CSE3150 Final Review

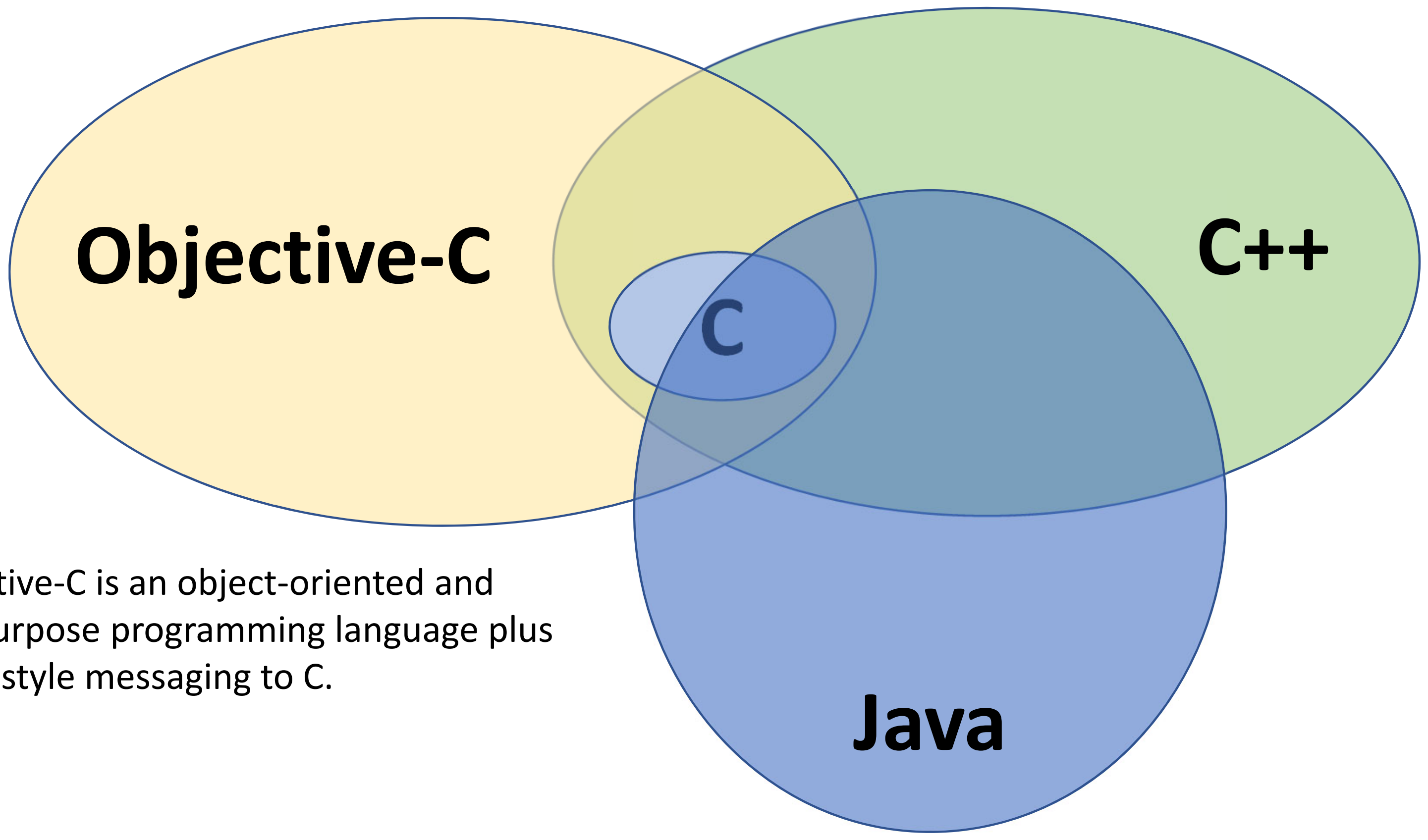
Date and Location: Dec. 6th in class

Scope: All materials taught this semester with an emphasis on the contents covered after the mid-term exam. Questions related to PAs and ICEs may be included.

Format: paper-based

- Same format as the mid-term
- Close book and notes
- 1 A4 double-sided reference sheet is allowed

