Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM

Ciência da Computação - Bacharelado

Trabalho Individual

Engenharia de Software I

Victor Emmanuel Susko Guimarães

Ouro Preto
Dezembro de 2024

Conteúdo

I Resumo

Este trabalho constitui um documento do projeto de uma API para o trabalho prático individual da disciplina de Engenharia de Software I. Consta aqui o projeto das funcionalidades (casos de uso) da API, dos testes em pseudocódigo, bem como um diagrama UML representativo das classes do software, que posteriormente será implementado em código.

O objetivo do projeto é construir uma API que consiga realizar simulações de eventos em sistemas ao longo do tempo, para que seja possível prever diferentes cenários condicionados a uma situação inicial do sistema, podendo este ou não sofrer interferências de fatores externos.

Dessa forma, para este projeto, existem 3 principais componentes da API: o sistema, o fluxo e o modelo.

- O **Sistema** constitui um agente que está sujeito a mudanças ao longo do tempo. Ele pode representar seres vivos, conjuntos, locais, cidades, ou qualquer entidade que possa sofrer algum tipo de alteração de estado. Ex: o sistema poderia ser um conjunto de galinhas, que começa com uma população inicial e pode sofrer ação de predadores, alimentos disponíveis, espaço, etc. É possível que um sistema contenha outros sistemas.

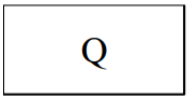
- O **Fluxo** é o componente que promove uma alteração no estado de um sistema, e o usuário deve definir, em cada fluxo, qual a sua expressão matemática associada. Como o fluxo possui um sentido, ou seja, de onde os recursos vêm, e para onde vão, a expressão matemática em conjunto com o sentido do fluxo definirão se um sistema perde ou ganha recursos ao longo do tempo. Vale ressaltar que é possível que um fluxo não esteja necessariamente associado a um sistema, ou seja, a API deve permitir que a simulação aconteça mesmo que não haja sentido lógico de existir um fluxo sozinho.

- O **Modelo** representa um conjunto de sistemas e de fluxos, cabendo a ele realizar o gerenciamento de uma aplicação inteira. Por isso, ele possui o controle do tempo decorrido durante a simulação, do lapso temporal, das ordens de execução, bem como adicionar ou remover fluxos e sistemas.

II Casos de uso

Definirão-se, aqui, quais são os possíveis usos da API, ou seja, quais as possíveis interações entre os componentes listados. São eles:

Caso 1: Sistema isolado



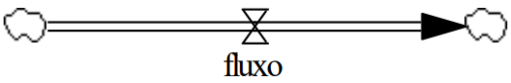
Representação em código com construtor:

```
//creating model and system
Model model = Model("model");
System* q   = new System("Q", 100);

//adding the system to the model
model.add(q);
```

Exemplo alternativo com uso de setters:

```
//creating model and system
Model model = Model("model");
System* alt = new System();
alt->setName("alternative");
alt->setValue(50);

//adding alternative system
model.add(alt);
```

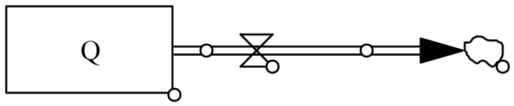
Caso 2: Fluxo isolado



Representação em código:

```
//defining an expression for flow
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return 1;}
};

//creating model and system
Model model    = Model("model");
Expression* f  = new Expression("flow");

//adding flow to the model
model.add(f);
```

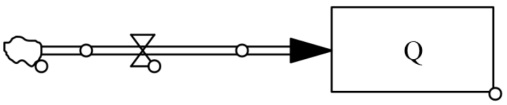
Caso 3: Fluxo com origem em um sistema, mas sem destino



Representação em código:

```
//defining an expression for flow
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return 0;}
};

//creating model, flow and system
Model model     = Model("model");
Expression* f   = new Expression("flow");
System* q       = new System("Q", 100);

//adding flow and system to the model
model.add(q);
model.add(f);

//setting flow source
f.setSource(q);
```

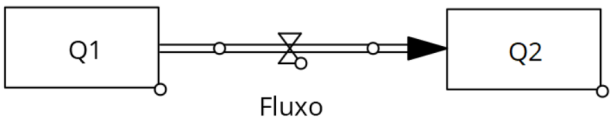
Caso 4: Fluxo com destino em um sistema, mas sem origem



Representação em código:

```
//defining an expression for flow
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return (10* target->getValue());}
};

//creating model, flow and system
Model model     = Model("model");
Expression f    = new Expression("flow");
System q        = new System("Q", 100);

//adding flow and system to the model
model.add(q);
model.add(f);

//setting flow target
f.setTarget(q);
```

