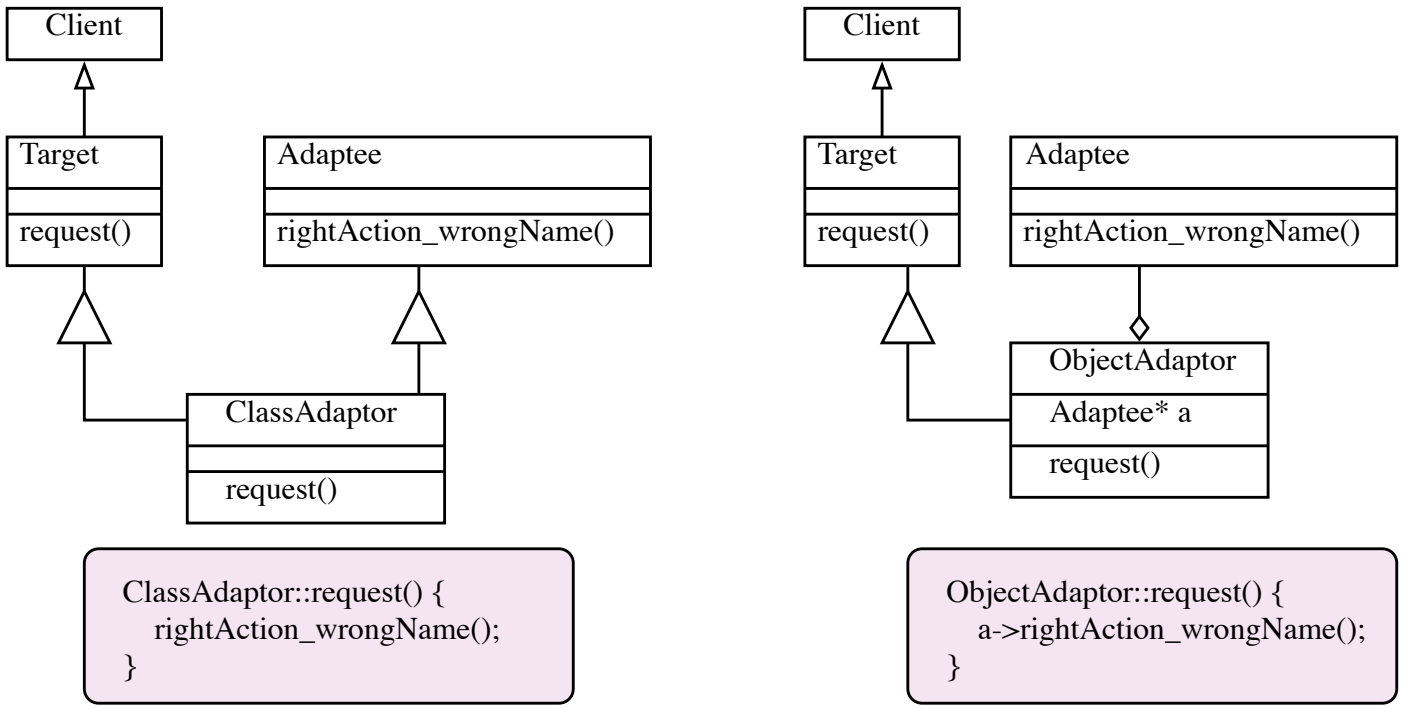
What is a Design Pattern?

An elegant, adaptable and reusable solution to a common software development problem, including:

- A description of the type of problem.
- A design for a set of classes and class relationships that solve the problem,
- Including the necessary data and function members of the classes.
- Guidance for implementing the solution,
- Reasons why the given solution is wise.

