

Fábio Martins Gonçalves Ferreira

*Desenvolvimento e Aplicações de um
Framework Orientado a Objetos para
Análise Dinâmica de Linhas de Ancoragem
e de Risers*

Maceió – AL

Dezembro / 2005

Fábio Martins Gonçalves Ferreira

*Desenvolvimento e Aplicações de um
Framework Orientado a Objetos para
Análise Dinâmica de Linhas de Ancoragem
e de Risers*

Dissertação apresentada ao Programa de
Pós-Graduação em Engenharia Civil como
parte dos requisitos para a obtenção do título
de Mestre em Engenharia Civil

Orientador:

Prof. Eduardo Setton Sampaio da Silveira

Co-orientador:

Prof. Eduardo Nobre Lages

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA CIVIL
DEPARTAMENTO DE ENGENHARIA ESTRUTURAL
CENTRO DE TECNOLOGIA
UNIVERSIDADE FEDERAL DE ALAGOAS

Maceió – AL

Dezembro / 2005

Catálogo na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável: Helena Cristina Pimentel do Vale

- F383d Ferreira, Fábio Martins Gonçalves
Desenvolvimento e aplicações de um framework orientado a objetos para análise dinâmica de linhas de ancoragem e de risers / Fábio Martins Gonçalves Ferreira. – Maceió, 2005.
107f. : il.
- Orientador: Eduardo Setton Sampaio da Silveira.
Co-Orientador: Eduardo Nobre Lages.
Dissertação (mestrado em Engenharia Civil : Estruturas) – Universidade Federal de Alagoas. Centro de Tecnologia. Maceió, 2006.
- Bibliografia: f. 97-100.
Apêndices: f. 101-107.
1. Engenharia – Estruturas marítimas. 2. Estruturas – Cabos.
3. Estruturas – Análise dinâmica. 4. Estruturas – Análise Não Linear.
5. Programa computacional – Framework orientado a objetos. I. Título.
- CDU: 624.07

*À minha esposa Aline,
pelo apoio e incentivo.*

Agradecimentos

Aos professores Eduardo Setton Sampaio da Silveira e Eduardo Nobre Lages, pelas orientações, conselhos e incentivos.

Ao Ivan Fábio Mota de Menezes e ao Luiz Cristovão Gomes Coelho, pela sugestões dadas ao trabalho.

Ao Luiz Eugênio Fernandes Tenório (LEFT), pela auxílio na definição do arcabouço, ou melhor, do *framework*.

À minha esposa Aline, pela ajuda na revisão deste trabalho.

À CAPES, pela concessão da bolsa de mestrado.

À todos os colegas do Mestrado em Mecânica Computacional da UFAL.

Resumo

A crescente demanda incentivada pela indústria do petróleo por sistemas computacionais complexos que sejam capazes de simular o comportamento físico de estruturas *offshore* motiva o desenvolvimento deste trabalho. Os sistemas computacionais existentes para análise dinâmica de linhas de ancoragem e de *risers* apresentam algumas limitações no que diz respeito à incorporação de novas implementações, não querendo dizer com isso que a extensão desses códigos não seja possível. Contudo, por diversas vezes a incorporação de novas funcionalidades a esses sistemas demanda, além de um esforço inicial para o entendimento do código, um dispendioso trabalho para implementar as novas funcionalidades. Com o objetivo de aumentar a eficiência no desenvolvimento desses sistemas, apresenta-se neste trabalho a implementação de um *framework* para análise dinâmica geometricamente não linear de linhas de ancoragem e *risers*. Esse *framework* tende a facilitar o entendimento, o reuso, a extensão e a manutenção do código. Ainda são mostrados no decorrer deste trabalho os principais aspectos que norteiam o desenvolvimento do *framework*, como os relacionados ao problema físico modelado pelo *framework*, a conceitos computacionais importantes, à validação do *framework* através de aplicações e às contribuições mais relevantes deste trabalho.

