**Definition:**

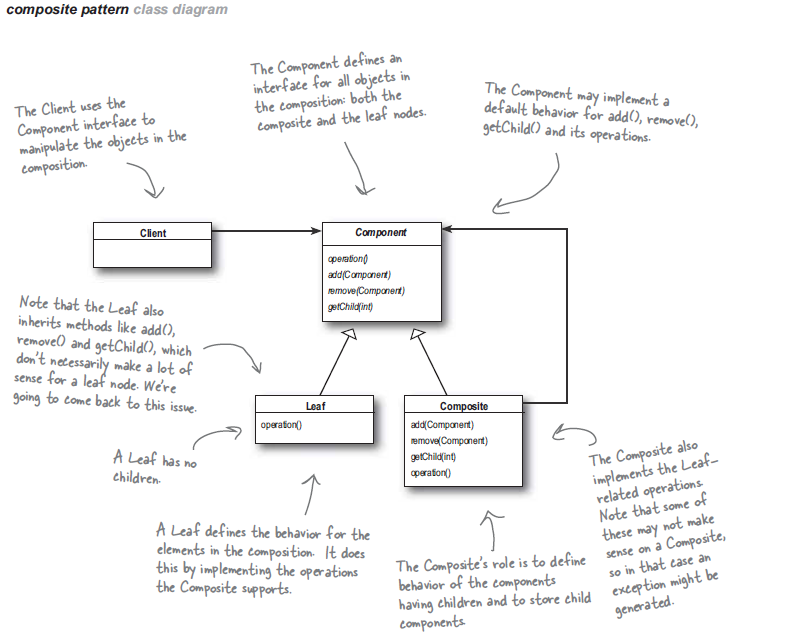
The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**The Composite Pattern has four participants:**

* 1. **Component** – Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
  2. **Leaf** – Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
  3. **Composite** – Composite stores child components and implements child related operations in the component interface.
  4. **Client** – Client manipulates the objects in the composition through the component interface.

Composite and leaf are derived from Component.

Composite is a container class - which can store many other Components.



**Intent**

* Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
* Recursive composition
* "Directories contain entries, each of which could be a directory."

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently.

Examples:

* Menus that contain menu items, each of which could be a menu.
* Directories that contain files, each of which could be a directory.
* Containers that contain Elements, each of which could be a Container.

**Related Patterns**

* Decorator Pattern - Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Example:

The program shows an Electronic system.

The system contains the following type of components.

\* Resistor

\* Capacitor

\* ICChip

\* PCB

All these components share the common features of "**ElectronicComponet**" abstract class.

In terms of Composite pattern, the class "**ElectronicComponent**" acts as the "Component".

The Resistor & Capacitor classes acts like Leafs since they don't have further subdivision.

ICChip s contains many Resistors and Capacitors inside it, so it will be a Composite of Electronic Components.

Furthermore, the PCB (Printed Circuit board) can contains, Resistor, Capacitor and ICChips on it. So that also is a Composite Electronic Component.

Simply, all Electronic Components has some common feature that it accept a Voltage