Family Tree

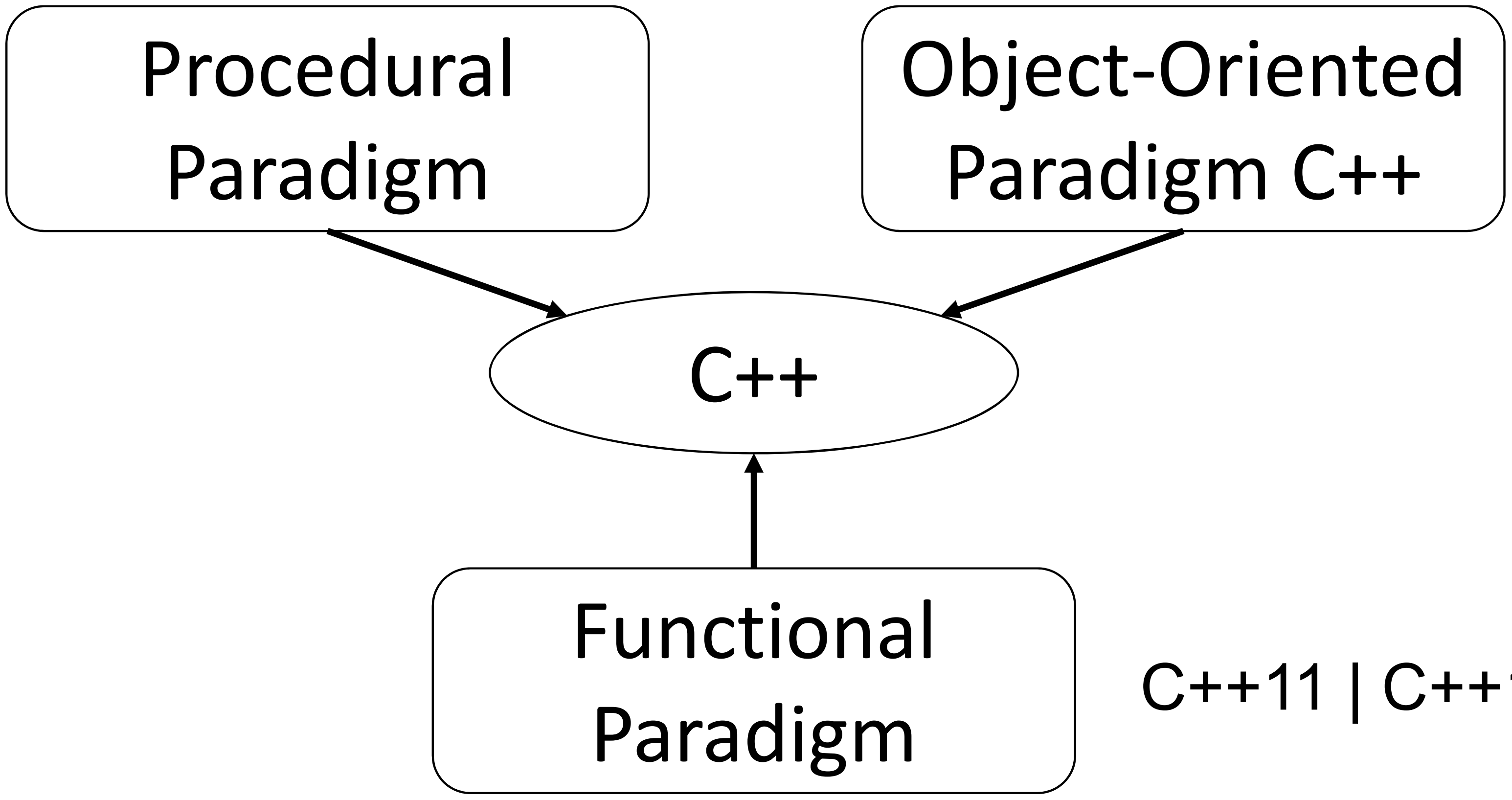


The objective-C is an object-oriented and general-purpose programming language plus Small talk style messaging to C.

Top-Down vs. Bottom-Up Approach

No.	Top-Down Approach	Bottom-Up Approach
1.	In this approach, the problem is broken down into smaller parts.	In this approach, the smaller problems are solved.
2.	It is generally used by structured programming languages such as C, COBOL, FORTRAN, etc.	It is generally used with object oriented programming paradigm such as C++, Java, Python, etc.
3.	It is generally used with documentation of module and debugging code.	It is generally used in testing modules.
4.	It does not require communication between modules.	It requires relatively more communication between modules.
5.	It contains redundant information.	It does not contain redundant information.
6.	Decomposition approach is used here.	Composition approach is used here.
7.	The implementation depends on the programming language and platform.	Data encapsulation and data hiding is implemented in this approach.

Paradigm Blend



C++11 | C++14

C++ String

- Built-in class for C++
- A replacement for char* in C
- Can hold constant strings, variable strings
- Support a range of operators

http://en.cppreference.com/w/cpp/string/basic_string

C++ Constness

- Makes an entity immutable so no risk of accidental change
- Annotation added on declarations of variables, of functions’ arguments

```
#include <iostream>
using namespace std;


int main()
{
    cout << "Please enter your first name:";
    string name;
    cin >> name;
    const string greet = "Hello " + name + " !";
    const int width = greet.size();
    width ++;
    const string star(width+4,'*’);
    const string row2 = "* " + string(width,' ') + " *";
    cout << star << endl << row2 << endl << "* " + greet
+ " *" << endl << row2 << endl << star << endl;
    return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;

void printVector(const vector<int>& vec)
{
  for(int i=0; i < vec.size(); i++)
  {
    vec[i] ++;
    cout << vec[i] << endl;
  }
}

int main()
{
  vector<int> vec(3,10);
  printVector(vec);
  return 0;
}
```

C++ Numeric Types

- Same as in C99
  - char [1] , short [2] , int [4] , long [4,8], long long [8]
  - float [4], double [8]
  - [signed] and [unsigned] variants.

- One addition
  - There is a new type: bool (same as char)

C++ Pointers

- Get the “address” of something
- Dereference an “address” to get to something
- Compute the address of something

Memory Models

- Three pools of memory
    - Static
    - Stack
    - Heap

- Each pool features
    - Different lifetime
    - Different allocation/deallocation policy

Memory Models

Static

- Memory is allocated at compile time
- Memory is read/write
- One copy alone. Not dynamic
- Use the "static" keyword
- Lifetime spans the entire execution of the program

```
void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    count++;
}
```

```
int main()
{
    for (int i = 0; i < 5; i++)
        demo();
    return 0;
}
```

Stack

- Memory allocated for local variables of functions
- Allocated automatically when entering the function
- De-allocated automatically when you leave the function.
- Each recursive invocation gets its own set of local variables!

Heap

- Memory comes from for manual "on-the-fly" allocations
- Two simple APIs: **new** and **delete**
- The programmer is in charge for both allocation / deallocation
- Lifetime of memory blocks is as long as they are not freed!

Pointer & Constness

Example 1

```
#include <iostream>

int main()
{
   using namespace std;
   int x = 10;
   const int* px = &x;
   cout << "px = " << px << endl;
   cout << "*px = " << *px << endl;
   *px = 20;
   return 0;
}
```

Example 2

```
#include <iostream>

int main()
{
   using namespace std;
   int x = 10;
   int* const px = &x;
   cout << "px = " << px << endl;
   cout << "*px = " << *px << endl;
   *px = 20;
   px = px + 1;
   return 0;
}
```

Reading Types using the Spiral Rule

There are three simple steps to follow: (http://c-faq.com/decl/spiral.anderson.html)

1. Starting with the unknown element, move in a spiral/clockwise direction; when encountering the following elements replace them with the corresponding English statements:

[X] or []

    => Array X size of... or Array undefined size of...

(type1, type2)

    => function passing type1 and type2 returning...

*****

    => pointer(s) to...

2. Keep doing this in a spiral/clockwise direction until all tokens have been covered.

3. Always resolve anything in parenthesis first!

References

- Only exist in C++
- Serves the same purpose as pointers
- With a few advantages
    - Cleaner syntax
    - Stricter semantics
- Can be mixed with pointers and constness

References with const

- Same semantics as for pointers
- Use the spiral rule to read the type too!
- Enforces ***read-only*** semantics

```
int sumOf(const string& a) {
    int ttl = 0;
    for(int i=0;i<a.size();i++)
        ttl += a[i];
    return ttl;
}
```

Control Flows

- Reviewing C Control Flow
    - if | while | for | switch | …
- C++ Specifics
    - try-catch-throw
    - range loop
    - Intervals

Handling Exceptions in C++

- The exception must be a value
  - int | float | char | …
  - String
  - Any object!
  - Pointer to something…

- C++ offers two mechanisms
  - One to report exceptions
  - One to handle exceptions

Report exceptions in C++

- **throw** <expr>
  - A statement
  - With an expression argument
  - Value of expression is what is “thrown”

- Semantics
  - Normal control-flow path is altered
  - Flow is transferred to a catch site
  - If there is no catch-site in this function, return to the callee and handle it there! (i.e., look for a catch site …)
  - If we reach main and still no catch site the OS kills the program

Handling exceptions in C++

```
try {
   <block0>
} catch(<decl1>) {
   <block1>
} …
} catch(<declk>) {
   <blockk>
}
```

Semantics:

Try to run block0. If an exception is raised within block0 (or any callee), then the catch clauses (1..k) below are potential catch sites. Check them in sequence. Execute the block of the first applicable catch.