Abstract

This work is motivated by the increasing demand that has been stimulated by the oil industry for complex computational systems which are capable to simulate the physical behavior of offshore structures. Computational systems available for dynamic analysis of mooring lines and risers have some limitations for including new implementations. The extension of these codes is possible, though. However, many times the incorporation of new functionalities to these systems demands hard work to implement new functionalities. Moreover, an initial effort is needed for the understanding of code. A framework for geometrically nonlinear dynamic analysis of mooring lines and risers is built to increase the efficiency in the development of these systems. This framework facilitates understanding, reuses, extension and maintenance of code. In addition, this work shows the main aspects that guides framework development, such as the physical problem modeling, important computational concepts, validation of framework through applications and contributions most important of this work.

Sumário

Lista de Figuras

Lista de Tabelas

| | | |
|----------|--|-------|
| 1 | Introdução | p. 14 |
| 1.1 | Motivação | p. 17 |
| 1.2 | Soluções | p. 18 |
| 1.3 | Revisão Bibliográfica | p. 19 |
| 1.4 | Objetivos | p. 20 |
| 1.5 | Definição de Conceitos | p. 21 |
| 1.5.1 | Orientação a Objetos | p. 21 |
| 1.5.2 | <i>Framework</i> Orientado a Objetos | p. 23 |
| 1.5.2.1 | Benefícios de um <i>framework</i> | p. 24 |
| 1.5.2.2 | Diferença entre um <i>framework</i> e uma biblioteca | p. 24 |
| 1.5.3 | UML | p. 25 |
| 1.6 | Organização do Trabalho | p. 27 |
| 2 | Formulação do Problema | p. 28 |
| 2.1 | Equação de Movimento | p. 28 |
| 2.1.1 | Problemas Lineares | p. 29 |
| 2.1.2 | Problemas Não Lineares | p. 30 |
| 2.2 | Métodos de Integração Direta | p. 31 |
| 2.2.1 | Métodos Explícitos de Integração Direta | p. 31 |

| | | |
|----------|--|--------------|
| 2.2.2 | Métodos Implícitos de Integração Direta | p. 32 |
| 3 | Modelagem do <i>Framework</i> e Descrição de suas Classes | p. 33 |
| 3.1 | Modelagem da Solução | p. 33 |
| 3.2 | Descrição das Classes | p. 37 |
| 3.2.1 | Classe Model | p. 38 |
| 3.2.2 | Classe Soil | p. 40 |
| 3.2.3 | Classe Kelvin | p. 40 |
| 3.2.4 | Classe Current | p. 42 |
| 3.2.5 | Classe Load | p. 44 |
| 3.2.5.1 | Classe Buoy | p. 45 |
| 3.2.5.2 | Classe Clump | p. 46 |
| 3.2.5.3 | Classe Force | p. 48 |
| 3.2.6 | Classe Seabed | p. 49 |
| 3.2.6.1 | Classe Plane | p. 49 |
| 3.2.7 | Classe Element | p. 51 |
| 3.2.8 | Classe Material | p. 53 |
| 3.2.8.1 | Classe Hooke | p. 54 |
| 3.2.8.2 | Classe Zener | p. 55 |
| 3.2.9 | Classe Node | p. 56 |
| 3.2.10 | Classe Support | p. 57 |
| 3.2.11 | Classe Prescribed | p. 58 |
| 3.2.11.1 | Classe Displacement | p. 58 |
| 3.2.11.2 | Classe Velocity | p. 59 |
| 3.2.12 | Classe Function | p. 60 |
| 3.2.12.1 | Classe Piecewise | p. 61 |
| 3.2.12.2 | Classe StepCosine | p. 62 |

