

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

# **Sprint1 - PROJETO DA API do simulador de Sistemas Dinâmicos**

BCC322 - Engenharia de Software I

Nicolas E. L. Mendes  
Professor: Tiago Garcia de Senna Carneiro

Ouro Preto  
Outubro de 2023

## **Panorama geral dos requisitos do cliente**

Para reconhecer os casos de uso e os critérios de avaliação do cliente, tomei, conforme orientação presente no pdf fornecido, as informações presentes no arquivo ValidacaoMyVENSIM.mdl. Nesse arquivo estão presentes 3 modelos (casos de aceitação), em diferentes configurações, o que implica que a API a ser desenvolvida deve ser capaz de simulá-los e obter os mesmos resultados apresentados pelo software Vensim, constituindo, assim, os critérios de aceitação do cliente.

Sequencialmente, a fim de construir uma abordagem que parte de representações mais simples para as mais complexas, retirarei os casos de uso da API que estão presentes nos modelos fornecidos pelo arquivo.

Seguidamente, nesse processo de concepção em alto nível, a priori, creio que a representação da estrutura da API mais enxuta seria constituída de três classes: Model, System e Flow.

Com o fito de dar dinamicidade ao processo de concepção, nesse primeiro momento, sem abordar nenhum caso de uso, infiro momentaneamente os seguintes componentes das classes (posteriormente, conforme surgem necessidades e demandas para o uso da API esses componentes serão alterados):

### **Model**

Método run(), para execução do modelo a ser simulado.

Método add(), para adição de System e Flow, atributos da mesma.

Métodos CRUD, para manipulação dos System e Flow associados.

### **System**

Atributo value, para armazenar o estado quantitativo do mesmo;  
getters e setters

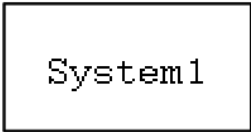
### **Flow**

-

## **Casos de uso**

Inicialmente, com o propósito de dar início ao processo de moldagem da API, os casos de uso serão mapeados e serão apresentados em conjunto com o pseudo-códigos correspondentes. Ademais, algumas discussões e observações serão estabelecidas acerca de múltiplas abordagens para tratativa de diversas possíveis soluções para uma representação ou funcionalidade específica. Desse modo, busca-se que o processo de análise e construção do projeto torne-se mais explícito.

### **1. Caso de uso “Sistema sozinho”**

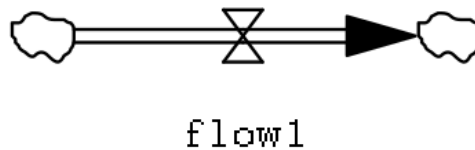


System1

```
Model m;  
System system1;  
m.add(system1);  
m.run(t_inicial, t_final);
```

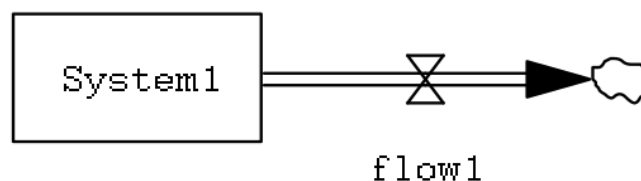
**Observação 1 (Número de iterações atrelado ao tempo):** Conforme presente no PDF orientador, o número de iterações é calculado conforme os tempos iniciais e finais passados. Inicialmente, pensei na criação de uma classe Tempo, cujos tempos final e inicial seriam atributos da mesma, entretanto, no que refere-se a métodos, a classe não teria nenhum além de getters e setters, o que implica que a manutenção da implementação da mesma não é viável, dada sua relevância. Portanto, a fim de simplificar e tornar o software mais enxuto e compacto possível, o tempo inicial e final serão parâmetros da função run() do Modelo.