Range Loop

```
#include <iostream>
#include <string>

int main()
{
   using namespace std;
   string vals = {‘H’,’e’,’l’,’l’,’o’};
   for(char v : vals) {
      cout << "v = " << v << endl;
   }
   return 0;
}
```

for (<decl> **:** <expr>)
   <singleStmt>

Syntax:

- Declare a local variable
- Draw it from a sequence
- Sequence comes from expression

Semantics:

Evaluate <expr> to get a sequence. Use an open range to iterate over the sequence content. At each iteration bind the typed named in <decl> and execute the <singleStmt>

Standard Template Library (STL)

Three fundamental pillars

- Containers
- Iterators
- Algorithms

STL Containers

Containers or container classes store objects and data.

Sequence Containers: implement data structures that can be accessed in a sequential manner.
Vector, list, deque, arrays, forward_list

Container Adaptors: provide a different interface for sequential containers.
Queue, priority_queue, stack

Associative Containers: implement sorted data structures that can be quickly searched (O(log n) complexity).
Set, multiset, map, multimap

Unordered Associative Containers: implement unordered data structures that can be quickly searched
unordered_set, unordered_multiset, unordered_map, unordered_multimap

Stack

A type of container adaptors with LIFO (Last In First Out) type of working.

Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
  stack<int> si;
  for(int i=0;i<5;i++) si.push(i);
  while (!si.empty()) {
    cout << "value:" << si.top() << endl;
    si.pop();
  }
  return 0;
}
```

Vector

Vector provides

- A sequence
- Dynamically sized
- Ability to add / remove anywhere
- Random access
- Bounds safety

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> vec;
  for(int i=0;i<10;i++) vec.push_back(i);
  for(int v : vec) cout << v << " ";
  cout << endl;
  return 0;
}
```

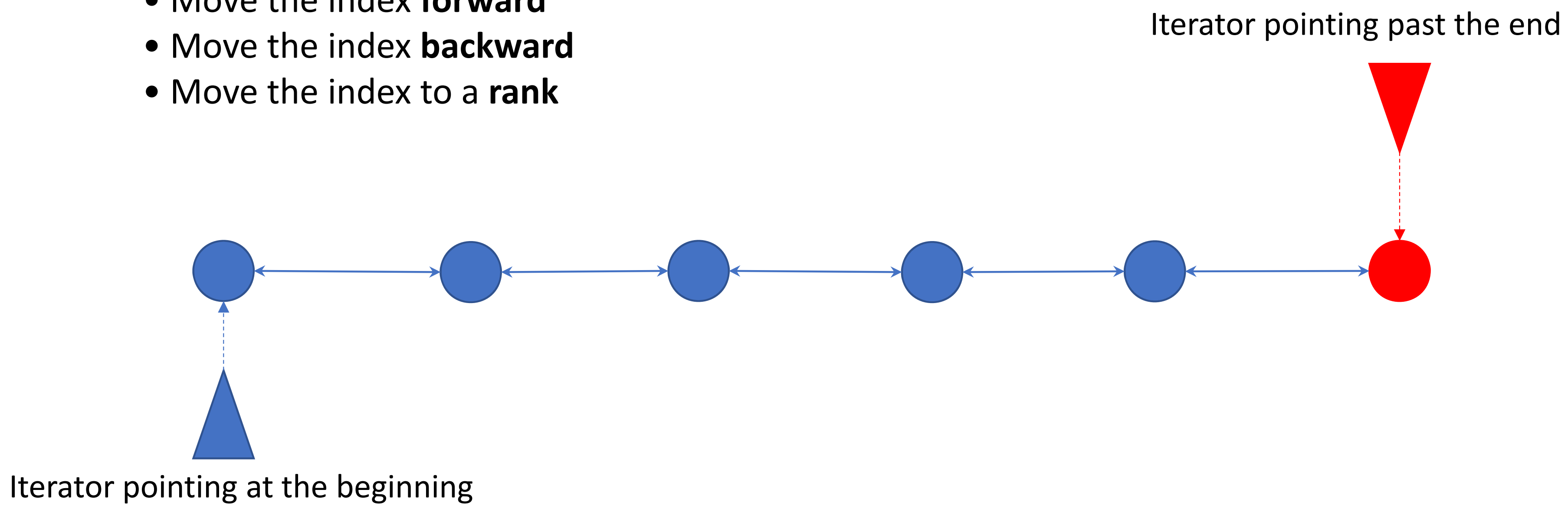
```
int sum(vector<int>& vec) {
  int ttl = 0;
  for(int v : vec) ttl += v;
  return ttl;
}

double sum(vector<double>& vec) {
  double ttl = 0;
  for(double v : vec) ttl += v;
  return ttl;
}
```

```
template <class T>
  T sum(vector<T>& vec) {
    T ttl;
    for(T v : vec) ttl += v;
    return ttl;
}
```

Iterators

Iterators are abstractions for “indexes” into “containers”

You can
- Make the iterator “point” to the **beginning**
- Make the iterator “point” ***past* the end**
- Move the index **forward**
- Move the index **backward**
- Move the index to a **rank**


Iterator pointing past the end
Iterator pointing at the beginning

STL and Iterators

The **container** has methods to manufacture iterators
- Pointing at the right place: front | end
- Going in a direction: forward | backward

An **iterator** is an object representing an index
- Can move forward
- Can lookup the value it is “indexing to”
- Can be compared with iterators (e.g., to test end!)