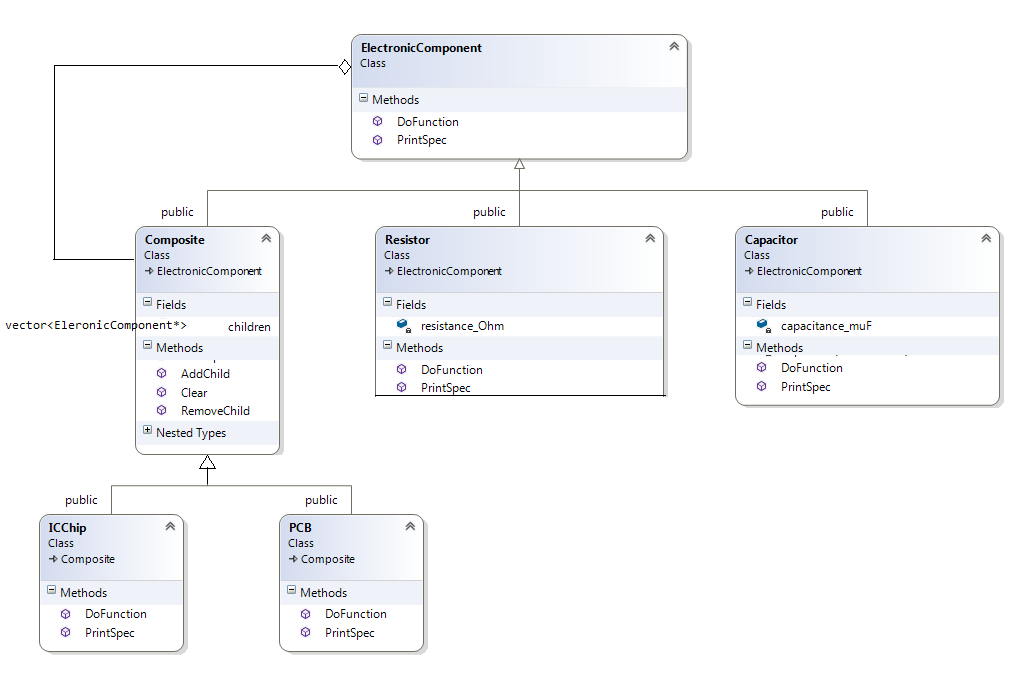
And Current as Input and generates a varied value of Voltage and current as Output.

So a function named DoFunction() is there which does the components actual processing.

Also all electronic components will have a specification.

For e.g. Resister has X number of Ohms, and Capacitor has X micro fared etc.

So a function named PrintSpec() is there which prints the component spec.



*Relationship between component & composite.*

Composite has a component. But both are independent to each other. Composite have an collection of component which is not created inside the Composite class.

What about the ownership? (Definitely not a hard ownership, composition)

Suppose a chip using register r1 & capacitor c1. Both chip & components(r1,c1) are independent to each other, but if a chip using r1 & c1 one thing is sure no other chip will use same component at that time. To use with other chipset we need to disassemble r1 & c1 first then after only component integrate with other chip.

So, we can say that Composite have logic ownership on component but lifetime of both are independent to each others. And this relationship called aggregation.

//The COMPONENT class which declares the common functionalities of all Electronic //Components.

class ElectronicComponent

{

public:

virtual void PrintSpec(ostream& ostr, string prefix = "") = 0;

virtual void DoFunction(float& voltage, float& current) = 0;

};

//Leaf Class Register & Capacitor which extends ElectronicComponent.

class Resistor :public ElectronicComponent

{

float resistance\_Ohm; // Resistance value of resistor.

Resistor() {}

public:

Resistor(float ohm) :resistance\_Ohm(ohm) {}

void PrintSpec(ostream& ostr, string prefix = "")

{

ostr << prefix.c\_str() << "Resistor (" << resistance\_Ohm << " Ohm )\n";

}

//Performs the Resistor's real function of modifying input Voltage & Current

//Now I just put a printing of Voltage and current as place holder.

void DoFunction(float& voltage, float& current)

{

cout << endl << "Resistor Input (" << voltage << " V ," << current << "

Amp) Resistance = " << resistance\_Ohm << endl;

}

};

class Capacitor :public ElectronicComponent

{

float capacitance\_muF; //Capacitance value in micro fared.

Capacitor() {}

public:

Capacitor(float muF) :capacitance\_muF(muF){}

void PrintSpec(ostream& ostr, string prefix = "")

{

ostr << prefix.c\_str() << "Capacitor (" << capacitance\_muF << " muF )\n";

}

void DoFunction(float& voltage, float& current)

{

cout << endl << "Capacitor Input (" << voltage << " V ," << current << " Amp) Capacitance = " << capacitance\_muF << endl;

}

};

class Composite : public ElectronicComponent

{

protected:

vector<ElectronicComponent\*> children; //Child components list.

public:

//An iterator type is defined which can be used from Composite class

//To perform extensive operations on the Children list like Insert, Search etc.