## 2. Caso de uso “Fluxo sozinho”



```
Model m;  
Flow flow1;  
m.add(flow1);  
m.run(t_inicial, t_final);
```

## 3. Caso de uso “Fluxo somente com origem”



```
Model m;  
System system1;  
Flow flow1;  
flow1.setSource(system1);  
m.add(flow1);
```

```
m.add(system1);  
m.run(t_inicial, t_final);
```

**Observação 2 (Conectar fluxos a sistemas):** Como solução para efetuar a conexão entre fluxos e sistemas, há múltiplas abordagens, dentre elas:

- I. Função connect, pertencente à classe flow, que recebe dois argumentos, origem e destino (respectivamente)

```
...  
System s1, s2;  
Flow f1;  
f1.connect(s1, s2);
```

Possível empecilho: No caso em que o fluxo possui somente origem ou somente destino, a expressão dos argumentos não ficaria a mais legível e expressiva, pois, por algo diferente de um sistema deveria ser passado como argumento, ficando algo do tipo:

```
f1.connect(null, s1) //f1 só possui s1 como destino
```

- II. Função connect, pertencente à classe system, que recebe dois argumentos, origem e destino (respectivamente)

```
...  
System s1;  
Flow f1, f2;  
s1.connect(f1,f2);
```

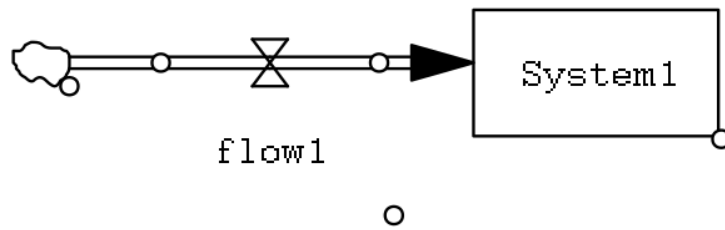
Possível empecilho: No caso em que o sistema possui múltiplos fluxos, seja em sua entrada ou saída, essa função não seria útil, pois, da maneira como está proposta, não seria capaz de comportar mais de um fluxo na entrada de um sistema, por exemplo.

- III. Funções setSource e setTarget, pertencente à classe flow, onde a primeira recebe um sistema, representando a origem, e a segunda recebe, também, um sistema, representando o destino

```
...  
System s1, s2;  
Flow f1;  
f1.setSource(s1);  
f1.setTarget(s2);
```

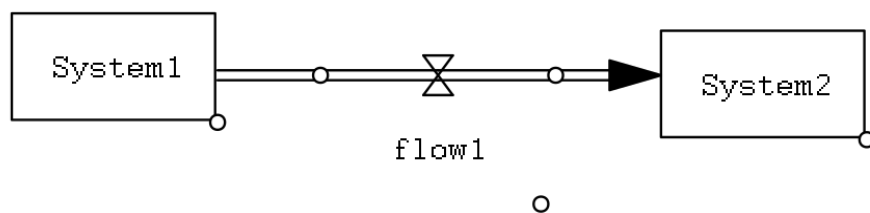
Dentre as soluções propostas, creio que essa seja, aparentemente, a mais viável, pois encobre todas as problemáticas apresentadas nas soluções anteriores, associando essa função a classe fluxo e dividindo em duas atribuições, origem e destino.

4. Caso de uso “Fluxo somente com destino”



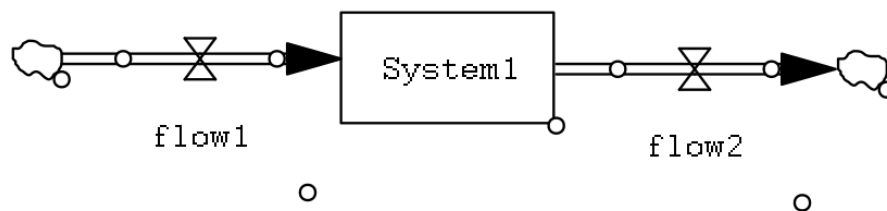
```
Model m;
System system1;
Flow flow1;
flow1.setTarget(system1);
m.add(system1);
m.add(flow1);
m.run(t_inicial, t_final);
```

5. Caso de uso “Fluxo com origem e destino”



```
Model m;
System system1, system2;
Flow flow1;
flow1.setSource(system1);
flow1.setTarget(system2);
m.add(system1);
m.add(system2);
m.add(flow1);
m.run(t_inicial, t_final);
```