AUTO

- The types in the previous example is big, unreadable and hard to remember
- Using type inference to let the compiler *figure out* what the type is!

```
int main() { …

  cout << endl << "Reverse!" << endl;
  for(vector<int>::reverse_iterator k = vec.rbegin(); k != vec.rend(); k++)
    cout << *k << " ";
  cout << endl;
  return 0;
}
```

```
int main() { …

  cout << endl << "Reverse!" << endl;
  for(auto k = vec.rbegin();k != vec.rend();k++)
    cout << *k << " ";

  cout << endl;
  return 0;
}
```

Struct vs. Class

The two constructs are identical in C++ except that in structs the default accessibility is **public**, whereas in classes the default is **private**.

```
struct Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;
};
```

```
class Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;
};
```

Function Overloading vs. Overriding

Function Overloading	**Function Overriding**
Function Overloading provides multiple definitions of the function by changing signature.	Function Overriding is the redefinition of base class function in its derived class with same signature.
An example of compile time polymorphism.	An example of run time polymorphism.
Function signatures should be different.	Function signatures should be the same.
Overloaded functions are in same scope.	Overridden functions are in different scopes.
Overloading is used when the same function has to behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some different job than the base class function.
A function has the ability to load multiple times.	A function can be overridden only a single time.
In function overloading, we don’t need inheritance.	In function overriding, we need an inheritance concept.

Lifetime of an Object - Birth

Birth with constructors

- Execute a method when creating an object
- **Default constructor:** Called when no arguments provided
- **Custom constructor:** Called with arguments to setup the instance
- **Copy constructor:** Initializes an object using another object of the same class. The source object that is passed to the constructor will have its member fields *copied* into the new object.
- **Move constructor:** The source object that is passed to the constructor will have its member fields *moved* into the new object.

```
struct Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;

   Student() { _mt = _final = 0;}
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

_name and _homeworks are classes and will call their default constructors for initialization.

```
struct Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;
   Student() { _mt = _final = 0;}
   Student(const std::string& s) {
    _name = s;_mt=_final=0;}
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Copy vs. Move Constructors

```
struct Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;
   Student() { _mt = _final = 0;}
   Student(const std::string& s)
   { _name = s;_mt=_final=0;}

   Student(const Student& s2) {
      _name = s2._name;_mt=s2._mt;_final=s2._final;
      for(double d : s2._homeworks)
         _homeworks.push_back(d);
   }
   void read(std::istream& is);
   void print(std::ostream& os);
};
```

```
struct Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;
   Student() { _mt = _final = 0;}
   Student(const std::string& s)
   { _name = s;_mt=_final=0;}

   Student(const Student& s2)
   : _name(s2._name), _mt(s2._mt), _final(s2._final),
     _homeworks(s2._homeworks) {}

   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Calls the copy constructors rec.

```
struct Student {
   std::string _name;
   double _mt,_final;
   std::vector<double> _homeworks;
   Student() { _mt = _final = 0;}
   Student(const std::string& s)
   { _name = s;_mt=_final=0;}

   Student(const Student& s2)
   : _name(s2._name), _mt(s2._mt),_final(s2._final),
   _homeworks(s2._homeworks) {}

   Student(Student&& s2)
   : _name(std::move(s2._name)),
     _mt(s2._mt),_final(s2._final),
     _homeworks(std::move(s2._homeworks)) {}

   void read(std::istream& is);
   void print(std::ostream& os);
};
```

Transfer Operators

Simple idea: Execute a method invoking assignment (e.g., x = y;)

Copy Operators

Move Operators

```
Student& operator=(const Student& s) {
   if (this == &s) return *this;
   _name = s._name;
   _mt = s._mt;
   _final = s._final;
   _homeworks = s._homeworks;
   return *this;
}

Student& operator=(Student&& s) {
   _name = std::move(s._name);
   _mt = s._mt;
   _final= s._final;
   _homeworks = std::move(s._homeworks);
   return *this;
}
```

Lifetime of an Object - Death

Death with destructors

- Release whatever resource is held by the instance (memory, files, network connections, etc.)
- There is only one destructor.
- You can use the “virtual” keyword to handle polymorphic destruction.

Protection

Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program.

Private: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

Protected: The protected access modifier is similar to the private access modifier in the sense that it can’t be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Struct Protection vs. Class Protection

C-style **struct assume** that everything is **public**

- you can change that by using public: or private:
- It is backward compatible with C (where everything is public)

By Default everything is **private** in class

- Attributes
- Methods

You can change (several times) the default in class

- Declaration that follows a privacy change use the new privacy

Abstract class

A class with
- (state +) behavior
- no implementation! [or a partial implementation]

Corollary
- One **never instantiates** an abstract class
- One **only sub-classes** an abstract class

How to use abstract classes
- Use ***inheritance*** to claim that you support the contract
- Provide an ***implementation*** in the sub-class for “**pure**” methods

```
class AStudent {
public:
   virtual void read(std::istream& is) = 0;
   virtual void print(std::ostream& os) = 0;
   virtual void setName(const std::string& n) = 0;
};
```

Inheritance

From a **pragmatic** standpoint
- Promote code reuse

From a **theory** standpoint
- Support type refinements
- aka sub-typing through sub-classing

Let **A**,**B** be two types
- We write **A <: B** to state that A is a subtype of B

Subsumption: Whenever A <: B
- Anytime a B is expected, one can provide an A

Inheritance & Privacy

When inheriting you can “alter” privacy of what you inherit

Public Inheritance Semantics:

What you inherit keeps its status
- public ➔ public
- protected ➔ protected
- private ➔ private

When class A publicly inherits from B
- A
- Its sub-classes and
- Every function knows that A inherits from B

Inheritance & Privacy (Cont.)

Protected Inheritance Semantics:

What you inherit changes status
• public ➔ protected
• protected ➔ private

When class A inherits in a protected way from B
• A and its sub-classes know that they inherit from B
• Nobody else knows that fact.

Inheritance & Privacy (Cont.)

Private Inheritance Semantics:

What you inherit changes status

- public ➔ private
- protected ➔ private

When class A inherits in a private way from B

- A knows that it inherits from B
- A’s sub-classes are clueless
- Everything else is clueless
- That inheritance fact is completely hidden.

Inheritance & Overriding

When class A inherits from class B
- It has all the attributes of B
- It has all the methods of B
- It can add attributes or methods

But it can also
- **Upgrade [refine]** some of the methods it inherits
- That’s called **overriding**

Need to achieve **RUNTIME** polymorphism through **Dynamic Binding** of overriden methods

