Design Pattern – Mẫu thiết kế hướng đối tượng

CSC10003 – Phương pháp lập trình hướng đối tượng

Nội dung thực hành lần này sẽ tập trung phân tích và hướng dẫn một số mẫu thiết kế hướng đối tượng thường sử dụng trong lập trình

1. Singleton Design Pattern

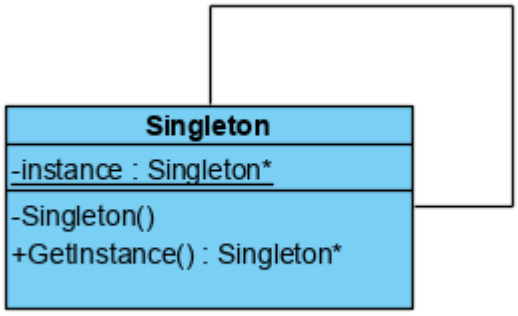
- Singleton là mẫu thiết kế trong lập trình hướng đối tượng thuộc vào nhóm khởi tạo (Creational).
- Mục tiêu của mẫu thiết kế này:
  - **Mỗi lớp đảm bảo chỉ có một thể hiện duy nhất** trong suốt chương trình chạy.
  - Cung cấp cách thức **truy xuất toàn cục**
- Vì sao cần đảm bảo chỉ có một thể hiện duy nhất của lớp này?
  - Để cần chỉ có một đối tượng chính xác điều phối trên toàn hệ thống
    - Ví dụ: trong bài toán về quản lý nhân sự của trường đài học, đảm bảo chỉ có duy nhất một đối tượng quản lý danh sách nhân sự chung cho toàn trường.
    - Ví dụ: chỉ có thể mở được một Task Manager trên Windows để tránh việc tranh giành tài nguyên
- Cách tiếp cận cho việc cài đặt sử dụng hướng đối tượng:
  - Để đảm bảo chỉ có duy nhất 1 thể hiện của lớp trong suốt chương trình chạy, cần phải chặn việc tạo mới đối tượng (không cho phép new trực tiếp)
    - Ý tưởng: để chặn việc tạo mới đối tượng (new), **cần đặt phương thức tạo lập vào private**
    - Cung cấp phương thức public để lấy thể hiện của lớp này. Nếu chưa có thì tạo mới 1 lần, nếu có rồi thì trả về thể hiện duy nhất này.
    - Thuộc tính của lớp chính là biến con trỏ trỏ đến thể hiện duy nhất của lớp.
  - Để cung cấp cách thức truy xuất toàn cục để người dùng chỉ được phép sử dụng đối tượng sẵn có này, cần phải khai báo thuộc tính và phương thức mang tính chất toàn cục.
    - Nếu xài biến toàn cục trong chương trình, vậy sẽ mất đi tính chất đóng gói (encapsulation) của hướng đối tượng
    - Ý tưởng: **đặt thuộc tính và phương thức của lớp này là thành phần tĩnh** (static).
    - Thành phần tĩnh (static) là thành phần chung cho mọi lớp đối tượng tương ứng, do chỉ có duy nhất 1 lớp đối tượng, việc sử dụng thành phần tĩnh rất phù hợp và vẫn giữ được tính chất của hướng đối tượng

```
//Lớp Singleton chỉ cho phép tạo 1 đối tượng duy nhất
class Singleton
{
private:
	//Đối tượng duy nhất của lớp này, là thuộc tính kiểu con trỏ, tĩnh (static)
	static Singleton* instance;
	//Các thuộc tính khác ...
	//Đặc điểm: phương thức tạo lập phải để trong private
	Singleton()
	{

	}

public:
	//Public phương thức lấy đối tượng duy nhất này
	//Nếu chưa có thì new 1 lần, nếu có rồi thì trả lại đối tượng
	static Singleton* GetInstance()
	{
		if (instance == NULL)
		{
			instance = new Singleton();
		}
		return instance;
	}

	//Public phương thức để xóa đối tượng duy nhất tránh memory leak
	//Nếu chưa có thì không làm gì hết, có rồi thì delete và trả về null
	static void DeleteInstance()
	{
		if (instance)
		{
			delete instance;
			instance = NULL;
		}
	}
	//Nếu delete trong destructor dẫn đến infinite loop do tự xóa chính nó (static)
	~Singleton()
	{
		//Xóa các thuộc tính khác ngoài thuộc tính static
	}

	//Các phương thức khác
};
```