| | | |
|----------|--|--------|
| 3.2.12.3 | Classe Harmonic | p. 64 |
| 3.2.13 | Classe IntAlg | p. 65 |
| 3.2.13.1 | Classe ChungLee | p. 67 |
| 3.2.13.2 | Classe MEGAlpha | p. 68 |
| 4 | Exemplo e Aplicações | p. 71 |
| 4.1 | Exemplo: <i>Hello Framework!</i> | p. 73 |
| 4.2 | Aplicação 1: PREA3D | p. 81 |
| 4.3 | Aplicação 2: TPN (Tanque de Provas Numérico) | p. 85 |
| 4.4 | Aplicação 3: DYNASIM | p. 87 |
| 5 | Considerações Finais | p. 93 |
| 5.1 | Principais Contribuições | p. 94 |
| 5.2 | Sugestões para Trabalhos Futuros | p. 95 |
| | Referências Bibliográficas | p. 97 |
| | Apêndice A – Serviços Oferecidos pelo Framework | p. 101 |
| A.1 | Serviços de Construção | p. 101 |
| A.2 | Serviços de Consulta | p. 104 |
| A.3 | Serviço de Simulação | p. 106 |

Lista de Figuras

| | | |
|----|---|-------|
| 1 | Imagens de estruturas que utilizam cabos estruturais. | p. 14 |
| 2 | Estrutura e operação <i>offshore</i> | p. 16 |
| 3 | Exemplo de um diagrama de classes da linguagem UML (Neto, 2005). . . | p. 26 |
| 4 | Ilustração do modelo físico. | p. 34 |
| 5 | Ilustração do modelo computacional. | p. 35 |
| 6 | Diagrama de classes do <i>framework</i> desenvolvido. | p. 36 |
| 7 | Diagrama de classe de <code>Model</code> | p. 39 |
| 8 | Criação de uma instância de <code>Model</code> | p. 39 |
| 9 | Linha-solo. | p. 40 |
| 10 | Diagrama de classe de <code>Soil</code> | p. 40 |
| 11 | Modelagem do solo marinho como uma base viscoelástica. | p. 41 |
| 12 | Diagrama de classe de <code>Kelvin</code> | p. 41 |
| 13 | Criação e associação de uma instância de <code>Kelvin</code> | p. 42 |
| 14 | Diagrama de classe de <code>Current</code> | p. 43 |
| 15 | Ilustração do perfil de corrente. | p. 44 |
| 16 | Criação e associação de uma instância de <code>Current</code> | p. 44 |
| 17 | Diagrama de classe de <code>Load</code> | p. 45 |
| 18 | Diagrama de classe de <code>Buoy</code> | p. 45 |
| 19 | Situações possíveis que a bóia pode se encontrar. | p. 46 |
| 20 | Criação e associação de uma instância de <code>Buoy</code> | p. 46 |
| 21 | Diagrama de classe de <code>Clump</code> | p. 47 |
| 22 | Situações que a poita pode se encontrar. | p. 47 |

| | | |
|----|---|-------|
| 23 | Criação e associação de uma instância de Clump | p. 48 |
| 24 | Diagrama de classe de Force | p. 48 |
| 25 | Criação e associação de uma instância de Force | p. 49 |
| 26 | Diagrama da classe Seabed | p. 49 |
| 27 | Diagrama de classe de Plane | p. 50 |
| 28 | Criação e associação de uma instância de Plane | p. 51 |
| 29 | Modelo de massa concentrada. | p. 52 |
| 30 | Criação e associação de uma instância de Element | p. 53 |
| 31 | Diagrama de classe de Material | p. 53 |
| 32 | Diagrama de classe de Hooke | p. 54 |
| 33 | Criação e associação de uma instância de Hooke | p. 55 |
| 34 | Modelo constitutivo de Zener | p. 55 |
| 35 | Diagrama de classe de Zener | p. 56 |
| 36 | Criação e associação de uma instância de Zener | p. 56 |
| 37 | Diagrama de classe de Node | p. 57 |
| 38 | Criação e associação de uma instância de Node | p. 57 |
| 39 | Diagrama de classe de Support | p. 57 |
| 40 | Criação e associação de uma instância de Support | p. 58 |
| 41 | Diagrama de classe de Prescribed | p. 58 |
| 42 | Diagrama de classe de Displacement | p. 59 |
| 43 | Criação e associação de uma instância de Disp | p. 59 |
| 44 | Diagrama de classe de Velocity | p. 60 |
| 45 | Criação e associação de uma instância de Veloc | p. 60 |
| 46 | Diagrama de classe de Function | p. 61 |
| 47 | Exemplo da função <i>piecewise linear</i> | p. 61 |
| 48 | Diagrama de classe de Piecewise | p. 62 |