```
#include <iostream>

class Mammal {
public:
   Mammal() {}
   void print() { std::cout << "I (" << this << ")'m a mammal!" << std::endl;}
};

class Human :public Mammal {
public:
   Human() {}
   void print() { std::cout << "I (" << this << ")'m a human!" << std::endl;}
};

int main() {
   Mammal m;
   Human h;
   Mammal& mr = h;
   mr.print();
   return 0;
}
```

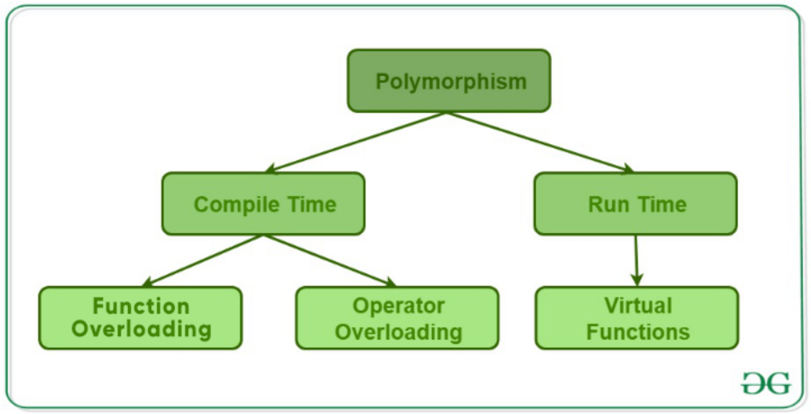
C++ Polymorphism

Compile-Time Polymorphism

- Function overloading
- Operator overloading

Runtime Polymorphism

- Provide the ability for an object to respond to messages based on its *dynamic* type rather than its *compile-time* type.
- Function Overriding and Virtual Function
- The “virtual” keyword switches from static binding to dynamic binding



https://www.geeksforgeeks.org/cpp-polymorphism/

Finality

There is also a way to state that a method can no longer be overridden in sub-classes.

- Add the final qualifier

```
#include <iostream>

class B {
public:
   virtual void f(int) {std::cout << "B::f" << std::endl;}
};
class D : public B {
public:
   virtual void f(int) override final {std::cout << "D::f" << std::endl;}
};
class F : public D {
public:
   virtual void f(int) override {std::cout << "F::f" << std::endl;}
};

int main() {
   B* aPtr = new F;
   aPtr->f(1);
   return 0;
}
```

Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. The constructors of inherited classes are called in the same order in which they are inherited.

The destructors are called in reverse order of constructors.

```
#include<iostream>
usingnamespacestd;

classA{
public:
  A()  { cout << "A's constructor called" << endl; }
};

classB{
public:
  B()  { cout << "B's constructor called" << endl; }
};

classC: publicB, publicA  {
public:
  C()  { cout << "C's constructor called" << endl; }
};

intmain(){
    C c;
    return0;
}
```

Object-Oriented (OO) Design

Compare OO design with structural design
– Encapsulation (info hiding), polymorphism, abstraction

Objects package data and procedures (methods) that operate on the data
– Client -> Send a request (message) ->Server performs an operation
– Request is the **only** way to change object’s internal data: encapsulation

OO Design: decompose into a set of manageable and interacting objects
– encapsulation, granularity, flexibility, performance, evolution, reusability, and so on

Object Interface and Inheritance

Signature of operation/method: name, parameters, and return value

Object interface: the set of all signatures of its operations

Type: name of an interface
– Object *a* has type A ⇔ *a* accepts all requests in A
– One object with multiple types (multiple inheritance)
– Widely different objects with same type
– Subtype: inherit another interface

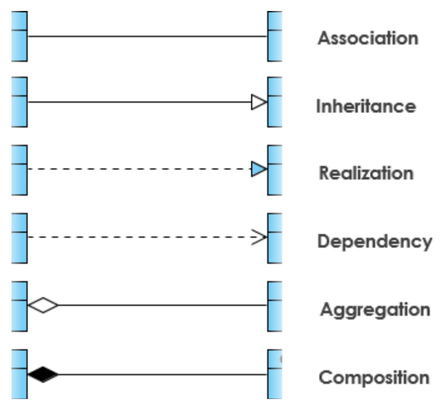
Inheritance: A way of building software incrementally

Polymorphism: a variable referring to type A can refer to an object of type B if B is a subclass of A (through dynamic binding)

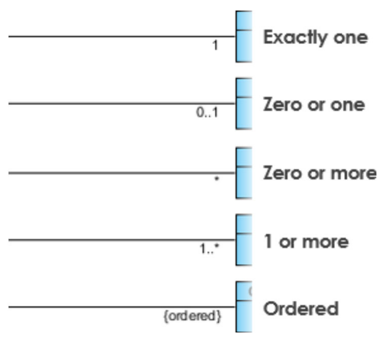
Unified Modeling Language (UML)

A widely adopted standard notation for representing OO designs

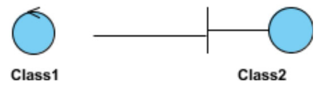
Relationships between classes



Cardinality

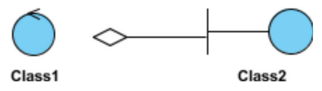


Association



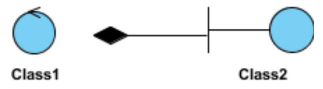
Typically named using a verb or verb phrase which reflects the real world problem domain.

Aggregation



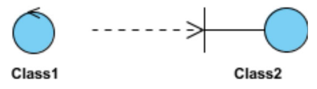
A special type of association, representing a "part of“ relationship. Many instances of Class2 can be associated with Class1. Objects of Class1 and Class2 have separate lifetimes.

Composition



A special type of aggregation where parts are destroyed when the whole is destroyed. Objects of Class2 live and die with Class1. Class2 cannot stand by itself.

Dependency



An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

OO Principles

Difference between class and type:

- An object’s class defines how the object is implemented. The class defines the object’s internal state and the implementation of its operations.
- An object’s type only refers to its interface - the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

Difference between class inheritance and interface inheritance

- Class inheritance defines an object’s implementation in terms of another object’s implementation. In short, it’s a mechanism for code and representation sharing.
- Interface inheritance (or subtyping) describes when an object can be used in place of another.

Principle #1: Program to an interface, not implementation!

- Client unaware of server’s implementation (e.g. which class), only interface matters

Reuse: inheritance vs. composition

Inheritance: white-box reuse
- The term “white-box” refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.

Composition: assemble objects to get complex functionality. Black-box reuse
- Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible.

Principle #2: Favor object composition over class inheritance.

Design for change

– Create an object by specifying a particular class.

– Specify a particular operation

– Dependence on hardware/software platform.

– Dependence on object representation and implementation.

– Algorithmic dependence.

– Tight coupling.

– Subclassing.

– Inability to alter classes conveniently.

OO Design vs. procedural design

• **OO design:**
– Pros: flexible, reusable, easier to maintain through decoupling.
– Cons: more complex, less efficient

• **Procedural design:** better if software needs little change.

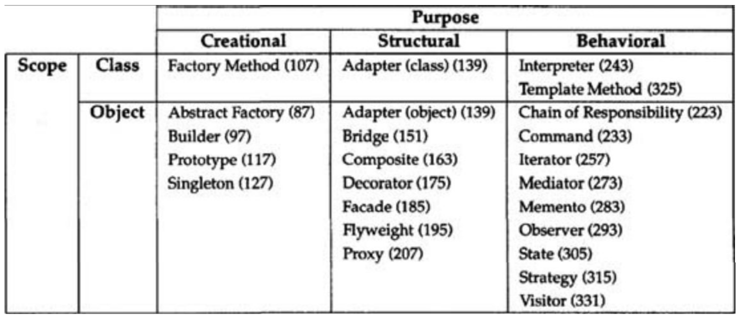
• **How to learn OO design?** Learn through experience, from expert and case studies.

Design Patterns

A design pattern **names**, **abstracts**, and **identifies** the key aspects of a common design structure that make it useful for creating a **reusable object-oriented design**. We classify design patterns by two criteria.

The Purpose criterion reflects what a pattern does:
Creational patterns concern the process of object creation.
Structural patterns deal with the composition of classes or objects.
Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The Scope criterion specifies whether the pattern applies primarily to classes or to objects.
Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time.
Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Design pattern space

Creational Design Patterns

Goal: Create the classes/objects flexibly to support the **design for change principle** and minimize binding.

Two major approaches: Inheritance and Composition

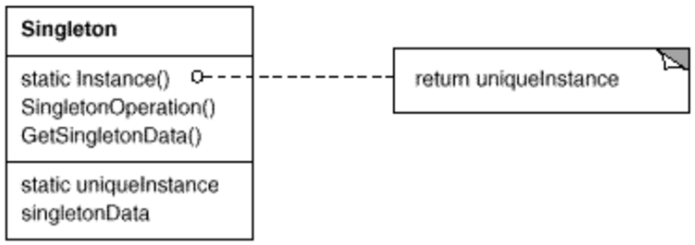
Two categories of flexibility in creating objects: compile-time and run-time.

Singleton

Typically we have **multiple** instantiated objects from **one** class.

But sometimes only want **one and only one**. And this single object is needed by many different places.

Problem: How to ensure one class has a **single** instance, and provide global access to it?