//This is provided just for Ease of use for Composite class developers.

typedef typename vector<ElectronicComponent\*>::iterator ChildIterator;

Add a Component to the child list. Check to avoid duplication of same child in the

list.

void AddChild(ElectronicComponent\* child)

{

ChildIterator itr = find(children.begin(), children.end(), child);

if (itr == children.end())

{

children.push\_back(child);

}

}

//Remove a component from the child list if it exist there.

void RemoveChild(ElectronicComponent\* child)

{

ChildIterator itr = find(children.begin(), children.end(), child);

if (itr != children.end())

{

children.erase(itr);

}

}

//Remove all child from the list.

void Clear()

{

children.clear();

}

virtual ~Composite()

{

}

};

ICChip is a COMPOSITE class which consists of many ElectronicComponents -such as Resistors & Capacitors, so it can be said a ***Composite of ElectronicComponents***.

Thus it is inherited from -Composite. This makes it a container class of other ElectronicComponents and at the same time it, it by itself an ElectronicComponent.

class ICChip :public Composite

{

public:

void PrintSpec(ostream& ostr, string prefix = "")

{

ostr << prefix.c\_str() << "ICChip With " << children.size() <<

" Components " << endl;

for (unsigned int i = 0; i < children.size(); i++)

{

ostr << prefix.c\_str();

children[i]->PrintSpec(ostr, "\t");

}

}

//Real functioning of ICChip is achieved by passing its input Voltage & Current

//To the sub components inside it.So the DoFunction - of ICChip passes its input

//Voltage & Current to all the child component's DoFunction .

void DoFunction(float& voltage, float& current)

{

cout << endl <<"ICChip Input (" << voltage << " V ," << current << " Amp)";

for (unsigned int i = 0; i < children.size(); i++)

{

children[i]->DoFunction(voltage, current);

}

}

};

//Same as ICChip

class PCB :public Composite

{

public:

void PrintSpec(ostream& ostr, string prefix = "")

{

ostr <<prefix.c\_str() << "PCB With " <<children.size() << " Components\n; "

for (unsigned int i = 0; i < children.size(); i++)

{

ostr << prefix.c\_str();

children[i]->PrintSpec(ostr, "\t");

}

}

Real functioning of PCB is achieved by sending its input Voltage & Current

To the sub components inside it. So the DoFunction - of PCB send its input Voltage

& Current to all the child components inside it.

void DoFunction(float& voltage, float& current)

{

cout << endl << "PCB Input (" << voltage << " V ," << current << " Amp) ";

for (unsigned int i = 0; i < children.size(); i++)

{

children[i]->DoFunction(voltage, current);

}

}

};

int main()

{

Resistor r1(50), r2(70); //Define Resistors

Capacitor c1(200), c2(300); //Define Capacitors

ICChip ic1; //Create a Chip

ic1.AddChild(new Resistor(2000)); // Add a Resistor inside the ICChip

ic1.AddChild(new Capacitor(1000)); // Add a Capacitor inside the ICChip

PCB pcb1; //Make PCB Object and add the Resistor, Capacitors and ICChip on it

pcb1.AddChild(&r1);

pcb1.AddChild(&c1);

pcb1.AddChild(&c2);

pcb1.AddChild(&ic1);

pcb1.AddChild(&ic1); // Duplicate child entries are ignored.

cout << "\n=========== Printing the PCB Spec ==========" << endl;

pcb1.PrintSpec(cout);

float v = 110, i = 5;

cout << "\n=========== DoFunction(110,5) of PCB ==========" << endl;

pcb1.DoFunction(v, i);

cout << "\n=========== Removing c2 from PCB ==========" << endl;

pcb1.RemoveChild(&c2);

cout << "\n=========== Printing the PCB Spec ==========" << endl;

