Grupo I

a) Alguma das classes do problema é abstracta? Se sim, diga qual e por que motivo a considera. [0.7 val.]
A classe Conta, por ter um método virtual puro.

b) Diga de que métodos construtores dispõe cada uma das classes. [1.1 val.]
Conta: por defeito; de cópia; de conversão Conta(double s);
CPrazo: de cópia; CPrazo(double tx, double dep=0.0);
CPEspecial: por defeito; de cópia; de conversão CPEspecial(double tx);

c) Apresente o resultado que será visualizado na saída standard após a execução do programa. [3.2]
Abertura com deposito inicial 0 a Prazo Especial 200
Abertura com deposito inicial 500 a Prazo
Abertura com deposito inicial 0 a Prazo Especial 100
Especial Conta com taxa 3.5 fechada com saldo 200
Conta com taxa 2.5 fechada com saldo 500
Especial Conta com taxa 1.5 fechada com saldo 100

d) E o que seria visualizado se o método destrutor da classe Conta não fosse virtual? [1 val.]
~~Especial~~ Conta com taxa 3.5 fechada com saldo 200

e) Que tipo de alteração proporia à solução apresentada de modo a garantir … a mesma taxa de juro. [1 val.]
O atributo txJuro passaria a estático na classe CPrazo

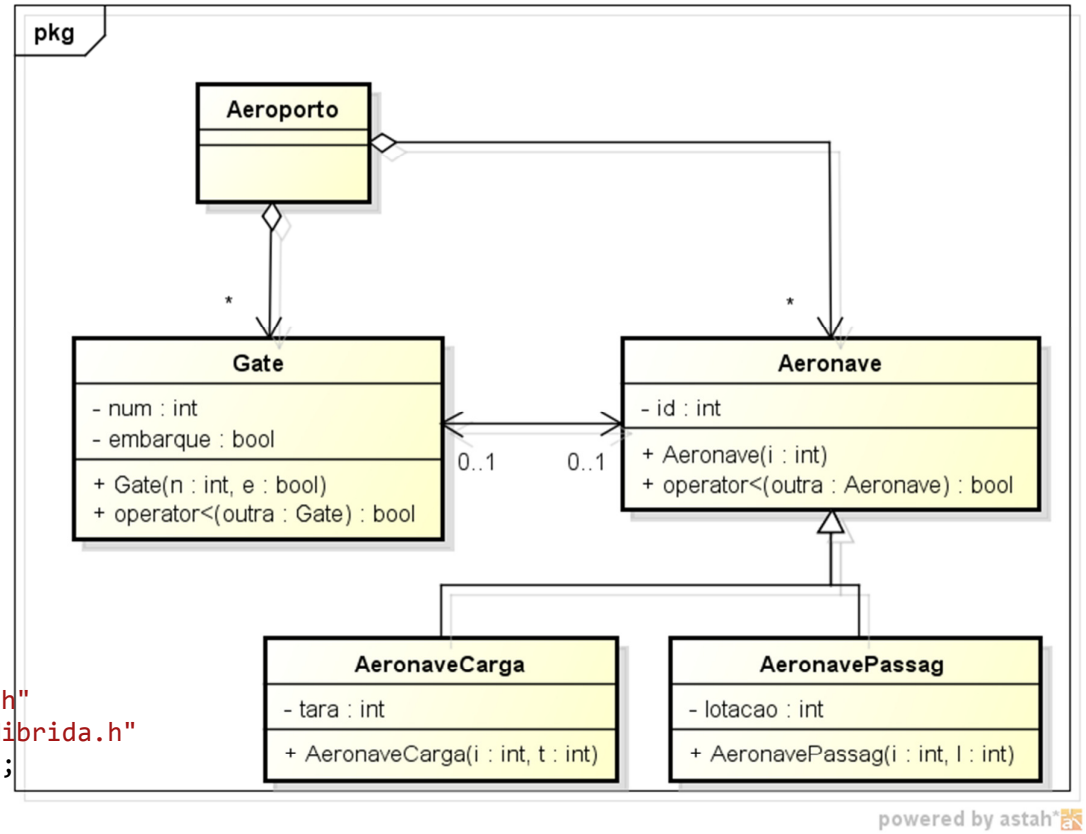
f) diga … se cada uma das instruções `c1=c2;` e `c1++;` é ou não válida no problema considerado. [1 val.]
c1=c2; válida – está a ser invocado o operador afectação que é gerado de forma autom
c1++; inválida – a classe CPEspecial não dispõe do operador incremento

g) Identifique dois erros presentes nesse código e corrija-os. [2 v.]

```
CPrazo contas[10]; -> a classe CPrazo não tem construtor por defeito

      CPrazo::CPrazo(){txJuro=0.0;}

contas[i]+=10; -> a classe CPrazo não tem o operador +=

      void operator+=(double val){saldo+=val;}
```