- Thành phần tĩnh (Static) :
  - Thành phần xài chung của tất cả đối tượng có chung lớp
  - Không thuộc vào một đối tượng cụ thể
  - Nếu sử dụng trong lớp, yêu cầu cần phải khởi tạo trước 1 lần khi sử dụng
- Khi khai báo thuộc tính kiểu con trỏ của thành phần tĩnh, không được phép xóa trong phương thức hủy, vì tính chất xài chung cho tất cả đối tượng, khi gọi destructor sẽ gặp tình trạng tự bản thân thành phần này gọi xóa chính nó, dẫn đến việc lặp vô hạn !!!
  - Cách xử lý để tránh gặp rò rỉ bộ nhớ : tạo một phương thức tĩnh riêng để xử lý việc xóa tương ứng
- Sau khi cài lớp Singleton, ví dụ sau đây cho thấy việc sử dụng lớp này :

```
//Gán lần đầu khi mới vào chương trình là NULL (do biến static)
Singleton* Singleton::instance = NULL;

void main()
{
	//Gọi tạo đối tượng 1 lần duy nhất
	Singleton* object = Singleton::GetInstance();

	//Các thao tác trên đối tượng duy nhất này...

	//Thực hiện gọi xóa đối tượng duy nhất này
	//Xóa thành phần tĩnh KHÔNG cài qua destructor
	Singleton::DeleteInstance();
}
```

- Mô hình UML cho mẫu thiết kế này :
  - Thành phần static ký hiệu gạch chân



2. Prototype Design Pattern

- Prototype là mẫu thiết kế trong lập trình hướng đối tượng thuộc vào nhóm khởi tạo (Creational).
- Đặt vấn đề: thông thường, việc tạo mới hay sao chép một đối tượng khác có thể được gọi thông qua lệnh new hay phương thức tạo lập sao chép. Điều này dẫn đến các vấn đề:
  - Nếu đối tượng đang chứa bên trong biến con trỏ của lớp cha (kế thừa), vậy làm sao để biết được chính xác đối tượng con bên trong để thực hiện sao chép? (bài toán nhân bản đối tượng)
    - Ví dụ: lớp con mèo, con heo kế thừa từ lớp động vật, làm sao để biết biến con trỏ DongVat* chứa lớp con nào bên trong? Ví dụ đoạn mã nguồn sau

```
void main()
{
	DongVat* dv = new ConMeo();
	DongVat* dv2 = new ConHeo();
	//Làm sao để biết được lớp con chứa bên trong biến con trỏ DongVat
là lớp nào để copy?
	//… Các xử lý khác
}
```

  - Khi tạo một loại đối tượng trong thao tác nhập, lúc nào cũng cần phải dựa trên việc đánh số để phân loại và dựa vào sử dụng kiểm tra if else hoặc switch case nhiều lần để phân loại đối tượng. Nếu có một đối tượng khác, phải vào trong đoạn mã nguồn này thay đổi thêm vào, khiến cho mã nguồn không còn linh hoạt (bài toán tạo đối tượng nhờ tham số chứa tên lớp)

```
void main()
{
	DongVat* dv;
	cout << "Nhap vao loai dong vat can tao: 1. Con Meo 2. Con Heo\n";
	int LoaiDV = 0;
	cin >> LoaiDV;
	if (LoaiDV == 1) dv == new ConMeo();
	else if (LoaiDV == 2) dv == new ConHeo();
	//Nếu có thêm con vịt, viết thêm if else nữa
	//else if (LoaiDV == 3) dv == new ConVit();
}
```