| | | |
|----|--|-------|
| 49 | Criação e associação de uma instância de <code>Piecewise</code> | p. 62 |
| 50 | Exemplo da função <i>step cosine</i> | p. 63 |
| 51 | Diagrama de classe de <code>StepCosine</code> | p. 63 |
| 52 | Criação e associação de uma instância de <code>StepCosine</code> | p. 64 |
| 53 | Exemplo da função <i>harmonic</i> | p. 64 |
| 54 | Diagrama de classe de <code>Harmonic</code> | p. 65 |
| 55 | Criação e associação de uma instância de <code>Harmonic</code> | p. 65 |
| 56 | Diagrama de classe de <code>IntAlg</code> | p. 66 |
| 57 | Diagrama de classe de <code>ChungLee</code> | p. 67 |
| 58 | Algoritmo de integração de Chung-Lee para problema não linear (Silveira, 2001). | p. 68 |
| 59 | Criação de uma instância de <code>ChungLee</code> | p. 68 |
| 60 | Diagrama de classe de <code>MEGAlpha</code> | p. 69 |
| 61 | Algoritmo de integração de Hulbert-Chung com dissipação numérica ótima (Silveira, 2001). | p. 70 |
| 62 | Criação de uma instância de <code>MEGAlpha</code> | p. 70 |
| 63 | Interação de um <i>cliente</i> com o <i>framework</i> | p. 72 |
| 64 | Esquema do problema. | p. 73 |
| 65 | Função <i>main</i> da aplicação. | p. 75 |
| 66 | Código da função <i>CriarModelo</i> | p. 75 |
| 67 | Código da função <i>CriarSolo</i> | p. 75 |
| 68 | Código da função <i>CriarPerfilDeCorrente</i> | p. 76 |
| 69 | Código da função <i>CriarFundoDoMar</i> | p. 76 |
| 70 | Código da função <i>CriarNos</i> | p. 77 |
| 71 | Código da função <i>CriarElementos</i> | p. 78 |
| 72 | Código da função <i>CriarCarregamento</i> | p. 79 |
| 73 | Código da função <i>CriarSuportes</i> | p. 80 |

| | | |
|----|--|-------|
| 74 | Código da função <i>CriarAlgoritmoDeIntegracao</i> | p. 80 |
| 75 | Código da função <i>AnalisarModelo</i> | p. 81 |
| 76 | Interface gráfica do PREA3D. | p. 82 |
| 77 | Opção de <i>Dynamic Equilibrium</i> da linha ou <i>riser</i> selecionado. | p. 83 |
| 78 | Janela <i>Dynamic Equilibrium</i> | p. 83 |
| 79 | Pseudocódigo do método <i>run</i> da classe <i>PreadynData</i> | p. 84 |
| 80 | Metodologia da análise acoplada adotada no TPN. | p. 86 |
| 81 | Pseudocódigo do método <i>ComputeLineForce</i> da classe <i>cPreadyn</i> | p. 87 |
| 82 | Interface gráfica do Predyna, exibindo um modelo com plataforma e linhas. | p. 88 |
| 83 | Alguns casos em que não é possível obter a configuração de equilíbrio da linha, com o módulo atual. | p. 89 |
| 84 | Pseudocódigo do controle da simulação através do ângulo no topo. | p. 90 |
| 85 | Pseudocódigo do controle da simulação através da tensão no topo. | p. 91 |
| 86 | Metodologia adotada no DYNASIM para obter a configuração inicial da linha. | p. 91 |