GRUPO II

a)



b)

```
#include <iostream>
#include "Coleccao.h"
#include "ColeccaoHibrida.h"
using namespace std;

class Aeroporto{
	Coleccao<Gate> gates;
	ColeccaoHibrida<Aeronave*> aeronaves;
};

class Gate{
	int num;
	bool embarque;
	Aeronave *aeronave;
public:
	Gate(int n): num(n){aeronave=NULL;}
	bool operator<(const Gate &outra)const {return num<outra.num;}
};

class Aeronave{
	int id;
	Gate *gate;
public:
	Aeronave(int i){id=i; gate=NULL;}
	virtual bool operator<(const Aeronave &outra)const {return id<outra.id;}
};

class AeronavePassag: public Aeronave{
	int lotacao;
public:
	AeronavePassag(int i, int lot): Aeronave(i){lotacao=lot;}
};

class AeronaveCarga: public Aeronave{
	int tara;
public:
	AeronaveCarga(int i, int tr): Aeronave(i){tara=tr;}
};
```

c)

```
bool Aeroporto::addGate(int n){
      Gate g(n);
      return gates.insert(g);
}
bool Aeroporto::addAeronavePassag(int i, int lot){
      Aeronave* pa= new AeronavePassag(i, lot);
      return aeronaves.insert(pa);
}
bool Aeroporto::addAeronaveCarga(int i, int tr){
      Aeronave* pa= new AeronaveCarga(i, tr);
      return aeronaves.insert(pa);
}
```

d)

```
bool conectarGateAeronave(int ng, int na, bool en){
      Gate *g=findGate(ng);
      if(g!=NULL && g->getAeronave()==NULL){
            Aeronave *a=findAeronave(na);
            if(a!=NULL && a->getGate()==NULL){
                  g->setAeronave(a);
                  g->setEmbarque(en);
                  a->setGate(g);
                  return true;
            }else return false;
      }else return false;
}

Gate *Aeroporto::findGate(int i){
      Gate g(i);
      return gates.find(g);
}
Aeronave *Aeroporto::findAeronave(int i){
      Aeronave a(i);
      return aeronaves.find(&a);
}

void Gate::setEmbarque(bool en){ embarque=en;}

Aeronave *Gate::getAeronave(){ return aeronave;}
void Gate::setAeronave(Aeronave *a){ aeronave=a;}

Gate *Aeronave::getGate(){ return gate;}
void Aeronave::setGate(Gate *g){ gate=g;}
```

GRUPO III

a)

```
bool Aeroporto::gateConectada(int ng){
      Gate *g=findGate(ng);
      if(g!=NULL) return g->conectada();
      else return false;
}

bool Gate::conectada()const{return aeronave!=NULL;}
```

b)

```
bool Aeroporto::desconectar(int ng){
      Gate *g=findGate(ng);
      if(g!=NULL && g->conectada()){
            g->getAeronave()->setGate(NULL);
            g->setAeronave(NULL);
            return true;
      }else return false;
}
```

c)

```
void Aeroporto::remGate(int ng){
      if(gateConectada(ng)) desconectar(ng);
      Gate g(ng);
      gates.erase(g);
}
```

d)

```
void Aeroporto::listarAeronaves(){
      cout<<"Aeronaves:"<<endl;
      ColeccaoHibrida<Aeronave*>::iterator it;
      for(it=aeronaves.begin(); it!=aeronaves.end(); it++)
            (*it)->print();
}

virtual void Aeronave::print()const{
      cout<<"Num "<<id;
      if(gate!=NULL) cout<<" conectada";
}

void AeronavePassag::print()const{
      Aeronave::print();
      cout<<": Aeronave de passageiros com lotacao "<<lotacao<<endl;
}

void AeronaveCarga::print()const{
      Aeronave::print();
      cout<<": Aeronave de carga de tara "<<tara<<endl;
}
```

e)

```
void Aeroporto::listarGatesConectadas(){
	cout<<"Gates conectadas:"<<endl;
	Coleccao<Gate>::iterator it;
	for(it=gates.begin(); it!=gates.end(); it++)
		if(it->conectada()) it->print();
}

void Gate::print()const{
	cout<<"Gate "<<num;
	if(!conectada()) cout<<" desconectada"<<endl;
	else{
		cout<<(embarque?" Embarque":" Desembarque")<<" da aeronave :"<<endl<<"\t";
		aeronave->print();
	}
};
```

f)

```
	Aeroporto::~Aeroporto(){
		ColeccaoHibrida<Aeronave*>::iterator it;
		for(it=aeronaves.begin(); it!=aeronaves.end(); it++)
			delete *it;
	}
```

g)

```
void main(){
	Aeroporto ap;
	ap.addAeronavePassag(1, 100);
	ap.addAeronaveCarga(2, 25000);
	ap.addGate(5);
	ap.addGate(7);
	ap.conectarGateAeronave(5,1,true);
	ap.conectarGateAeronave(7,2,false);
	ap.listarAeronaves();
	ap.listarGatesConectadas();
	ap.desconectar(7);
	ap.remGate(7);
}
```