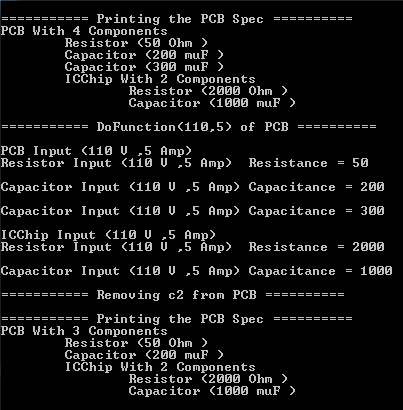
pcb1.PrintSpec(cout);

getchar();

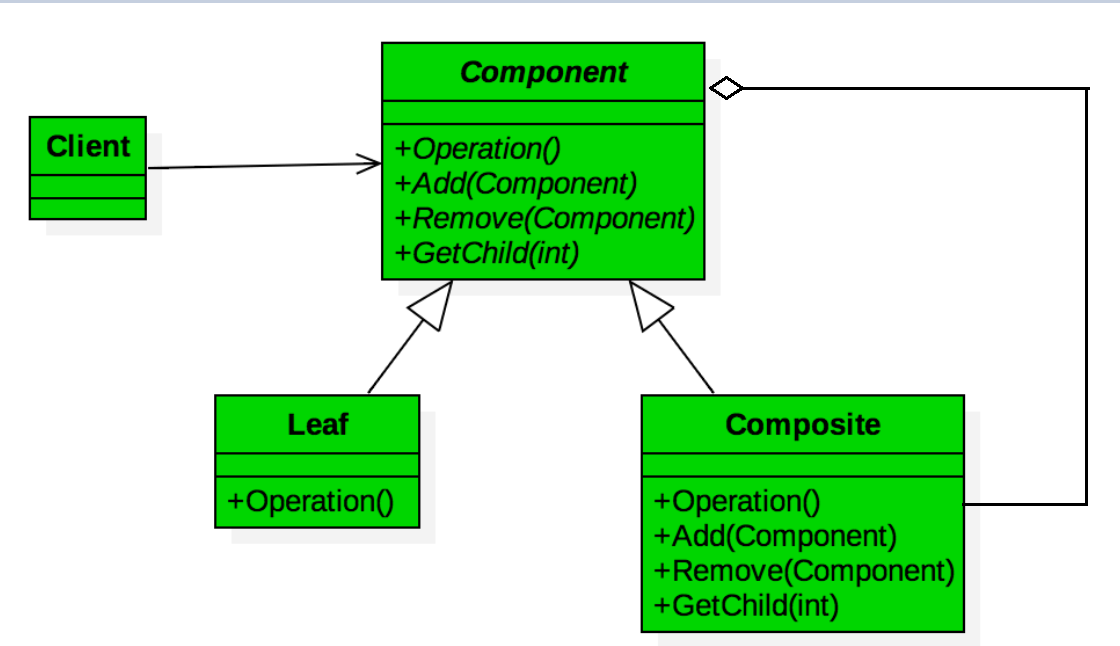
return 0;

}

Output:



Example 2: Library Book collection



When looking at the class diagram for the Composite Pattern, we can see there is a common component interface, Component, for accessing our individual objects (Leaf) and composition of objects (Composite). As the definition of the pattern states, this interface allows clients to treat the Leaf and the Composite objects uniformly.