```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

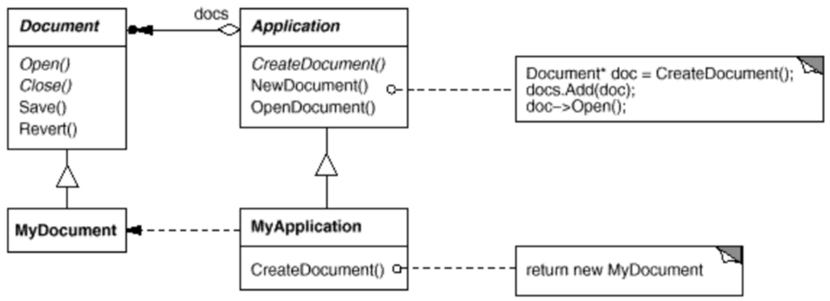
```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Factory Method (Virtual Constructor)

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class.

We call CreateDocument a **factory method** because it's responsible for "manufacturing" an object.



Abstract Factory

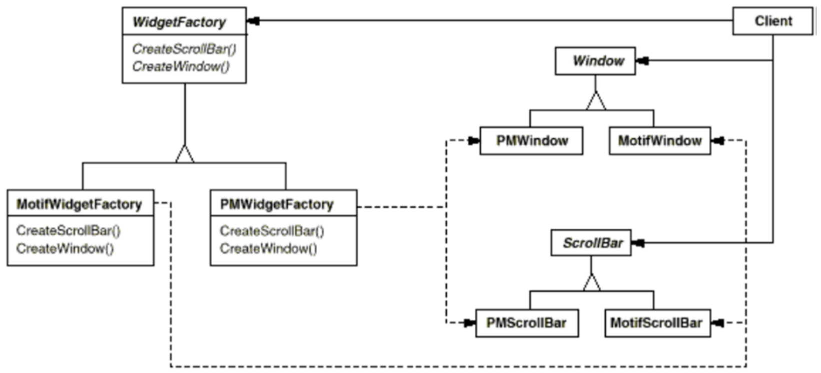
Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel.

Define an abstract WidgetFactory class that declares an interface for creating each basic kind of widget.

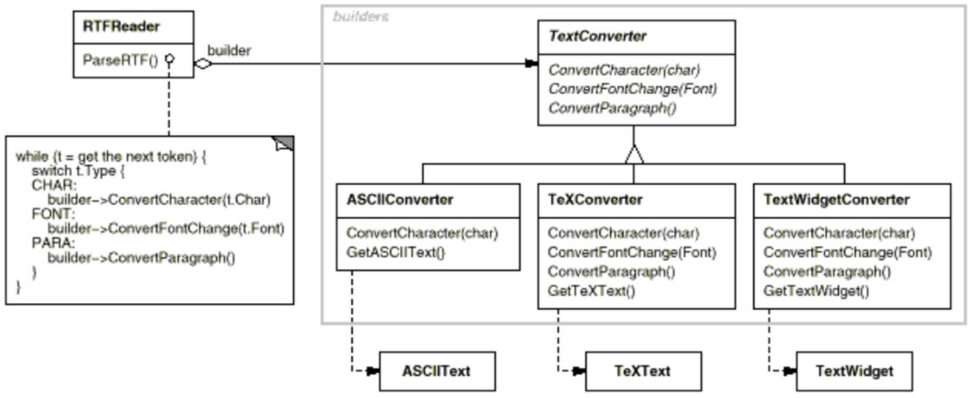
Define abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards.

WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.



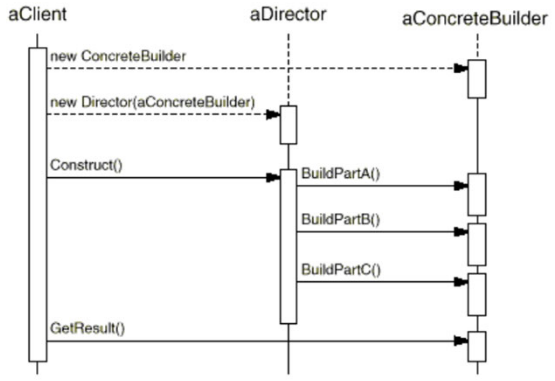
Builder

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.