- Giải pháp cho cả 2 vấn đề nêu trên: tạo ra đối tượng bản mẫu (Prototype) bằng cách thực hiện nhân bản đối tượng (Clone) do lớp con kế thừa xử lý sử dụng đa hình

- Lợi ích của mẫu Prototype:
  - Thực hiện tạo ra đối tượng lớp con mới mà không cần biết quá nhiều chi tiết bên trong lớp con đó (encapsulation)
  - Có khả năng mở rộng thêm mới đối tượng mà không cần phải thay đổi mã nguồn quá nhiều trong phần xử lý chung (điển hình là xử lý nhập thông tin)
- Ý tưởng thực hiện:
  - Cài đặt phương thức tạo lập sao chép cho mỗi lớp con cụ thể, vì nếu không cài đặt, khi gọi nhân bản bằng copy constructor sẽ gọi mặc định, điều này khá nguy hiểm khi bên trong thuộc tính của lớp là thuộc tính con trỏ!!
  - Cài đặt đa hình phương thức nhân bản Clone (có thể đặt tên khác nhưng ý nghĩa của phương thức vẫn là nhân bản) tại từng lớp con cụ thể để tự lớp con xử lý tạo bản sao và chép các thuộc tính của đối tượng lớp đó vào bảo sao.
- Cách giải quyết cho bài toán nhân bản đối tượng mà không cần biết kiểu lớp con bên trong:
  - Cài đặt copy constructor (có thể yêu cầu default constructor) ở lớp con
  - Cài đặt phương thức nhân bản đa hình (có thể thuần ảo) ở lớp cha
  - Cài đặt phương thức nhân bản ở lớp con để tự tạo bản mẫu sao chép

```
class DongVat
{
public:
	//Phương thức nhân bản cài đặt đa hình
	virtual DongVat* Clone() = 0;
};

class ConMeo : public DongVat
{
public:
	ConMeo() {}
	//Copy constructor
	ConMeo(const ConMeo& meo)
	{
		//Xử lý copy constructuor
	}
	//Trả về đối tượng cần nhân bản theo cơ chế đa hình
	DongVat* Clone()
	{
		//Gọi copy constructor để nhân bản
		return new ConMeo(*this); //Copy lại toàn bộ
	}
};
```

```
class ConHeo : public DongVat
{
public:
	ConHeo() {}
	//Copy constructor
	ConHeo(const ConHeo& heo)
	{

	}
	//Trả về đối tượng cần nhân bản theo cơ chế đa hình
	DongVat* Clone()
	{
		//Gọi copy constructor để nhân bản
		return new ConHeo(*this);//Copy lại toàn bộ
	}
};
DongVat* NhanBanDongVat(DongVat* dv)
{
	//Mã nguồn không thay đổi khi thêm bất kỳ loại động vật khác
	if (dv == NULL) return NULL;
	return dv->Clone(); //Trả về nhân bản đúng loại do gọi đa hình
}
```

- Cách giải quyết cho bài toán tạo đối tượng nhờ tham số chứa tên lớp:
  - Cài đặt copy constructor (có thể yêu cầu default constructor) ở lớp con
  - Cài đặt phương thức nhân bản đa hình (có thể thuần ảo) ở lớp cha
  - Cài đặt phương thức nhân bản ở lớp con để tự tạo bản mẫu sao chép
  - Cài đặt phương thức xác định tên lớp ở từng lớp con kế thừa
  - Khai báo mảng các đối tượng bản mẫu ở lớp cha (sử dụng static), mỗi lần có đối tượng mới được tạo, cần phải "đăng ký" vào bản mẫu để cho phép nhân bản
  - Hỗ trợ phương thức tạo đối tượng với tham số là chuỗi tên đối tượng ở lớp cha, bên trong xử lý tìm kiếm so sánh theo tên và trả về nhân bản tương ứng của đối tượng (mà không cần phải biết loại đối tượng cụ thể)

```
class DongVat
{
	static vector<DongVat*> DoiTuongMau; //vector các đối tượng bản mẫu
public:
	//Phương thức nhân bản cài đặt đa hình
	virtual DongVat* Clone() = 0;
	//Method lấy tên lớp đối tượng tương ứng
	virtual string LayTenDoiTuong() = 0;
```