Adapter

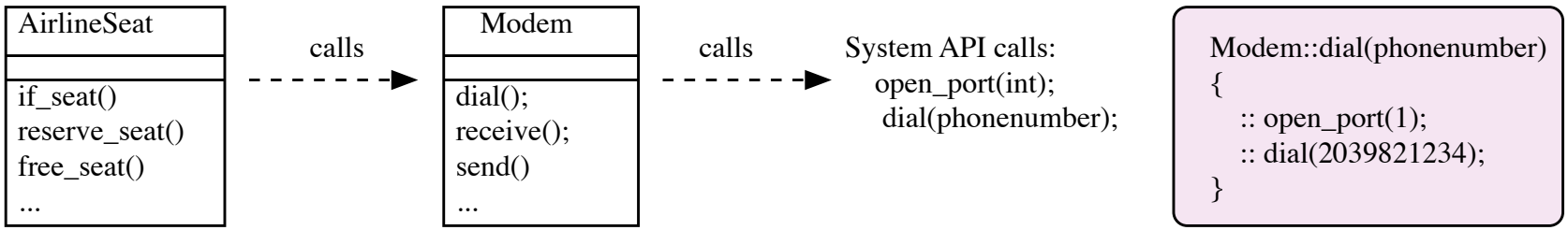
Use an Adapter when you have an existing class that does the right job but has the wrong interface. Use a ClassAdapter if multiple derivation is possible, and if new functionality needs to be added to the reused class. Otherwise, use an Object Adapter.



Indirection

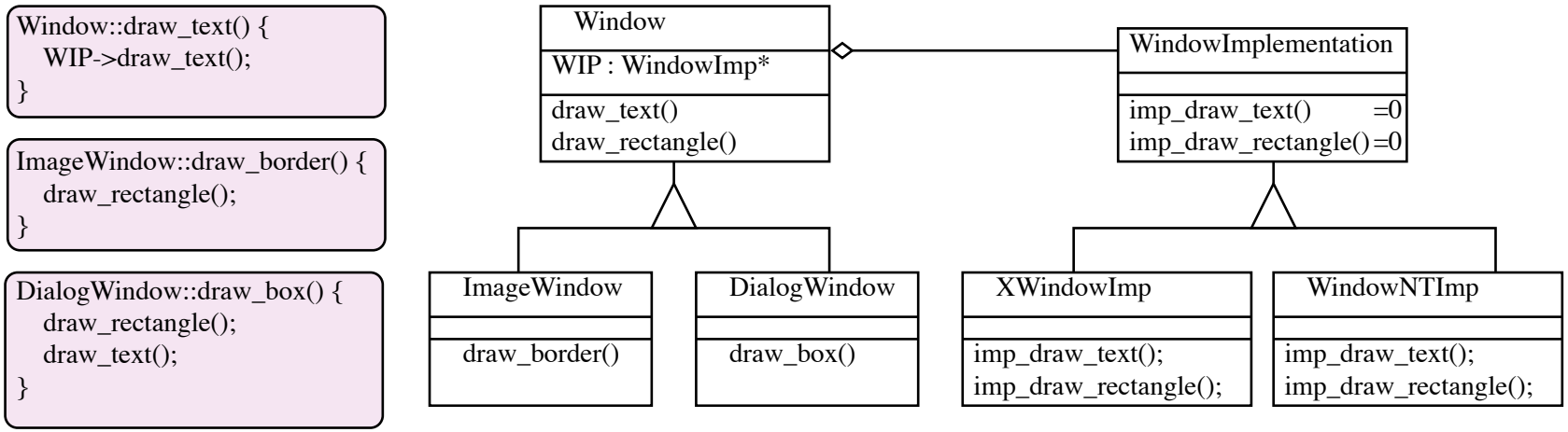
When your application must deal with an internet service (database, credit company), define a local class to make the connection. The rest of your program should use the external service indirectly, through the local class.

Reason: Keep uncontrollable external things isolated, and provide one class that might have to be changed if the external resource changes.



Bridge (Plug and Play)

To build portable systems, design essential modules and implement them as abstract classes. Then derive specific implementation modules from the abstract classes. Reason: Any implementation can be plugged into any abstract slot and be successfully combined with other parts of the system.

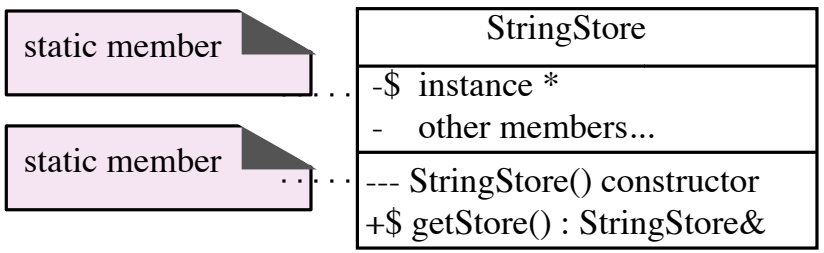


Singleton

In the old days, programmers used global variables because they were available everywhere in the program. However, they were incredibly vulnerable to all kinds of errors.