6. Caso de uso “Sistema com fluxo na entrada e fluxo na saída”



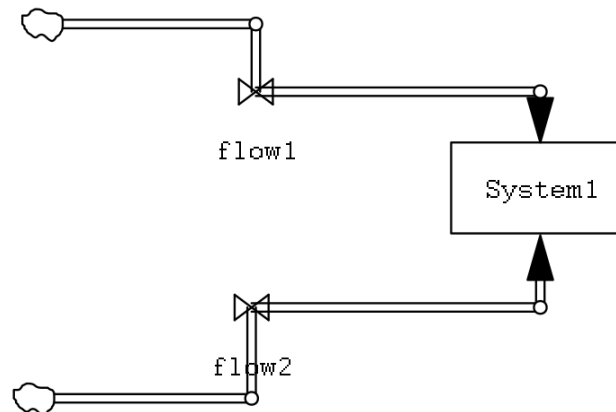
```
Model m;
System system1;
Flow flow1, flow2;
```

```

flow1.setTarget(system1);
flow2.setSource(system2);
m.add(system1);
m.add(flow1);
m.add(flow2);
m.run(t_inicial, t_final);

```

#### 7. Caso de uso “Sistema com vários fluxos de entrada”

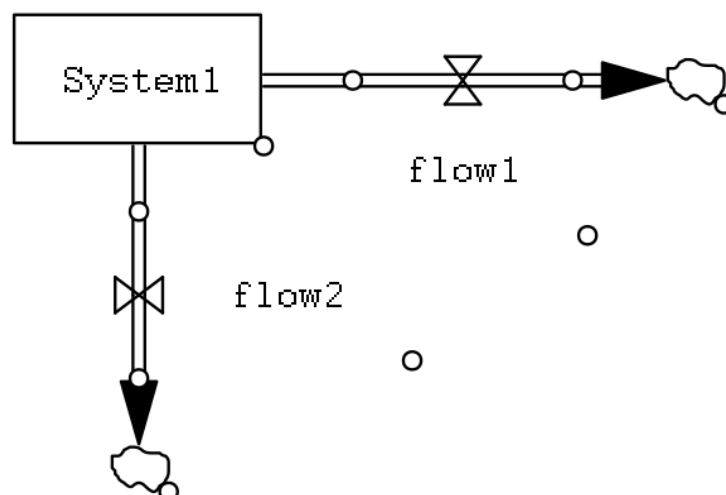


```

Model m;
System system1;
Flow flow1, flow2;
flow1.setTarget(system1);
flow2.setTarget(system1);
m.add(system1);
m.add(flow1);
m.add(flow2);
m.run(t_inicial, t_final);

```

#### 8. Caso de uso “Sistema com vários fluxos de saída”

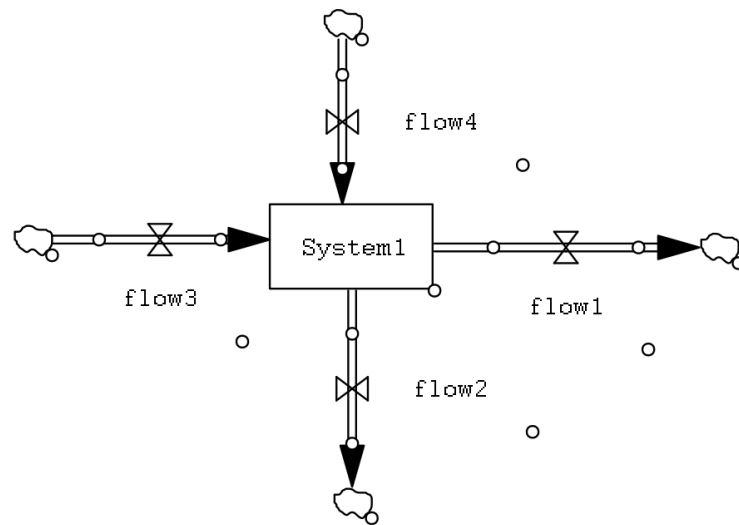


```

Model m;
System system1;
Flow flow1, flow2;
flow1.setSource(system1);
flow2.setSource(system1);
m.add(system1);
m.add(flow1);
m.add(flow2);
m.run(t_inicial, t_final);