Caso 5: Fluxo conectando sistemas



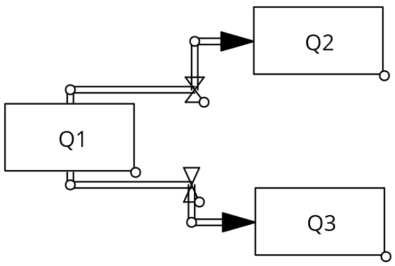
Representação em código:

```
//defining an expression for flow
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return (1.05 * source->getValue());}
};

//creating model, flow and systems
Model model  = Model("model");
System q1    = new System("Q1", 100);
System q2    = new System("Q2", 0);
Expression f = new Expression("flow", q1, q2);

//adding flow and both systems
model.add(q1);
model.add(q2);
model.add(f);

//simulate function
model.simulate(0,99,1);
```

Exemplo alternativo com uso de setters:

```
//creating model, flow and systems
Model model  = Model("model");
System q1    = new System("Q1", 100);
System q2    = new System("Q2", 0);

Expression* alt = new Expression();
alt->setName("alternative");
alt->setSource(q1);
alt->setTarget(q2);

//adding flow and both systems
model.add(q1);
model.add(q2);
model.add(alt);

//simulate
model.simulate(0,99,1);
```

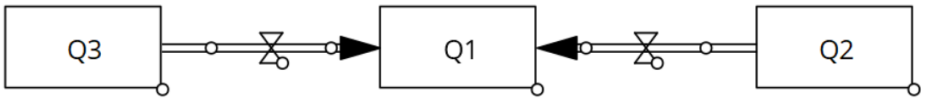
Caso 6: Múltiplos fluxos saindo de um sistema



Representação em código:

```
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return source->getValue();}
};

//creating model, flows and systems
Model model   = Model("model");
System q1     = new System("Q1", 200);
System q2     = new System("Q2", 0);
System q3     = new System("Q3", 0);
Expression f1 = new Expression("f1",q1,q2);
Expression f2 = new Expression("f2",q1,q3);

//adding existing elements to the model
model.add(f1);
model.add(f2);
model.add(q1);
model.add(q2);
model.add(q3);

//simulate
model.simulate(0,99,1);
```

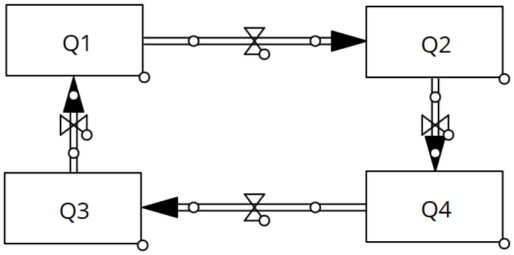
Caso 7: Múltiplos fluxos entrando em um sistema



Representação em código:

```
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return 0;}
};

//creating model, flows and systems
Model model   = Model("model");
System q1     = new System("Q1", 0);
System q2     = new System("Q2", 100);
System q3     = new System("Q3", 100);
Expression f1 = new Expression("f1",q3,q1);
Expression f2 = new Expression("f2",q2,q1);

//adding existing elements to the model
model.add(f1);
model.add(f2);
model.add(q1);
model.add(q2);
model.add(q3);

//simulate
model.simulate(0,99,1);
```

Caso 8: Sistemas e fluxos encadeados em um ciclo



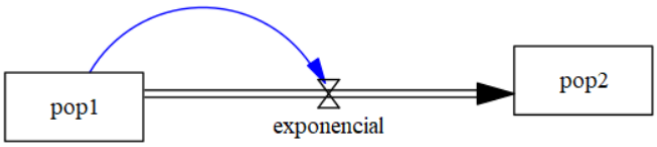
Representação em código:

```
class Expression : public Flow {
    public:
        Expression(){}
        Expression(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return 0;}
};

//creating model, flow and system
Model model   = Model("model");
System q1     = new System("Q1", 0);
System q2     = new System("Q2", 100);
System q3     = new System("Q3", 0);
System q4     = new System("Q4", 100);
Expression f1 = new Expression("f1",q1,q2);
Expression f2 = new Expression("f2",q2,q4);
Expression f3 = new Expression("f3",q4,q3);
Expression f4 = new Expression("f4",q3,q1);

//adding flow and system to the model
model.add(f1);model.add(f2);
model.add(f3);model.add(f4);
model.add(q1);model.add(q2);
model.add(q3);model.add(q4);

//simulate
model.simulate(0,99,1);
```

III Critérios de Aceitação

Para validação do projeto, o cliente requisita que 3 situações possam ser representadas computacionalmente e consigam reproduzir os mesmos valores nos seguintes casos:

Exponencial:



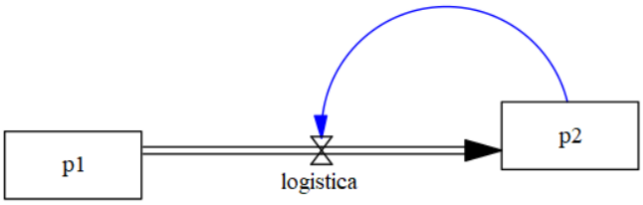
A população 1 começa em 100, a população 2 começa de 0 e o fluxo "exponencial"dita que a seguinte expressão:

$$exponencial = 0,01 \times pop1$$

Representação em código:

```
class Exponential : public Flow {
    public:
        Exponential(){}
        Exponential(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return (source->getValue() * 0.01);}
};

System* pop1 = new System("pop1", 100);
System* pop2 = new System("pop2", 0);

Exponential* flow = new Exponential("exp", pop1, pop2);

Model model = Model("model");

model.add(pop1);
model.add(pop2);
model.add(flow);

model.simulate(0,99,1);

assert(fabs(pop1->getValue() - 36.6032) < 0.0001);
assert(fabs(pop2->getValue() - 63.3968) < 0.0001);

delete pop1;
delete pop2;
delete flow;
```

Logístico:



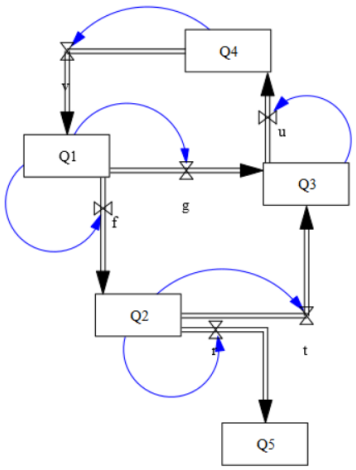
A população 1 começa em 100, a população 2 começa em 10 e o fluxo "logístico"dita a seguinte expressão:

$$logistica = 0,01 \times pop2 \times (1 - \frac{pop2}{70})$$

Representação em código:

```
class Logistical : public Flow {
    public:
        Logistical(){}
        Logistical(string _name, System* _source, System* _target) : Flow(
    _name,_source,_target){}
        virtual double execute(){return (target->getValue() * 0.01 * (1 -
    target->getValue() / 70));}
};

System* p1 = new System("p1", 100);
System* p2 = new System("p2", 10);

Logistical* flow = new Logistical("exp", p1, p2);

Model model = Model("model");

model.add(p1);
model.add(p2);
model.add(flow);

model.simulate(0,99,1);

assert(fabs(p1->getValue() - 88.2167) < 0.0001);
assert(fabs(p2->getValue() - 21.7833) < 0.0001);

delete p1;
delete p2;
delete flow;
```

Complexo:



Este seria o caso que envolve mais sistemas e fluxos, sendo os valores iniciais dos sistemas Q1, Q2, Q3, Q4 e Q5 respectivamente: 100, 0, 100, 0 e 0, e as expressões dos fluxos ”f”, ”g”, ”r”, ”t”, ”u”e ”v”estão definidas como:

$$f = 0,01 \times Q1$$
$$g = 0,01 \times Q1$$
$$r = 0,01 \times Q2$$
$$t = 0,01 \times Q2$$
$$u = 0,01 \times Q3$$
$$v = 0,01 \times Q4$$

Representação em código:

```
class Complex : public Flow {
    public:
        Complex(){}
        Complex(string _name, System* _source, System* _target) : Flow(_name,
    _source,_target){}
        virtual double execute(){return (source->getValue() * 0.01);}
};

System* q1 = new System("q1", 100);
System* q2 = new System("q2", 0);
System* q3 = new System("q3", 100);
System* q4 = new System("q4", 0);
System* q5 = new System("q5", 0);

Complex* flowG = new Complex("g", q1, q3);
Complex* flowF = new Complex("f", q1, q2);
Complex* flowR = new Complex("r", q2, q5);
Complex* flowT = new Complex("t", q2, q3);
Complex* flowU = new Complex("u", q3, q4);
Complex* flowV = new Complex("v", q4, q1);

Model model = Model("model");
```

```
model.add(q1); model.add(q2); model.add(q3);
model.add(q4); model.add(q5);
model.add(flowG); model.add(flowF); model.add(flowR);
model.add(flowT); model.add(flowU); model.add(flowV);

model.simulate(0,99,1);

assert(fabs(q1->getValue() - 31.8513) < 0.0001);
assert(fabs(q2->getValue() - 18.4003) < 0.0001);
assert(fabs(q3->getValue() - 77.1143) < 0.0001);
assert(fabs(q4->getValue() - 56.1728) < 0.0001);
assert(fabs(q5->getValue() - 16.4612) < 0.0001);

delete q1; delete q2; delete q3;
delete q4; delete q5;
delete flowG; delete flowF; delete flowR;
delete flowT; delete flowU; delete flowV;
```

IV Diagrama UML