PRIMEIRA SPRINT
ENGENHARIA DE SOFTWARE I

IGOR MACHADO CRUZ GUIMARÃES OLIVEIRA
21.1.4012

OURO PRETO - MG
2023

1. Introdução

O desenvolvimento da primeira sprint relativa ao trabalho proposto (trabalho individual) consiste na criação de uma API que consiga simular uma linguagem capaz de modelar e trabalhar com sistemas. Os principais requisitos desta API são:

- A capacidade de modelar sistemas de diferentes áreas do conhecimento
- Estabelecer relação de fluxos entre sistemas
- A API deve ser funcional e satisfazer todos os critérios de aceitação apresentados pelo cliente (professor)
- Simular todos os sistemas apresentados

Em primeiro lugar serão apresentados alguns casos de uso que a API deverá satisfazer, trata-se de casos genéricos que seguem os preceitos da linguagem DYNAMO. Além disso, também serão apresentados os critérios de aceitação do cliente, ou seja, a API deve ser capaz de criar e simular os modelos desejados pelo cliente.

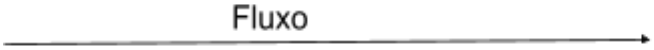
Segundamente será necessário estudar e pensar em algumas alternativas de uso dos casos apresentados anteriormente, para que assim seja possível ter a aceitação do cliente e logo após o desenvolvimento do pseudo-código relativo a esses casos.

Por fim, deve-se criar uma notação de classes (UML), para que seja especificado a API, lembrando que o desenvolvimento da mesma será feito em linguagem C++.

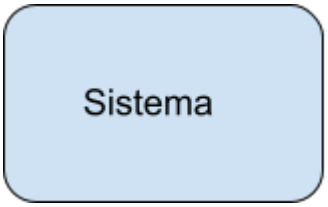
2. Casos de Uso e Critérios de aceitação

Segue abaixo os possíveis casos de uso da API que será desenvolvida:

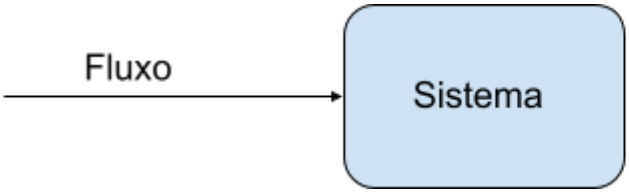
- **Apenas o fluxo, sem origem e sem destino**



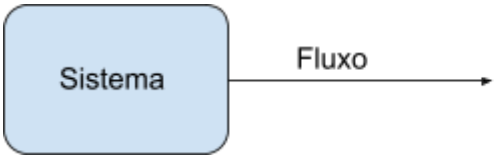
- **Sistema isolado, sem fluxo**



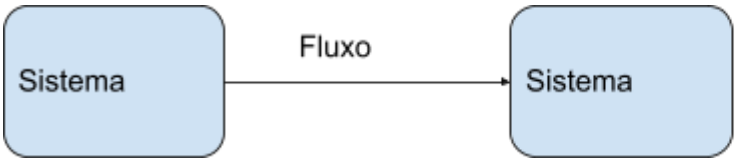
- **Um fluxo com apenas um sistema como destino**



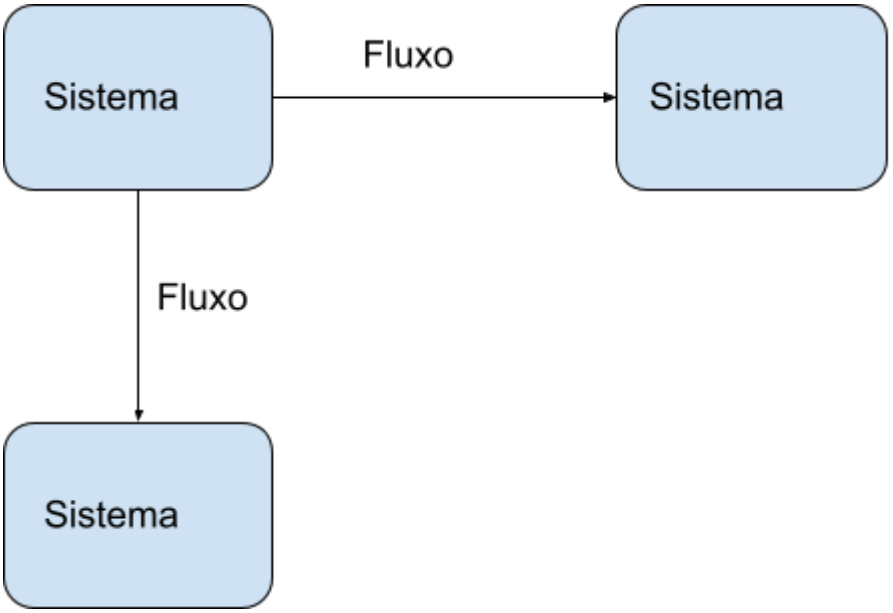
- **Um fluxo com apenas um sistema na origem**