Lista de Tabelas

| | | |
|---|--|-------|
| 1 | Propriedades físicas e geométrica do elemento. | p. 74 |
| 2 | Dados do perfil de corrente marítima. | p. 74 |
| 3 | Dados da bóia. | p. 74 |

1 *Introdução*

Nas últimas décadas houve um grande avanço na utilização de cabos como elemento estrutural na área da engenharia. Isso ocorreu devido a vantagens notáveis de estruturas que utilizam cabos, tais como: aparência estética, utilização eficiente e economia (Karoumi, 2000).

A utilização de cabos estruturais se estende a diversas áreas da engenharia. As estruturas mais comuns que utilizam os cabos como elemento são: pontes estaiadas, pontes pênses, torres de transmissão de energia etc. Na Figura 1 são mostradas imagens de algumas dessas estruturas.



Figura 1: Imagens de estruturas que utilizam cabos estruturais.

Outra utilização de elementos que atuam como cabos estruturais pode ser encontrada em sistemas *offshore*, mais especificamente nas linhas de ancoragem, que tem a função de amarração de plataformas e navios em águas profundas. Além das linhas, os *risers*, que são dutos que conduzem o óleo do fundo do mar até a superfície, têm um comportamento parecido com comportamento dos cabos estruturais. Esses sistemas *offshore* são utilizados no processo de exploração de petróleo e configuram a forma de exploração mais utilizada pelas empresas que realizam este tipo de exploração em águas profundas, como a PETROBRAS. Neste trabalho, utiliza-se a palavra “linhas” para se referir ao conjunto linhas de ancoragem e *risers*.

Com a crescente expansão das atividades de exploração de petróleo em águas ultra-profundas, tem-se motivado o desenvolvimento de sistemas computacionais para simular o comportamento dinâmico não linear do conjunto: casco (plataformas e navios) e suas linhas. Esses sistemas têm sido muito utilizados nas etapas de projeto, instalação e operação dessas estruturas.

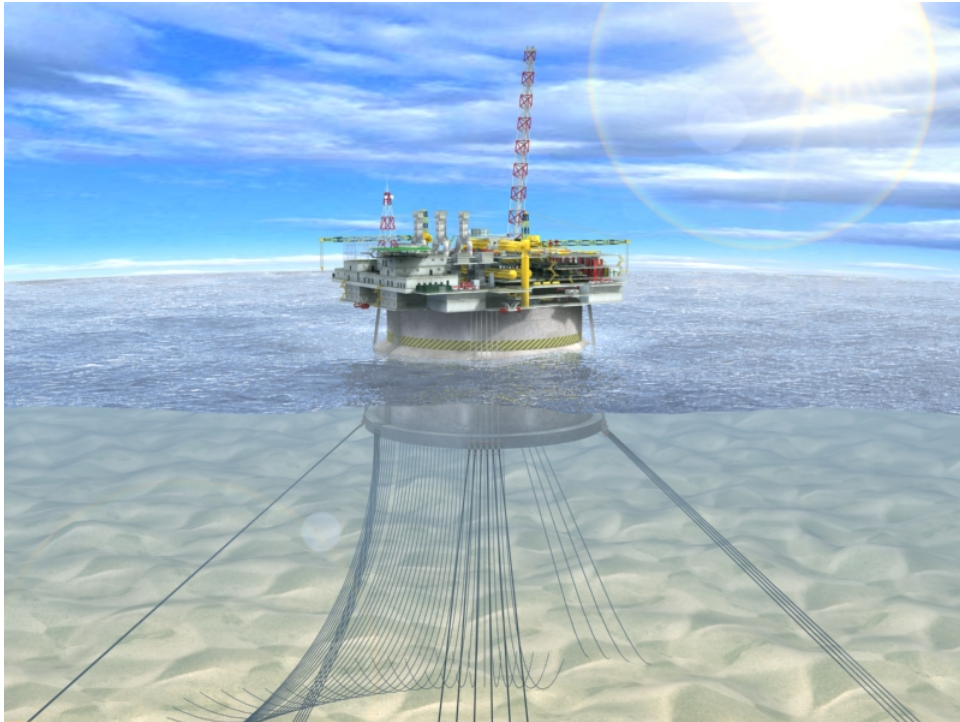
Contudo, para simular o comportamento das linhas é importante conhecer as ações externas que atuam sobre elas. As principais ações consideradas nos projetos de estruturas *offshore* são resultantes da interação fluido-estrutura. No caso dos cabos das torres de transmissão e das pontes, a ação do vento tem um efeito bastante considerável. Já no caso das linhas de ancoragem e dos *risers*, a ação da corrente marítima tem uma influência significativa. Ainda no caso das linhas, existem outras ações externas que podem atuar conjuntamente, tais como: a ação restauradora de contato decorrente do efeito do solo marinho e o movimento imposto pelo flutuador sobre a linha proveniente, por exemplo, da influência do flutuador.