Now we use Singleton – a class with full protection and exactly one instance that can be used anywhere in the program.

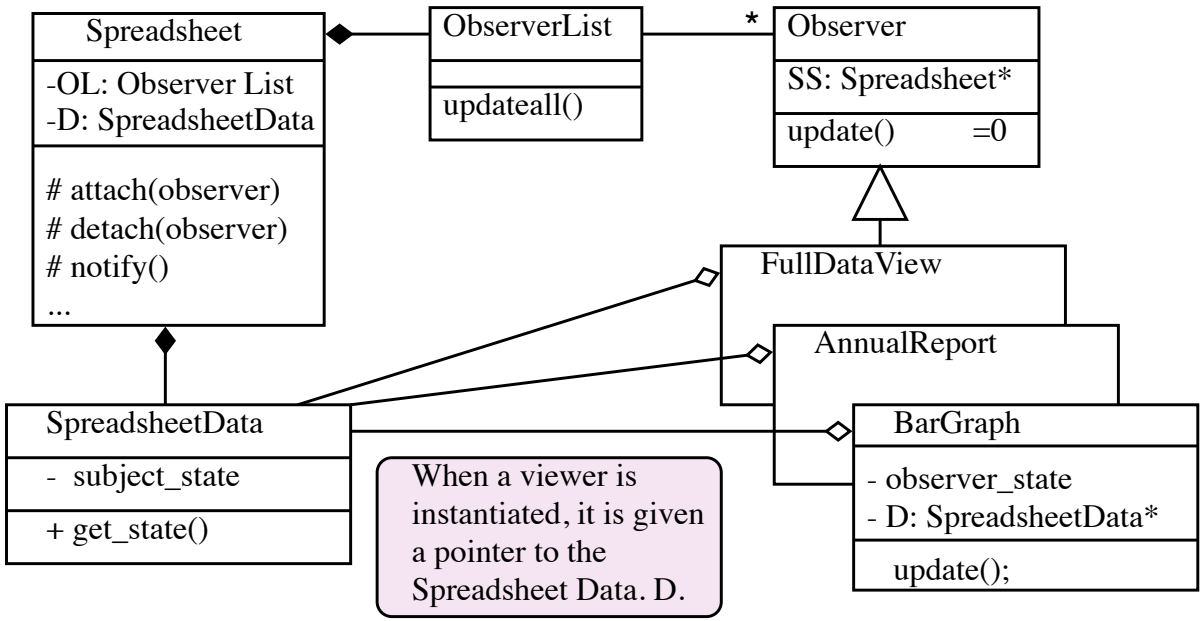
A Singleton instantiates itself the first time its static function is called.




```
StringStore& StringStore::getStore(){
   if (instance==NULL) instance = new StringStore;
   return instance;
}
```

Observer or Publish-Subscribe or Model-View-Controller

Many applications have one or more data classes that hold the state of the computation. Taken together, these form the model, which is owned by a central controller class. In addition, we might need one or many ways to view the model or parts of the model.




```
Spreadsheet::notify() {
    OL.updateall()
}
```

```
ObserverList::updateall() {
    for all  x in the list,
        x->update()
}
```

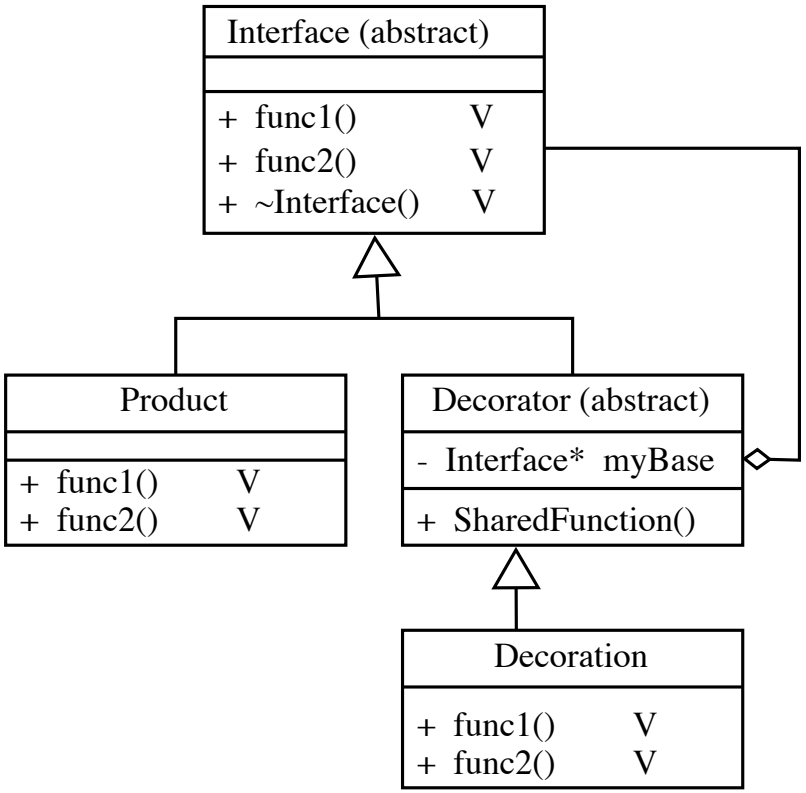
```
Observer::update() {
    observer_state =
        D->get_state();
}
```

```
Spreadsheet::attach( ob ) {
    OL.add (ob );
}
```