This is a common pattern for Menu like design where you would have a common MenuComponent interface with a MenuItem as your Leaf and a Menu or SubMenu representing the Composite object of multiple MenuComponents.

class BookComponent

{

public:

BookComponent(){}

virtual ~BookComponent(){}

virtual void Add(BookComponent \* newComponent){}

virtual void Remove(BookComponent \* newComponent){}

virtual void DisplayInfo() = 0;

};

class BookGroup : public BookComponent {

public:

BookGroup(std::string groupName);

virtual ~BookGroup();

virtual void Add(BookComponent \* newComponent);

virtual void Remove(BookComponent \* componentToRemove);

virtual void DisplayInfo();

private:

std::string m\_groupName;

std::vector<BookComponent\*> m\_bookComponents;

};

class Book : public BookComponent {

public:

Book(std::string bookTitle, std::string author);

virtual ~Book();

virtual void DisplayInfo();

private:

std::string m\_title;

std::string m\_author;

};

class Librarian {

public:

Librarian();

virtual ~Librarian();

void DisplayBookCollection();

private:

void BuildBookCollection();

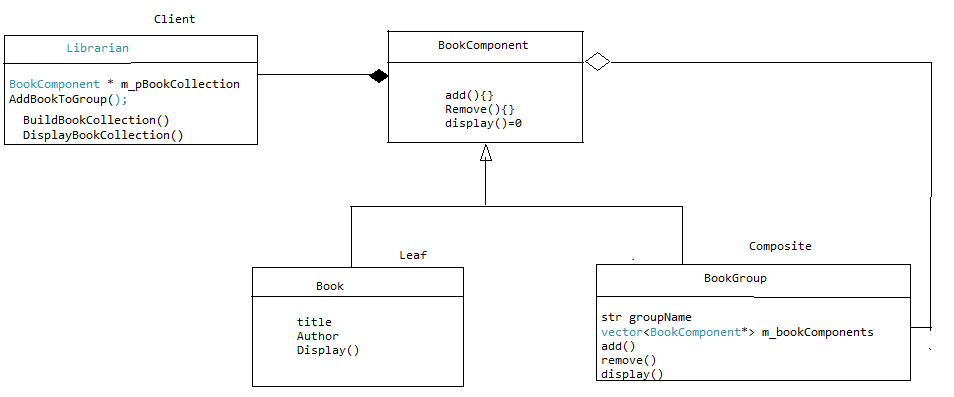
void AddBookToGroup(BookComponent \* group,string bookTitle, string author);

BookComponent \* m\_pBookCollection;

};

If you compare with previous example, you may notice the base class of example 1 having all its methods are pure virtual but in above example having one method is pure virtual.

Idea is very simple we don’t need add(), remove() likes method for leaf class, for that either we don’t include these methods in base class (component) or provide empty definition of these methods. So when we extend base class for leaf doesn’t need to provide definition at all and for composite class just override those methods.



We would like to make our book collection like this, where all bold words are Group (composite) others are leafs (Books)

***Technical***

***CS***

Operating system, Mr. Dhamdhre

Autometa theory, Linz

Algorithm, Corman

***Electronics***

Circuit theory, Known

Discrete Math, Known

***Non-Tech***

***Literature***

***Hindi***

***Story***

Karmbhumi, M. Premchandra

***Poem***

Madushala, Bachchan

Urvashi, R.Dinkar

Kurkchetra, R.Dinkar

***English***

***Story***

***Poem***

***Biography***

Fire of wings, APJ Kalam

The Story Of My Experiments With Truth, Gandhi

**Book.cpp**

Book::Book(std::string title, std::string author): m\_title(title), m\_author(author){}

Book::~Book(){}

void Book::DisplayInfo()

{

std::cout << "Book : " << m\_title << " by " << m\_author << "\n";

}

**BookGroup.cpp**

BookGroup::BookGroup(std::string groupName): m\_groupName(groupName){}

BookGroup::~BookGroup() {

std::vector<BookComponent \*>::iterator pos;

for(pos = m\_bookComponents.begin();

pos != m\_bookComponents.end(); ++pos)

{

BookComponent \* bookComponent = \*pos;

delete bookComponent;

}

m\_bookComponents.clear();

}

void BookGroup::Add(BookComponent \* newComponent)

{

m\_bookComponents.push\_back(newComponent);