A solução analítica de cabos sob o efeito dinâmico não é trivial, principalmente quando se consideram as não linearidades envolvidas no problema. Isso mostra a importância da aplicação de métodos numéricos na solução desse tipo de estrutura, que aliás vêm sendo bastante utilizados ao longo dos últimos anos.

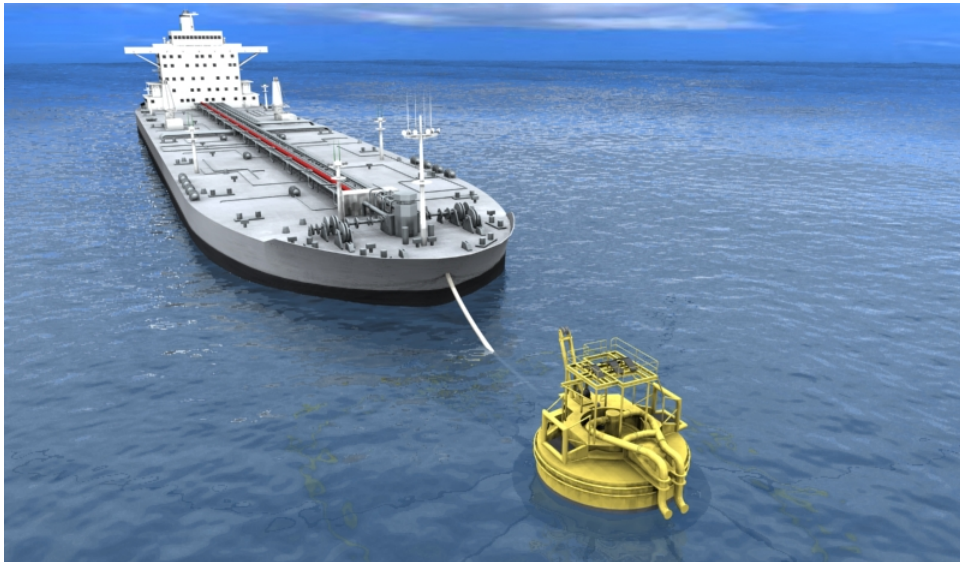
Em linhas gerais, o problema tratado neste trabalho se refere à análise dinâmica estrutural tridimensional de linhas de ancoragem e *risers* submetidos às condições naturais de instalação e operação dessas estruturas.

Na Figura 2(a) é ilustrada uma plataforma com as linhas de ancoragem e *risers*, compondo a estrutura *offshore*. Por sua vez, na Figura 2(b) é mostrada uma operação de *offloading*, onde é possível perceber o duto que faz a transferência de óleo de uma plataforma para um navio aliviador. Os exemplos mostrados nessas figuras têm

por objetivo ilustrar as aplicações que motivam o desenvolvimento deste trabalho, caracterizando o problema físico através da visão geral proporcionada por elas.