MVC Usage

- We use the MVC design pattern to allow an indefinite number of viewers to be created without requiring the Controller to know or guess ahead of time how many viewers there will be.
- Each viewer is registered with the controller (added to its list) when it is created.
- Later, whenever the model changes, each viewer on the list is notified. At that time (or later) the viewer will "pull" the revised data over the connecting link and update the view.

Decorator (See Coffee demo)



In Interface,
define virtual functions for handling products.
A virtual destructor is needed in many circumstances.
Non-virtual interface functions can also be here.

```
Decorator::Decorator( Interface* ac) {
   myBase = ac;
}
```

In Decorator,
delegate all Interface actions to myBaseObject.
Define functions that are common to all decorations.

In Decoration,
Define the abstract functions.

Decorator Usage

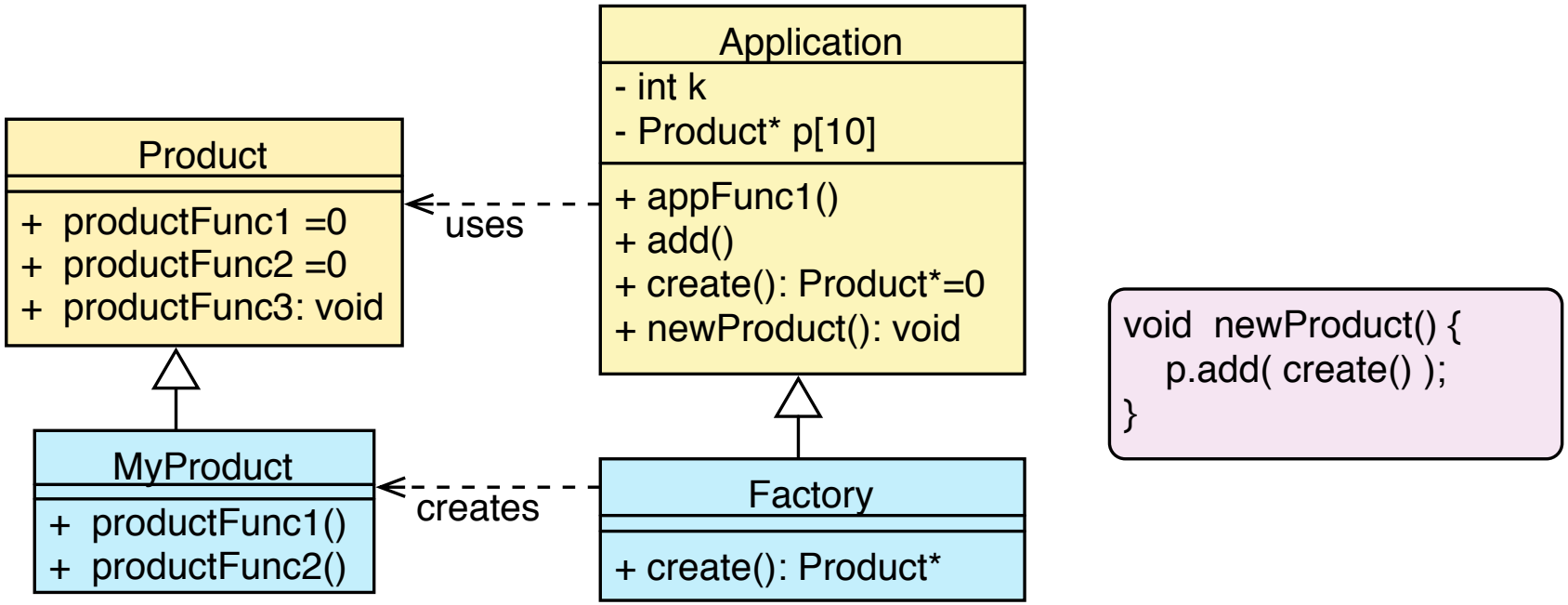
This pattern is used to extend the functionality of an object when there are several independent ways of doing so.