```

9. Caso de uso “Sistema com múltiplos fluxos de entrada e de saída”

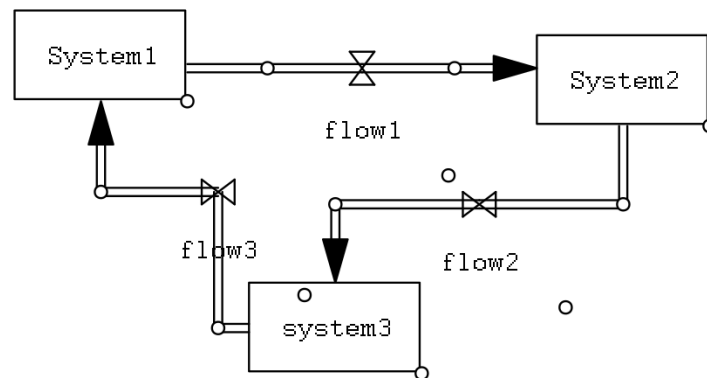


```

Model m;
System system1;
Flow flow1, flow2, flow3, flow4;
flow1.setSource(system1);
flow2.setSource(system1);
flow3.setTarget(system1);
flow4.setTarget(system1);
m.add(system1);
m.add(flow1);
m.add(flow2);
m.add(flow3);
m.add(flow3);
m.run(t_inicial, t_final);

```

10. Caso de uso “Múltiplos sistemas conectados por múltiplos fluxos”



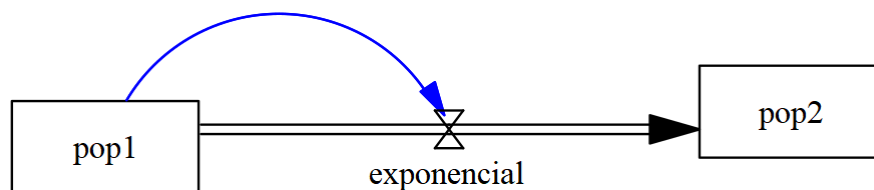
```

Model m;
System system1, system2, system3;
Flow flow1, flow2, flow3;
flow1.setSource(system1);
flow1.setTarget(system2);
flow2.setSource(system2);
flow2.setTarget(system3);
flow3.setSource(system3);
flow3.setTarget(system1);
m.add(system1);
m.add(system2);
m.add(system3);
m.add(flow1);
m.add(flow2);
m.add(flow3);
m.run(t_inicial, t_final);

```

### Critérios de aceitação

1. Caso de aceitação “Múltiplos sistemas conectados por um fluxo regido por uma equação”



```

Class Exponencial : public Flow{
public:
    double equation()
    {
        return 0.4 * x * pi() / 4.0;
    }
};

```



```

Model m;
System pop1, pop2;
Exponencial flow1;
flow1.setSource(pop1);
flow1.setTarget(pop2);
flow1.equation(exponencial);
m.add(pop1);
m.add(pop2);
m.add(flow1);
m.run(t_inicial, t_final);

```

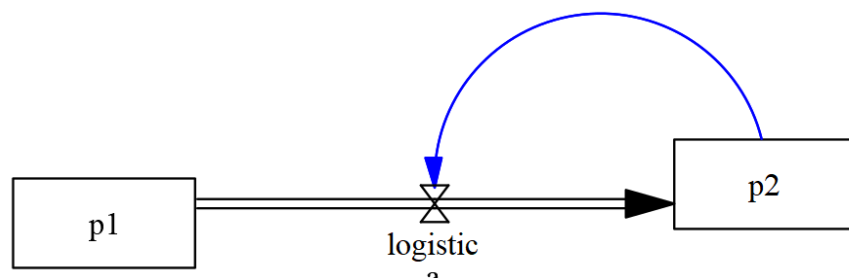
**Observação 3 (Parametrização da equação):** Para representar a equação que rege o fluxo em termos de parâmetros e variáveis têm-se múltiplas possibilidades.