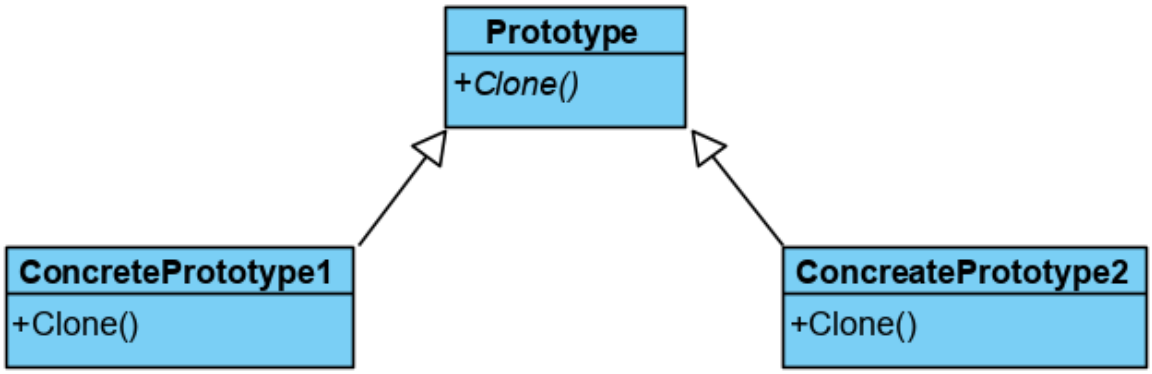
```
	//Method thêm đối tượng bản mẫu vào danh sách bản mẫu (static)
	static void ThemDoiTuongMau(DongVat* ns)
	{
		if (ns == NULL) return;
		int pos = -1;

		// Tìm đối tượng dựa theo tên
		//Nếu không có tên, vậy đối tượng mẫu chưa được thêm vào
		//Tiến hành thêm vào đối tượng mẫu
		for (int i = 0; i < DoiTuongMau.size(); i++)
		{
			DongVat* mau = DoiTuongMau[i];
			if (mau->LayTenDoiTuong() == ns->LayTenDoiTuong())
				pos = i;
		}
		if (pos == -1) DoiTuongMau.push_back(ns);
	}
	//Method hỗ trợ tạo đối tượng dựa theo tên (static)
	static DongVat* TaoDoiTuongTheoTen(string tenNS)
	{
		//Tìm đối tượng dựa theo tên
		//nếu có tên đối tượng thì trả về bản clone
		for (int i = 0; i < DoiTuongMau.size(); i++)
		{
			DongVat* mau = DoiTuongMau[i];
			if (mau->LayTenDoiTuong() == tenNS)
				return DoiTuongMau[i]->Clone();
		}
		//Nếu tìm không có, trả về NULL;
		cout << "Khong co ten doi tuong hop le\n";
		return NULL;
	}

	//Method cho phép xóa tất cả đối tượng mẫu (Static)
	static void XoaDoiTuongMau()
	{
		for (int i = 0; i < DoiTuongMau.size(); i++)
		{
			delete DoiTuongMau[i];
		}
	}
	//Nếu delete trong destructor dẫn đến infinite loop do tự xóa chính nó
(static)
	virtual ~DongVat()
	{
		//Xóa các thuộc tính khác ngoài thuộc tính static
	}
};
```

```cpp
class ConMeo : public DongVat
{
public:
	ConMeo() {}
	//Copy constructor
	ConMeo(const ConMeo& meo)
	{
		//Xử lý copy constructuor
	}
	string LayTenDoiTuong()
//Method trả về tên lớp đối tượng
	{
		return "ConMeo";
	}
	//Trả về đối tượng cần nhân bản
theo cơ chế đa hình
	DongVat* Clone()
	{
	//Gọi copy constructor nhân bản
		return new ConMeo(*this);
	}
};
```

```cpp
class ConHeo : public DongVat
{
public:
	ConHeo() {}
	//Copy constructor
	ConHeo(const ConHeo& heo)
	{
		//Xử lý copy constructuor
	}
	string LayTenDoiTuong() //Method
trả về tên lớp đối tượng
	{
		return "ConHeo";
	}
	//Trả về đối tượng cần nhân bản
theo cơ chế đa hình
	DongVat* Clone()
	{
	//Gọi copy constructor để nhân bản
		return new ConHeo(*this);
	}
};
```