- The common functions, shared by the product and the decorations, are declared by an abstract base class.
- They are implemented by the product and each decoration.
- The Decorator class provides a way to link all the decorations to each other and to the product.
- Application: Think of an online purchase where the price might be affected by any combination of the base price, tax, shipping, employee discounts, coupons and special sales. Business logic demands that we support any combination of these factors, in any order.

Decorating can give objects individualized properties at run-time.

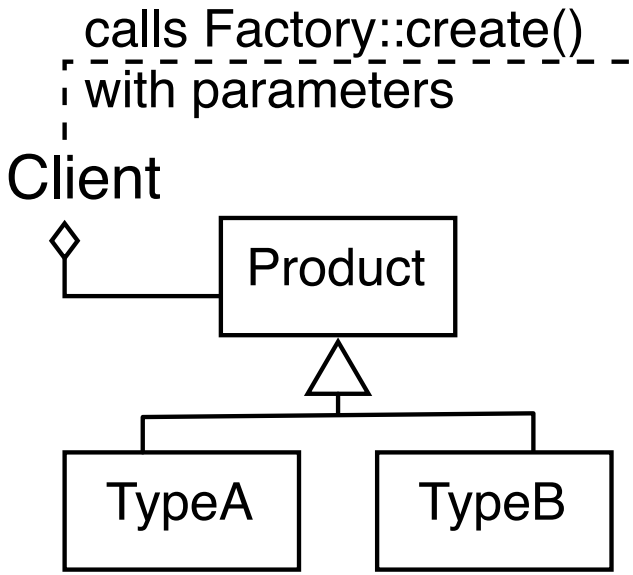
Factory Method used for a Framework

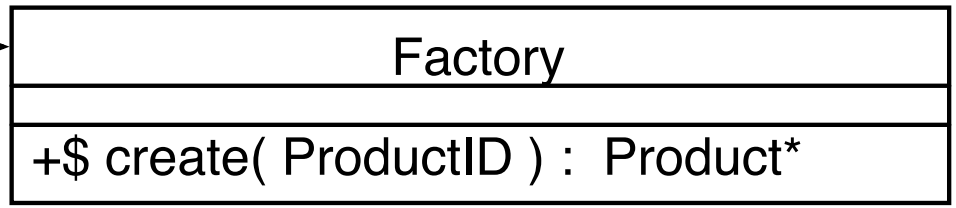
Use when a Framework knows when a new product is needed, but not how to create that product.



The yellow classes are defined by the framework.
The blue classes are derived to implement customized behaviors.

Factory Method with Parameters






```
Product* static create( ProductID ) {
    if(ProductID==A) return new TypeA();
    if(ProductID==B) return new TypeB()
}
```

Factory Method Usage

Using a Factory method decouples the client from the details of object creation.