Dentre elas, uma alternativa seria: o programador-usuário expressa a equação que rege o fluxo em uma função e passa um ponteiro para a mesma como argumento de um método da classe Flow, que assume essa função passada como a equação do fluxo (expresso no método equation). Entretanto, utilizando ponteiro de função, haveria uma quebra com um dos componentes da programação orientada a objetos, o polimorfismo. Além disso, essa estratégia não viabiliza controle a respeito de como a função do programador-usuário funcionará, não podendo, a API, definir o tipo de retorno e/ou como as operações são executadas.

Uma outra alternativa, também, constitui-se na passagem da equação por meio de uma String. Entretanto, na prática, essa alternativa demonstra-se inviável, pois, para assegurar a funcionalidade da equação, deve-se fazer uma análise léxica, sintática e semântica do que foi passado, efetuando o trabalho de um compilador na prática, algo desnecessariamente complexo para essa aplicação.

Portanto, uma outra estratégia, que será adotada, consiste em tornar a classe Flow uma classe abstrata, em que a equation será um método abstrato absoluto que o programador-usuário deverá implementar, criando uma nova classe que herde da classe Flow. Desse modo, pode-se usufruir de um dos pilares da programação orientada a objetos, o polimorfismo, além de definir alguns requisitos para a implementação da equação, como o tipo de retorno do respectivo método da equação.

2. Caso de aceitação “Múltiplos sistemas conectados por um fluxo regido por uma equação”

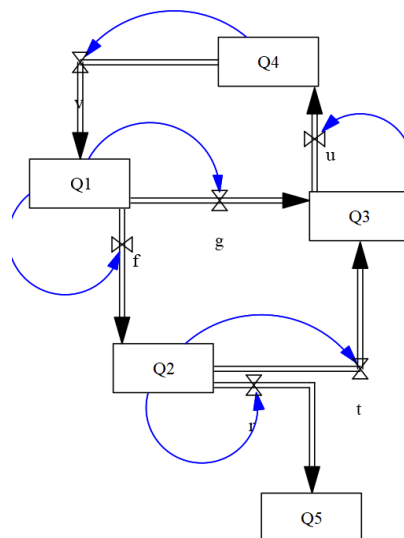


```

Class NovoFluxo : public Flow{
public:
    double equation()
    {
        return 0.3 * x * pi() / 2.0;
    }
};
Model m;
System p1, p2;
NovoFluxo flow1;
flow1.setSource(p1);
flow1.setTarget(p2);
m.add(p1);
m.add(p2);
m.add(flow1);
m.run(t_inicial, t_final);

```

3. Caso de aceitação “Múltiplos sistemas conectados por múltiplos fluxos regidos por múltiplas equações”



```

Class Flow1 : public Flow{
public:
    double equation()
    {
        return 0.8 * x / 5.0;
    }
};

```

```

Class Flow2 : public Flow{
public:
    double equation()
    {
        return 0.5 * x / 5.0;
    }
};

```

```

    }
};

Class Flow3 : public Flow{
public:
    double equation()
    {
        return 0.8 * x / 4.0;
    }
};

```

```

Class Flow4 : public Flow{
public:
    double equation()
    {
        return 0.6 * x / 2.0;
    }
};

```

```

Class Flow5 : public Flow{
public:
    double equation()
    {
        return 0.2 * x / 6.0;
    }
};

```

```

Class Flow6 : public Flow{
public:
    double equation()
    {
        return 0.8 * x / 3.0;
    }
};

```

```

Model m;
System q1, q2, q3, q4, q5;
Flow1 f;
Flow2 g;
Flow3 u;
Flow4 v;
Flow5 t;
Flow 6 r;
f.setSource(q1);
f.setTarget(q2);
g.setSource(q1);
g.setTarget(q3);
u.setSource(q3);

```

```

u.setTarget(q4);
v.setSource(q4);
v.setTarget(q1);
t.setSource(q2);
t.setTarget(q3);
r.setSource(q2);
r.setTarget(q5);
m.add(q1);
m.add(q2);
m.add(q3);
m.add(q4);
m.add(q5);
m.add(f);
m.add(g);
m.add(u);
m.add(t);
m.add(v);
m.add(r);
m.run(t_inicial, t_final);

```

## UML

Após avaliar os casos de uso e casos de aceitação, e modificação das estruturas, atributos e métodos propostos inicialmente, constitui-se a seguinte UML para representação da API.