}

void BookGroup::Remove(BookComponent \* componentToRemove)

{

std::vector<BookComponent \*>::iterator pos;

for(pos = m\_bookComponents.begin();

pos != m\_bookComponents.end(); ++pos)

{

if(\*pos == componentToRemove)

{

m\_bookComponents.erase(pos);

break;

}

}

}

void BookGroup::DisplayInfo()

{

static std::string spaces;

std::cout << "Group : " << m\_groupName << "\n";

spaces += std::string(" ");

std::vector<BookComponent \*>::iterator pos;

for(pos = m\_bookComponents.begin();

pos != m\_bookComponents.end(); ++pos)

{

std::cout << spaces;

BookComponent \* bookComponent = \*pos;

bookComponent->DisplayInfo();

}

spaces.pop\_back();

spaces.pop\_back();

spaces.pop\_back();

spaces.pop\_back();

}

**Librarian.cpp**

Librarian::Librarian()

: m\_pBookCollection(new BookGroup(std::string("Book Collection")))

{

BuildBookCollection();

}

Librarian::~Librarian() {

delete m\_pBookCollection;

}

void Librarian::BuildBookCollection()

{

BookComponent \* tech = new BookGroup(std::string("Technical Books"));

BookComponent \* cs = new BookGroup(std::string("Computer Science"));

BookComponent \* ecs = new BookGroup(std::string("Electronics & Computer

Science"));

BookComponent \* nTech = new BookGroup(std::string("Non-Technical Books"));

BookComponent \* literature = new BookGroup(std::string("Literature"));

BookComponent \* hindi = new BookGroup(std::string("HindiBooks"));

BookComponent \* hStory = new BookGroup(std::string("Hindi Story Books"));

BookComponent \* hPoem = new BookGroup(std::string("Hindi Poem Books"));

BookComponent \* english = new BookGroup(std::string("EnglishBooks"));

BookComponent \* eStory = new BookGroup(std::string("English Story Books"));

BookComponent \* ePoem = new BookGroup(std::string("English Poem Books"));

BookComponent \* bio = new BookGroup(std::string("Biography"));

m\_pBookCollection->Add(tech);

tech->Add(cs);

AddBookToGroup(cs, std::string("Operating system"), std::string("Mr. DhamDhre"));

AddBookToGroup(cs, std::string("Automata theory "), std::string("Linz."));

AddBookToGroup(cs, std::string("Algorithm"), std::string("Mr. Corman"));

tech->Add(ecs);

AddBookToGroup(ecs, std::string("Circuit Theory"), std::string("Known."));

AddBookToGroup(ecs, std::string("Discrete Math" ), std::string("Known."));

m\_pBookCollection->Add(nTech);

nTech->Add(literature);

literature->Add(hindi);

hindi->Add(hStory);

AddBookToGroup(hStory, std::string("Karmbhumi"), std::string("M. Premchandra"));

hindi->Add(hPoem);

AddBookToGroup(hPoem, std::string("Madhushala"), std::string("HR. Bachchan"));

AddBookToGroup(hPoem, std::string("Urvashi"), std::string("RS. Dinker"));

AddBookToGroup(hPoem, std::string("Kurukchetra"), std::string("RS. Dinker"));

literature->Add(english);

nTech->Add(bio);

AddBookToGroup(bio, std::string("Wings of fire"), std::string("APJ. Kalam"));

AddBookToGroup(bio, std::string("The Story Of My Experiments With Truth"),

std::string("Gandhi"));

}

void Librarian::AddBookToGroup(BookComponent \* group,

std::string bookTitle, std::string author)

{

BookComponent \* book = new Book(bookTitle, author);

group->Add(book);

}

void Librarian::DisplayBookCollection()

{

m\_pBookCollection->DisplayInfo();

}

**Driver Code:**

int main()

{

Librarian librarian;

librarian.DisplayBookCollection();

getchar();

}

**Output of a program:**