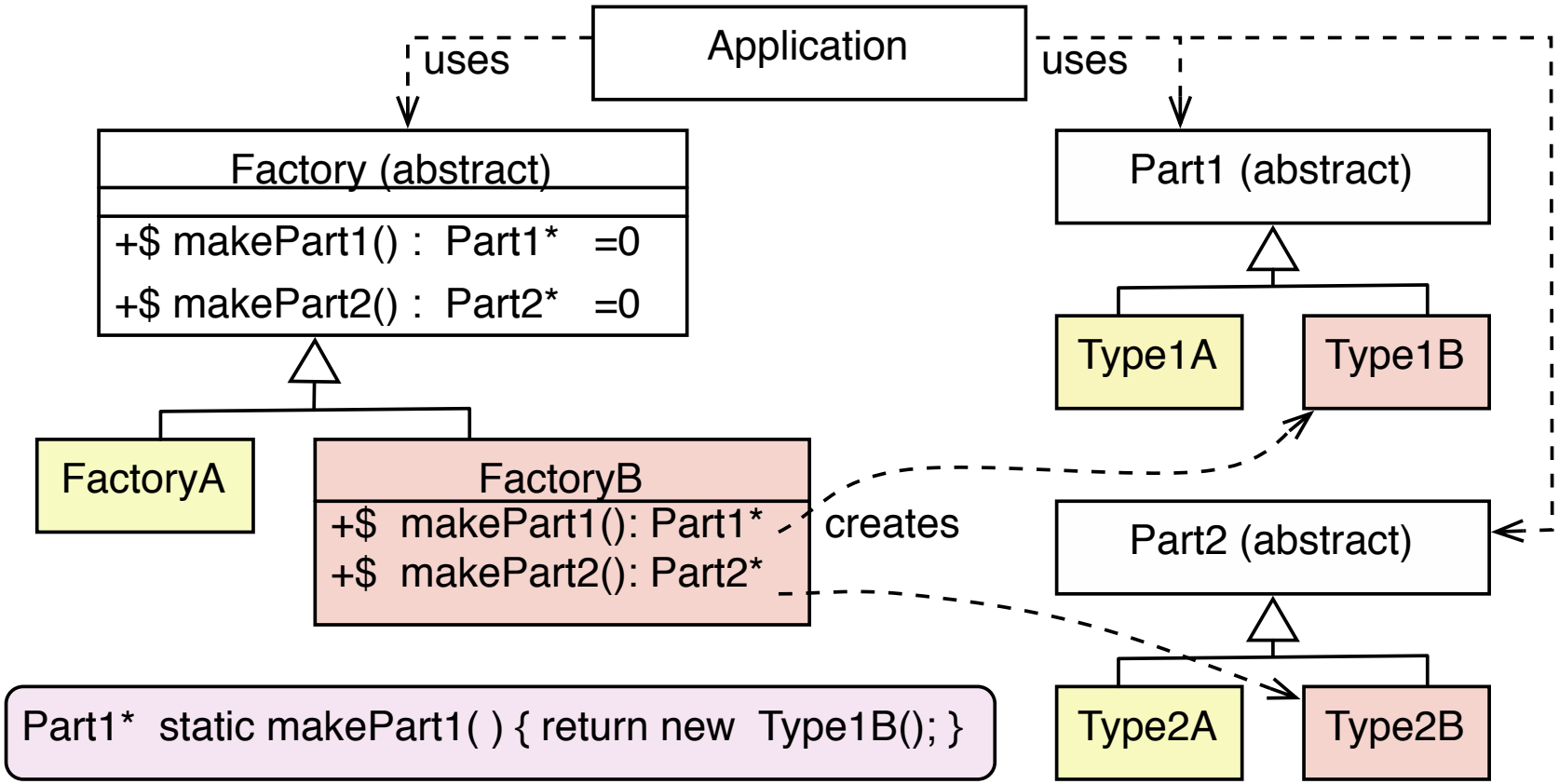
- The Factory class has one static virtual method, which may or may not have a default definition.
- Derived from it are one or more classes that create objects of different styles, but the same general functionality.
- A client calls the CreateProduct() function in the base factory class when it wants an instance of the product.
- This allows a client to select, at run time, the style of product that is wanted. The client does not need to know at compile time what styles are available or any details about how the products are constructed and initialized.

Factory Method Application

A program that builds an HTML photo album.

- The Album is one scrollable page that is a sequence of page-segments.
- Page segments come in several formats: one photo, two side-by-side, three: vertical beside top and bottom, etc.
- Your program (the client) calls the `create(int)` function in the factory class when it wants a new page segment. The parameter is an enum constant for the format.
- The Factory selects, instantiates, and returns a segment of type A or B or C, as requested.
- This allows a client to select, at run time, the style of product that is wanted. There is no need to know, at compile time, what styles will be wanted or any details about how the products are constructed and initialized.

Abstract Factory



Abstract Factory Usage

Using an Abstract Factory method decouples the client from the details of object creation, and permits construction of a set of parts that work together and implement one style.

- Each part is defined by an abstract class with one derived class for each style of part.
- The abstract factory declares a pure virtual function to make each kind of part.
- From the abstract factory, we derive a concrete factory for each style of parts. These classes define the abstract functions so that they instantiate parts of the correct style.

This pattern lets us define multiple parts of multiple styles without repeating large blocks of code.