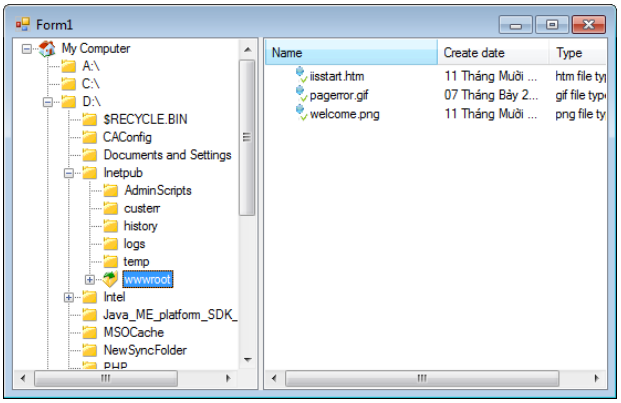
```cpp
//Khai báo trước 1 lần khi sử dụng thuộc tính static
vector<DongVat*> DongVat::DoiTuongMau;
void main()
{
	//Thêm vào đối tượng mẫu khi có 1 loại đối tượng mới ->Prototype
	DongVat::ThemDoiTuongMau(new ConMeo); //gọi qua static
	DongVat::ThemDoiTuongMau(new ConHeo);
	//DongVat::ThemDoiTuongMau(new ConVit); nếu có thêm loại động vật khác
	//Cài đặt việc nhập xuất ngắn gọn và hiệu quả hơn khi thêm động vật mới
	vector<DongVat*> TongDongVat;
	cout << "Nhap ten loai dong vat de them vao danh sach: ";
	string tenLoaiDV;
	cin >> tenLoaiDV;
	//Mã nguồn tại vị trí này không thay đổi khi thêm loại động vật mới
	//Phù hợp với tính chất đóng gói (encapsulation) của OOP
	//Khi thêm loại động vật mới vào, chỉ xử lý bên trong từng lớp
	//    và khai báo cài đặt theo đa hình (ảo, thuần ảo)
	DongVat* ns = DongVat::TaoDoiTuongTheoTen(tenLoaiDV);
	if (ns != NULL)
		TongDongVat.push_back(ns);
	cout << "So dong vat la: " << TongDongVat.size() << endl;
	for (int i = 0; i < TongDongVat.size(); i++)
		delete TongDongVat[i];
}
```

- Mô hình UML tổng quát cho mẫu thiết kế Prototype:



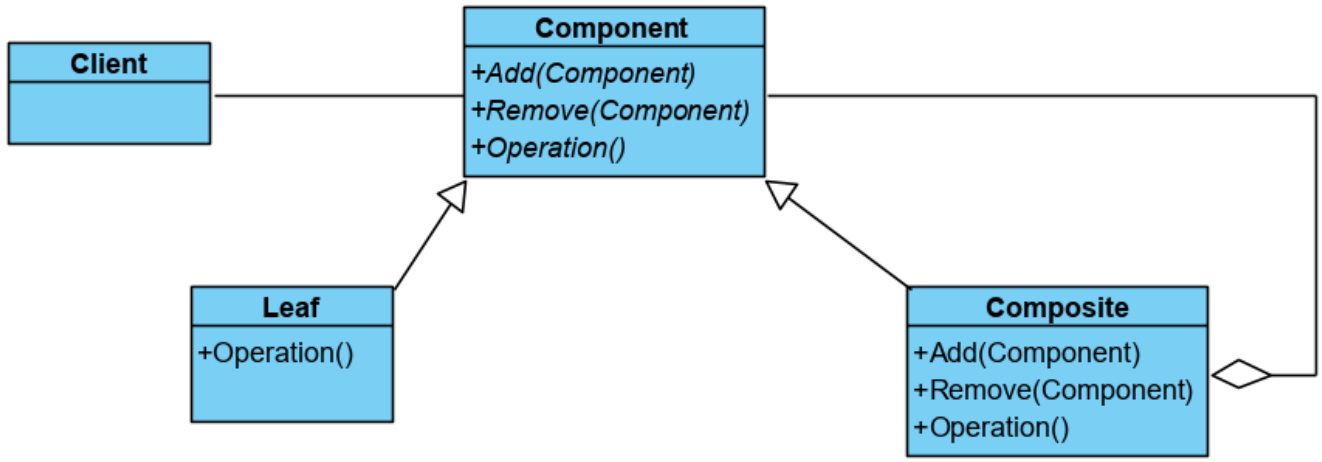
3. Composite Design Pattern

- Composite là mẫu thiết kế trong lập trình hướng đối tượng thuộc vào nhóm cấu trúc (Structural).
- Mẫu thiết kế Composite dùng để tạo ra các đối tượng mang tính chất phức hợp (Composite). Một đối tượng phức hợp Composite được tạo thành từ một hay nhiều đối tượng có chức năng tương tự nhau
  - Thao tác trên một nhóm đối tượng theo cách thao tác trên một đối tượng
- Nhận biết khi nào cần xài mẫu Composite:
  - Khi đối tượng cho phép chứa những loại đối tượng khác, bao gồm cả chính nó
  - Các thao tác của những đối tượng này có chức năng tương tự nhau
  - Mang tính chất đệ quy
- Mẫu Composite rất thích hợp cho việc xây dựng cấu trúc dạng phân lớp hay cấu trúc cây:
  - Ví dụ: trong hệ thống Windows Explorer, trong ổ đĩa sẽ chứa tập tin và folder, trong folder sẽ chứa tập tin và folder khác
    - Tập tin và folder đều có tên, dung lượng, có cách thức tính dung lượng tương tự nhau. (Nguồn ảnh từ internet)



- Ví dụ: trong bảng điện sẽ có mạch điện song song và mạch điện nối tiếp, trong mạch điện song song có thể chứa mạch điện nối tiếp và mạch điện song song khác
- Ví dụ: cấu trúc lưu trữ trang web dưới dạng file .html lưu bằng cách thẻ tag. Trong thẻ tag chứa thông tin của thẻ tag và có thể chứa các thẻ tag khác