```
while (t = get the next token) {
   switch t.Type {
   CHAR:
      builder->ConvertCharacter(t.Char)
   FONT:
      builder->ConvertFontChange(t.Font)
   PARA:
      builder->ConvertParagraph()
   }
}
```

- Document exchange problem: need to convert to different formats. Want to support new format easily.
- Document contains different types of units.
  – Character, paragraph, font, image, etc.
- Need to do differently on these units for different formats.



The Builder pattern separates the algorithm for interpreting a textual format (that is, the parser for RTF documents) from how a converted format gets created and represented. This lets us reuse the RTFReader's parsing algorithm to create different text representations from RTF documents—just configure the RTFReader with different subclasses of TextConverter.

Discussion

- Factory method: subclassing
- Abstract factory, builder: composition
  – A factory object
- Compare these two approaches
  – Abstract factory mainly for object creation
  – Builder can bring in my complex operations
- What design principles are behinds these design pattern? Design for Changes

Structural Design Patterns

Goal: Structural patterns are concerned with how classes and objects are composed to form larger structures.

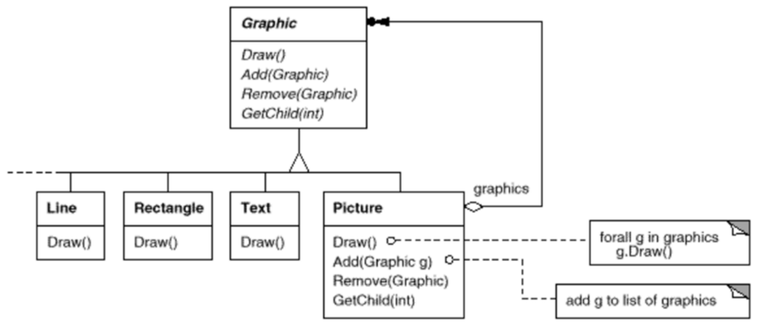
Some Examples:

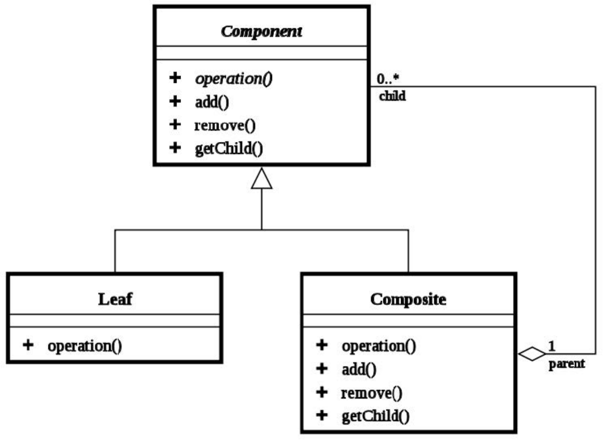
- Composite
- Bridge
- Proxy
- Decorator

Composite Pattern

Composite object: objects contain other objects.

Composite pattern: treat composite object the **same** as atomic object.

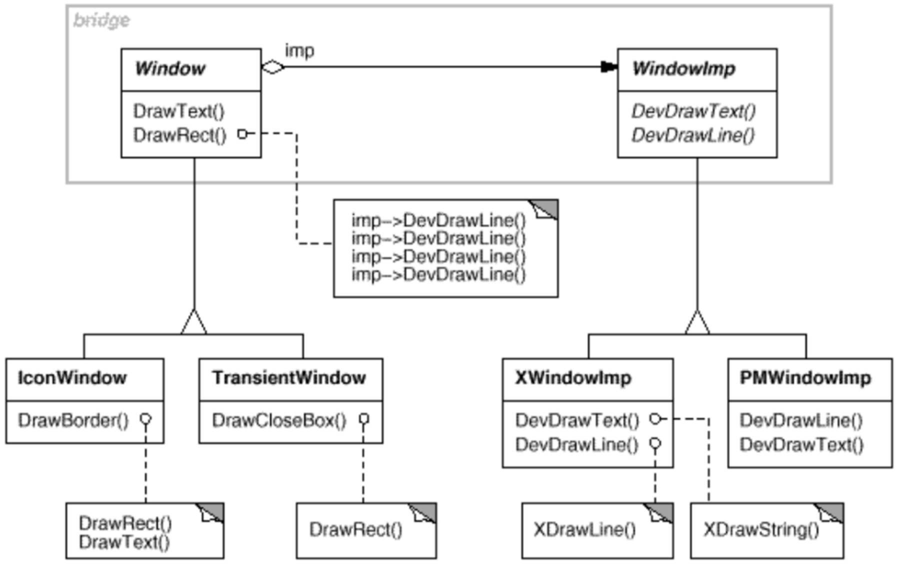




Bridge Pattern

Intent: Decouple an abstraction from its implementation so that the two can vary independently.

Motivation: When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.

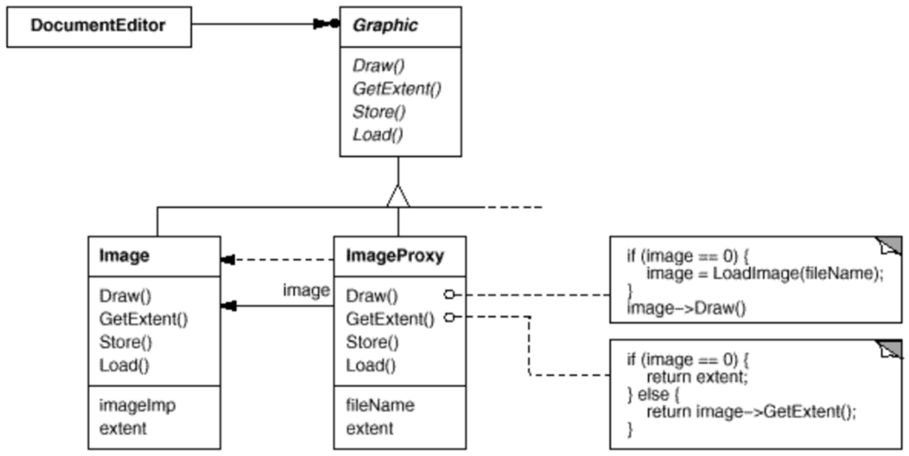


Proxy Pattern

Intent: Provide a surrogate or placeholder for another object to control access to it.

Motivation: One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.