- **Um fluxo com um sistema na origem e no destino**



- **Um sistema ligado a dois ou mais sistemas**



Como é possível perceber, os modelos que poderão ser construídos a partir da API são criados a partir de sistemas e fluxos. Além desses modelos genéricos, o

cliente disponibilizou os seguintes modelos de aceitação para a API, assim a mesma deve ser pensada e desenvolvida para satisfazer os mesmos.

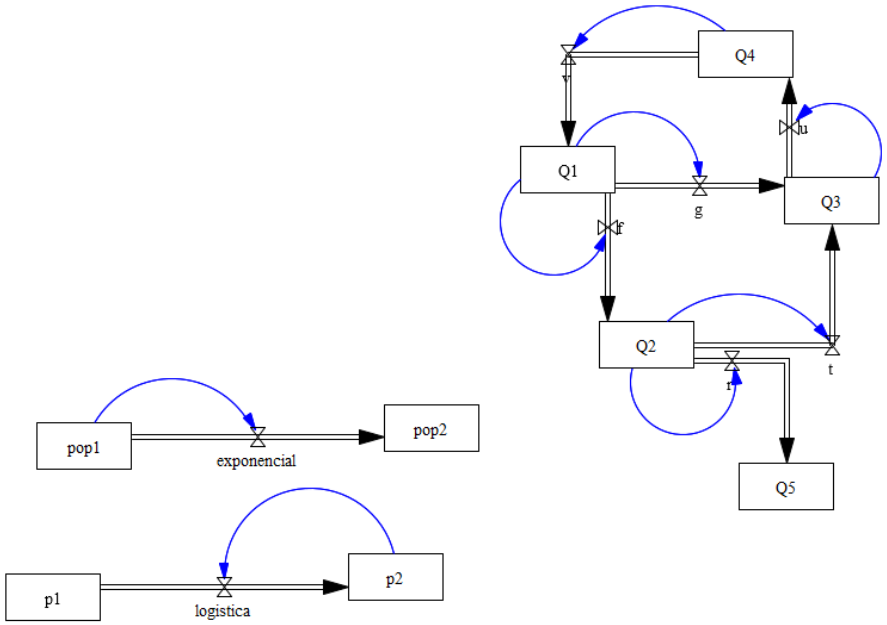


Figura 1 : Modelos de aceitação desejados pelo cliente

3. Estudos dos casos apresentados

Analisando os casos de usos utilizados como exemplo, se inicia o processo de concepção da API, onde serão idealizados algumas possíveis implementações para cada caso, a partir disso, a depender da situação e das necessidades que a aplicação deve satisfazer algumas mudanças podem ser necessárias.

Em primeiro lugar partimos da ideia de que serão trabalhadas três classes : Sistemas, Modelos e Fluxos. Um Modelo será modelado utilizando tanto de Fluxos quanto de Sistemas. Já os Sistemas irão armazenar um valor numérico que será alterado pelos fluxos dependendo da maneira em que o Modelo foi implementado. Por fim o Fluxo, irá ligar de um a dois sistemas, alterando o valor numérico do mesmo através de equações.

Vamos supor um exemplo, caso um modelo que se deseje criar simule uma conta-corrente de um banco. Pensando minimamente, deverá existir um Sistema que armazena o saldo desta conta, já as operações de saque ou depósito funcionam como fluxo.

- **Apenas o fluxo, sem origem e sem destino**

```
Flow f1;
```

- **Sistema isolado, sem fluxo**

```
System s1;
```

- **Um fluxo com apenas um sistema como destino**

```
System s1;
Flow f1;
f1.destiny(s1);
```

- **Um fluxo com apenas um sistema na origem**

```
System s1;
Flow f1;
f1.origin(s1);
```

- **Um fluxo com um sistema na origem e no destino**

```
System s1;
System s2;
Flow f1;
f1.origin(s1);
f1.destiny(s2);
```

- **Um sistema ligado a dois ou mais sistemas**

```
System s1;
System s2;
System s3;
Flow f1;
```

```
Flow f2;
f1.origin(s1);
f1.destiny(s2);
f2.origin(s1);
f2.destiny(s3);
```

Como foi possível perceber acima, os pseudo-códigos relativos aos casos de uso apresentados são simples e sem nenhuma elaboração muito complexa. Entretanto, se baseando nos modelos de validação, é possível se perceber alguns problemas nas implementações listadas. A principal delas é o fato de que cada fluxo terá sua respectiva equação e tomando como base o código acima, se torna impossível realizar essas mudanças.