- Ý tưởng thực hiện:
  - Cần phải xác định rõ các thành phần trong mẫu thiết kế Composite: gồm có 4 thành phần như sau:
    - Composite: thành phần phức hợp: chứa các loại thành phần đơn và bao gồm cả thành phần phức hợp (có tính đệ quy
      - Ví dụ: bài cây thư mục file, folder, trong folder có thể chứa file và chứa folder khác, vậy folder chính là thành phần phức hợp
      - Ví dụ: bài mạch điện, trong đó mạch điện song song có thể chứa mạch điện nối tiếp và mạch điện song song khác
    - Leaf: thành phần đơn: chỉ đối tượng đơn,
      - Ví dụ: tập tin (file) là thành phần đơn
    - Component : thành phần mang tính tổng quát, là đối tượng cha (thường là lớp trừu tượng) mang tính chất chung của các thành phần đơn và phức hợp
      - Ví dụ : file và folder đều có tên, dung lượng, vậy ta có thể có 1 component là DriveComponent chỉ chung hết tất cả các loại thành phần có thể có
    - Client : là nơi thực thi hoặc sử dụng các thành phần trên (hàm main,…)
  - Cài đặt trong lớp composite gồm 1 mảng chứa tất cả các thành phần chung (component), xử lý trên lớp composite bằng cách duyệt mảng
- Sơ đồ UML của mẫu thiết kế Composite :



```
class Component
{
protected:
	string name;
	float size;
public:
	virtual string GetName() = 0;
	virtual float GetSize() = 0;
	virtual void AddComponent(Component* ifile)
	{
		//Không xử lý gì tại chỗ này, để cho lớp Composite xử lý
		//Khai báo đa hình để gọi đệ quy trong Composite
	}
	virtual void RemoveComponent(Component* ifile)
	{
		//Không xử lý gì tại chỗ này, để cho lớp Composite xử lý
		//Khai báo đa hình để gọi đệ quy trong Composite
	}
	virtual ~Component()
	{
		//Nhớ đa hình phương thức hủy (hủy ảo)
	}
};

class Composite : public Component
{
	//Mảng chứa tất cả các component (bao gồm cả leaf và composite)
	vector<Component*> components;
public:
	string GetName()
	{
		return this->name;
	}
	//Cài đặt lại hàm GetSize, đệ quy trong Composite
	float GetSize()
	{
		float totalSize = 0;
		for (int i = 0; i < components.size(); i++)
			totalSize += components[i]->GetSize(); //đệ quy ngay tại vị trí này
	}
	void AddComponent(Component* iComp)
	{
		//Xử lý thêm 1 component vào vector<Component*>
	}
	void RemoveComponent(Component* iComp)
	{
		//Xử lý xóa 1 component ra vector<Component*>
	}
	~Composite() //Destructor xóa hết tất cả file và folder có trong folder
	{
		//Xử lý xóa hết toàn bộ Composite (đệ quy)
	}
};
```

```
class Leaf : public Component
{
public:
	string GetName()
	{
		return this->name;
	}
	float GetSize()
	{
		return this->size;
	}
	~Leaf() {}
};
```

4. Bài tập thực hành

Bài tập 1: Dựa trên bài tập 1 của tuần 8 – đa hình, sinh viên hãy cài đặt lại bài tập sử dụng kế thừa và đa hình với mẫu thiết kế Prototype thực hiện yêu cầu sau:

- Cài đặt thêm lớp đối tượng PhongQuanLy, đối tượng này nắm danh sách tất cả các nhân sự (kiểu con trỏ) hiện có của trường đại học ở dạng vector
- Dựa trên mẫu thiết kế Prototype, hãy cài đặt lại toàn bộ mã nguồn ở tất cả các lớp cho phép
  - Thêm vào một nhân sự thông qua tên loại nhân sự
  - Xóa một nhân sự dựa trên mã số nhân sự
- Lưu ý cần cài đặt để tránh rò rỉ bộ nhớ (memory leak)

Sinh viên vẽ sơ đồ lớp UML cho bài tập này.

Bài tập 2: Dựa trên bài tập 1 phía trên, sinh viên hãy cài đặt lại lớp PhongQuanLy sử dụng mẫu thiết kế Singleton để đảm bảo toàn bộ trường đại học chỉ có duy nhất một Phòng Quản Lý về tất cả nhân sự hiện có của trường đại học.

Bài tập 3: Sinh viên viết chương trình C++ cài đặt hệ thống lưu trữ ổ đĩa, thư mục, tập tin trong Windows Explorer sử dụng kế thừa và đa hình với mẫu thiết kế Composite. Mô tả chi tiết:

- Trong ổ đĩa sẽ có nhiều thư mục và tập tin. Ổ đĩa, thư mục và tập tin đều có tên.
- Trong thư mục sẽ có chứa nhiều tập tin và nhiều thư mục khác
- Mỗi tập tin gồm có thông tin là tên tập tin và dung lượng của tập tin đó (số byte)
- Yêu cầu:
  - Viết phương thức hỗ trợ liệt kê tất cả thư mục và tập tin có trong ổ đĩa
  - Viết phương thức tính tổng dung lượng của ổ đĩa
  - Vẽ UML cho bài tập này