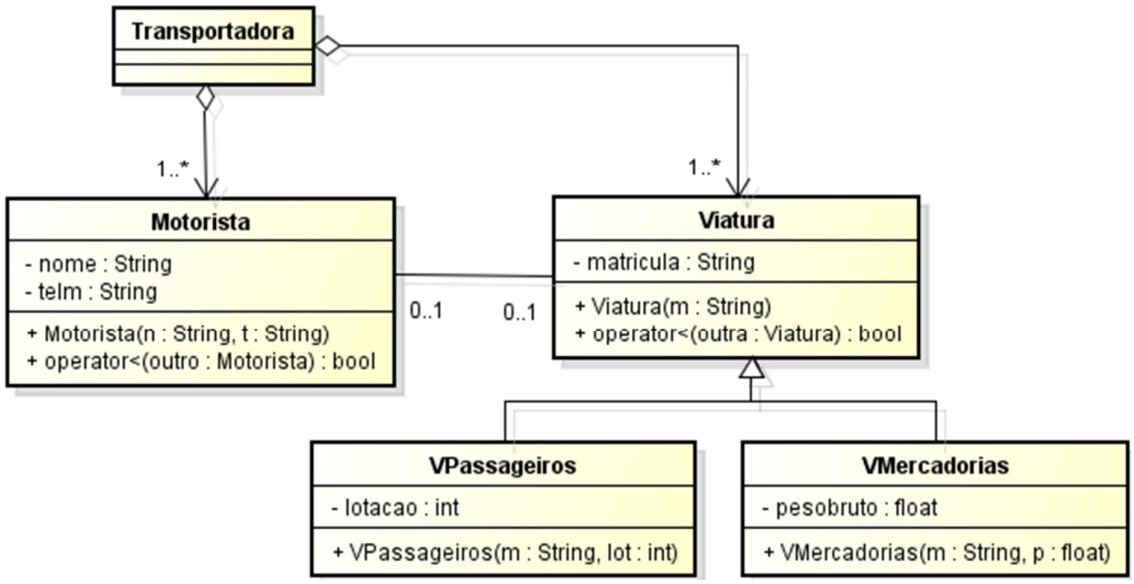
a) Apresente, através de um diagrama de classes em UML, a solução por si



b) Codifique em C++ a sua solução, tal como a descreveu no diagrama da anterior.

```
class Transportadora{
	Coleccao<Motorista> motoristas;
	ColeccaoHibrida<Viatura*> viaturas;
};

class Motorista{
	string nome;
	string telm;
	Viatura *viatura;
public:
	Motorista(const string &n, const string &t): nome(n), telm(t){
		viatura=NULL;
	}
	bool operator<(const Motorista &outro)const {return nome<outro.nome;}
};
class Viatura{
	string matricula;
	Motorista *motorista;
public:
	Viatura(const string &m): matricula(m){motorista=NULL;}
	virtual bool operator<(const Viatura &outra)const {
		return matricula<outra.matricula;
	}
};

class VPassageiros: public Viatura{
	int lotacao;
public:
	VPassageiros(const string &m, int lot): Viatura(m){lotacao=lot;}
};

class VMercadorias: public Viatura{
	float pesobruto;
public:
	VMercadorias(const string &m, float p): Viatura(m){pesobruto=p;}
};
```

c) Implemente os métodos necessários para adicionar (registar) à aplicação um novo motorista e um novo veículo de passageiros.

```
bool Transportadora::addMotorista(const string &n, const string &t){
	Motorista m(n,t);
	return motoristas.insert(m);
}

bool Transportadora::addVPassageiros(const string &m, int lot){
	VPassageiros *p = new VPassageiros(m,lot);
	return viaturas.insert(p);
}
```

d) Implemente o(s) método(s) necessários para afectar uma viatura a um motorista.

```
bool Transportadora::afectaViatMotorista(const string &m, conststring &n){
	Viatura *pv=findViatura(m);
	if(pv==NULL) {cout<<"viatura inexistente!\n"; return false;}
	else{
		Motorista *pm=findMotorista(n);
		if(pm==NULL) {cout<<"Motorista inexistente!\n"; return false;}
		else{
			pm->setViatura(pv);
			pv->setMotorista(pm);
			return true;
		}
	}
}
Viatura *Transportadora::findViatura(const string &m){
	Viatura v(m);
	return viaturas.find(&v);
}

Motorista *Transportadora::findMotorista(const string &n){
	Motorista m(n,"");
	return motoristas.find(m);
}

void Motorista::setViatura(Viatura *v){viatura=v;}

void Viatura::setMotorista(Motorista *m){ motorista=m;}
```

e) Implemente o(s) método(s) que permita(m) obter o número de telemóvel do motorista que conduza uma viatura com uma dada matrícula.

```
string Transportadora::getTelemMotViat(const string &m){
	Viatura *pv=findViatura(m);
	if(pv==NULL){
		cout<<"Viatura inexistente!";
		return "";
	}else if(pv->getMotorista()==NULL){
		cout<<"Viatura parada!";
		return "";
	}else return pv->getMotorista()->getTelem();
}

Motorista *Viatura::getMotorista() const{return motorista;}

string Motorista::getTelem() const{return telm;}
```

f) listar ... com indicação das matrículas e do nome dos respectivos condutores.

```
void Transportadora::listarViatEmCirc()const {
	ColeccaoHibrida<Viatura*>::iterator it;
	for(it=viaturas.begin(); it!=viaturas.end(); it++)
		if((*it)->getMotorista()!=NULL)
			cout<<(*it)->getMotorista()->getNome()
			    <<" conduz "<<(*it)->getMatricula()<<endl;
}

string Motorista::getNome() const{return nome;}

string Viatura::getMatricula() const{return matricula;}
```

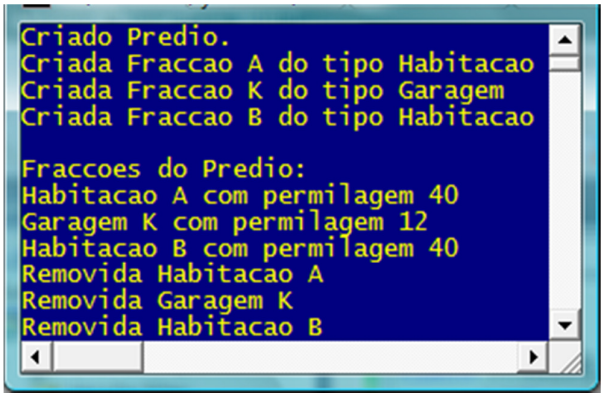
Grupo II

a) agregação/herança/agregação

b) não

c) `~Predio(){ for(int i=0; i<nfrac; i++) delete fraccoes[i];}`

d)



e)
Fraccoes do Predio:
A com permilagem 40
K com permilagem 12
B com permilagem 40

f)

```
void Predio::addLoja(char i, int p){
	if(nfrac<MAXFRAC) fraccoes[nfrac++]=new Loja(i,p);
}

class Loja: public Fraccao{
public:
	Loja(char i, int p): Fraccao(i,p){
		cout << " do tipo Loja" << endl;
	}
	void mostrar(){
		cout << "Loja ";
		Fraccao::mostrar();
	}
	~Loja(){ cout << "Removida Loja "<< id <<endl;}
};
```