Considerando esse problema, pensei em uma seguinte solução: a classe Fluxo poderá se tornar uma classe abstrata, pois dentro da mesma existirá um método virtual e abstrato com o nome por exemplo executar, que irá aplicar a equação relativa ao fluxo nos sistemas que o mesmo interage. Assim, pelo fato da classe se tornar abstrata, será impossível instanciar um objeto da classe fluxo, o cliente deverá criar uma classe que herda de fluxo e assim implementar o método executa de acordo com seu desejo.

Outro ponto importante de se trabalhar na implementação da API é a utilização de ponteiros para que cópias das classes que estão sendo trabalhadas sejam evitadas, assim economizando espaço de memória da aplicação a tornando mais eficiente.

- **Estudo modelo exponencial**

```
class ExponentialFlow : public Flow{
    public:
          ExponentialFlow(System& origin, System& destiny) :
Flow(origin, destiny){}
        double execute(){
            return 0.01 * getOrigin();
        }
};

//Para o caso exponencial, uma classe com o mesmo nome pode
ser criada, herdando os atributos e
```

```
//os métodos presentes na classe Fluxo, com a diferença de que ela implementa a equação do fluxo

int main(){
    System pop1(100);
    System pop2(0);

    //A classe ExponencialFlow implementa o método executar com a seguinte fórmula : 0.01 * pop1
    ExponentialFlow f1;
    f1.setOrigin(&pop1);
    f1.setDestiny(&pop2);

    //Criando o modelo
    Model model1;
    model1.add(&pop1);
    model1.add(&pop2);
    model1.add(&f1);

    //Executando (o metodo run executa o metodo execute do flow)
    model1.run(0, 100, 1);
    //ele executa do tempo inicial 0 até o tempo 100 somando de 1 em 1
    //Por fim seria feita a impressão dos dados
}
```

- **Estudo modelo logístico**

```
class LogisticFlow : public Flow{
    public:
            LogisticFlow(System& origin, System& destiny) : Flow(origin, destiny){}
        double execute(){
             return 0.01 * getOrigin() * (1 - getOrigin() / 70);
        }
```

```
};

//Para o caso logistico, uma classe com o mesmo nome pode ser criada, herdando os atributos e
//os métodos presentes na classe Fluxo, com a diferença de que ela implementa a equação do fluxo

int main(){
    System pop1(100);
    System pop2(10);

    //A classe LogisticFlow implementa o método executar com a seguinte fórmula : 0.01 * pop1
    LogisticFlow f1;
    f1.setOrigin(&pop1);
    f1.setDestiny(&pop2);

    //Criando o modelo
    Model model1;
    model1.add(&pop1);
    model1.add(&pop2);
    model1.add(&f1);

     //Executando (o metodo run executa o metodo execute do flow)
    model1.run(0, 100, 1);
    //ele executa do tempo inicial 0 até o tempo 100 somando de 1 em 1
    //Por fim seria feita a impressão dos dados
}
```

- **Estudo do terceiro modelo apresentado**

```
class ModelFlow : public Flow{
    public:
                ModelFlow(System& origin, System& destiny) : Flow(origin, destiny){}
```

```
        double execute(){
            return 0.01 * getOrigin();
        }
};

int main(){
    System q1(100);
    System q2(0);
    System q3(100);
    System q4(0);
    System q5(0);

    ModelFlow f;
    ModelFlow g;
    ModelFlow r;
    ModelFlow t;
    ModelFlow u;
    ModelFlow v;

    f.setOrigin(&q1);
    f.setDestiny(&q2);
    g.setOrigin(&q1);
    g.setDestiny(&q3);
    r.setOrigin(&q2);
    r.setDestiny(&q5);
    t.setOrigin(&q2);
    t.setDestiny(&q3);
    u.setOrigin(&q3);
    u.setDestiny(&q4);
    v.setOrigin(&q4);
    v.setDestiny(&q1);

    Model model1;
    model1.add(&q1);
    model1.add(&q2);
    model1.add(&q3);
```

```
    model1.add(&q4);
    model1.add(&q5);

    model1.add(&f);
    model1.add(&g);
    model1.add(&r);
    model1.add(&t);
    model1.add(&u);
    model1.add(&v);

    model1.run(0, 100, 1);
}
```

- **UML**