In this specific example, the image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it.

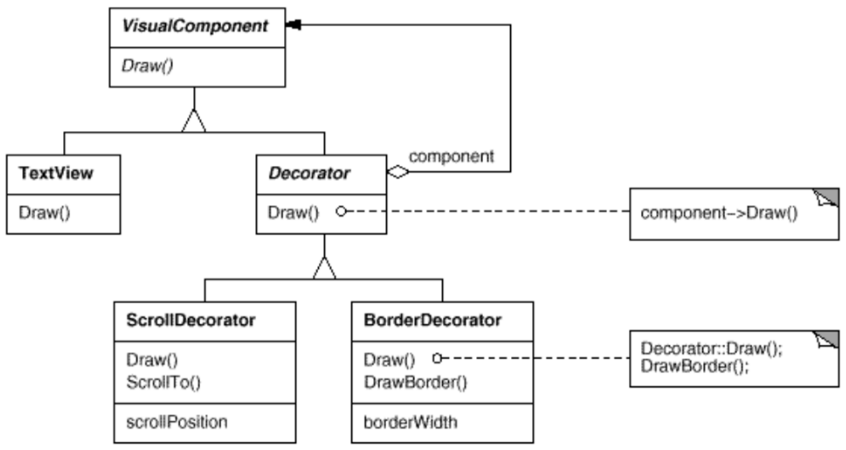


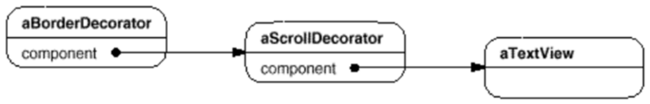
Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Motivation:

- More flexible than inheritance
- Transparent enclosure by forwarding.
- Decorator can perform additional actions before or afterwards.





Behavioral Design Patterns

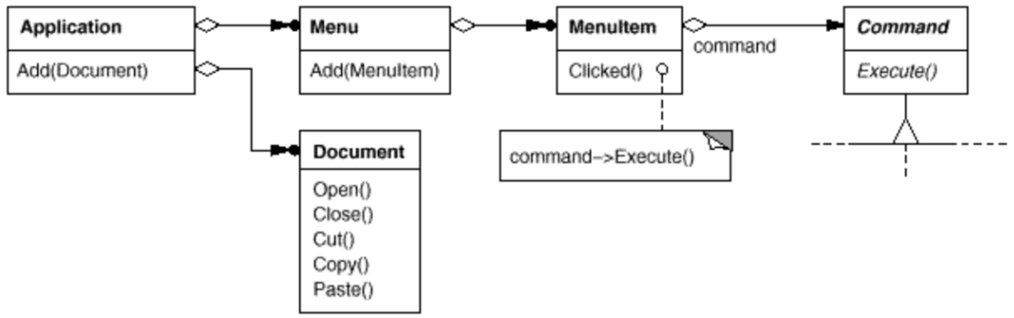
Goal: Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

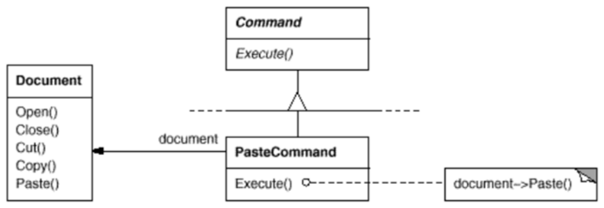
Some Examples:

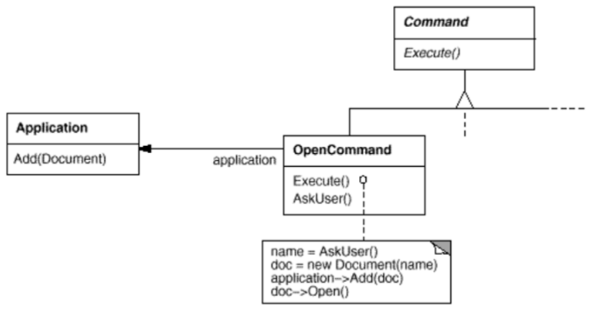
- Command
- Observer
- Strategy
- State

Command

Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.







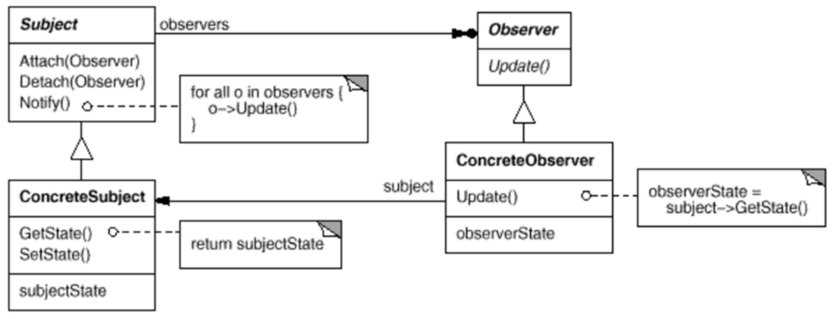
Undoability: Add an **Unexecute** method to Command interface.

Observer

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. (publish-subscribe)

• Strength
– Flexible
– Low coupling between subject and observer
– Broadcast updates

• Weakness
– Interaction between subjects and observer can be complex
– Order of notification: not determined

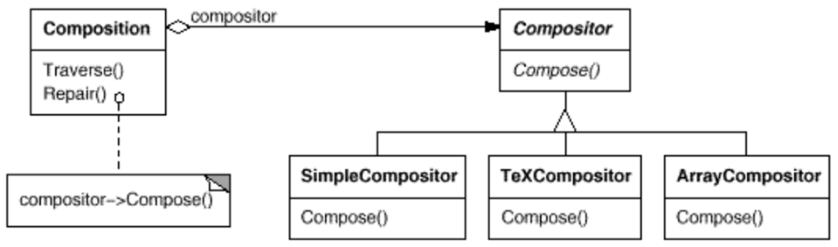


Strategy

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

• Strength
– Provides alternative w/o subclassing
– Eliminate large conditional statements
– Choice of implementation for the same behavior

• Weakness
– More objects
– All algorithms have the same interface



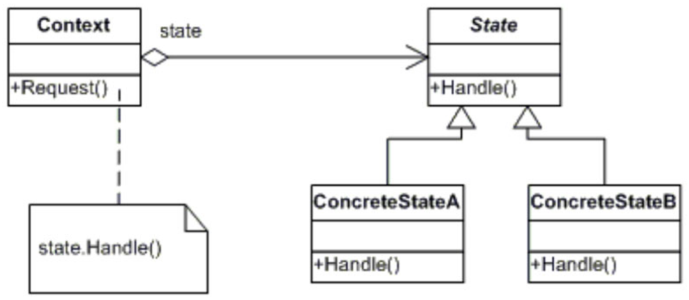
State

Intent: Allow an object to alter behavior when its internal state changes

Applicable when:

– An object behavior depends on its state, and it must change its behavior at run-time depending on that state.

– Operations have large, multipart conditional statements that depends on object’s state. In this case, state pattern puts each branch of the conditional in separate classes.


Context
+Request()
state
State
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()