(a) Estrutura *offshore* com linhas de ancoragem e *risers*



(b) Detalhe da operação de *offloading*

Figura 2: Estrutura e operação *offshore*.

A maioria dos sistemas computacionais de cálculo de estruturas *offshore* utilizam os métodos numéricos para analisar dinamicamente essas estruturas, sempre considerando as não linearidades físicas e geométricas. Dentre alguns desses sistemas, que são específicos para a análise de linhas, podem ser destacados: ANFLEX (Mourelle *et al.*,

1995; Mourelle *et al.*, 2001), ORCAFLEX (Orcina, 2005), FLEXCOM (MCS, 2005a) e ARIANE-3Dynamic (MCS, 2005b). Além desses, existem sistemas que fazem análise acoplada entre os corpos flutuantes e as linhas e *risers* conectados a eles, como: PROSIM (Jacob; Masetti, 1998), DYNASIM (Coelho *et al.*, 2001; Fucatu; Nishimoto, 2003) e TPN (Nishimoto *et al.*, 2004).

Um outro sistema para análise dinâmica de linhas de ancoragem e *risers* é o PREADYN, que foi desenvolvido por Silveira (2001). Trata-se de um sistema integrado que apresenta recursos de pré-processamento, análise numérica interativa-adaptativa e pós-processamento. Esse sistema surgiu com o objetivo principal de processar análises dinâmicas de linhas de ancoragem e de *risers* isoladamente, em análises desacopladas utilizadas nas etapas de projeto e instalação de linhas, para exploração de petróleo em águas ultra-profundas.

Então, com a utilização do PREADYN pela indústria do petróleo, sugeriu-se que o módulo de análise numérica desse sistema fosse incorporado a outros sistemas computacionais, tais como DYNASIM e TPN. Tais sistemas, que originalmente realizavam análises desacopladas do casco, passaram a incorporar o PREADYN. Isso viabilizou a realização de análises de sistemas de ancoragem de forma semi-acoplada (casco + linhas), uma vez que não é construída uma matriz de rigidez do sistema (casco + linhas) e não é considerado o efeito da linha sobre a embarcação para todo *time-step*.

Os sistemas DYNASIM, TPN e PREADYN foram desenvolvidos a partir de projetos do CENPES/PETROBRAS em parceria com a UFAL, USP, PUC-Rio e COPPE. Esses sistemas foram utilizados com sucesso pela PETROBRAS na análise e projeto de plataformas desde 1996, a exemplo da: MONOBR, FPSOBR, P43, P34 e Espadarte.

1.1 Motivação

Com a incorporação do PREADYN aos sistemas supracitados, houve um crescimento natural na utilização das rotinas de análise do PREADYN pelo CENPES e por algumas universidades do país, que desenvolvem trabalhos no setor de petróleo, como a PUC-Rio, a USP e a COPPE/UFRJ.

Isso ocasionou um aumento da demanda por implementações de novos modelos e métodos, de tal maneira que o módulo original precisou ser sistematicamente expandido e melhorado, de forma a se tornar cada vez mais genérico, robusto e eficiente para atender a demanda existente.

À medida que as solicitações por novas implementações foram sendo realizadas nesse módulo de análise, a manutenção e a expansão dos programas envolvidos foram se tornando cada vez mais difíceis e trabalhosas, uma vez que o módulo estava inserido no contexto de cada um desses programas. Por eles terem características próprias tornavam a tarefa de atualização bastante árdua. Além disso, cada implementação e/ou correção realizada em um dos programas teria que ser refeita nos outros sistemas que utilizam esse módulo, dificultando inclusive o gerenciamento de seus códigos fontes.

Então, com base na problemática apresentada acima é possível identificar algumas razões que motivam o desenvolvimento deste trabalho. Dentre essas razões podem ser destacadas:

- Necessidade de uma ferramenta computacional capaz de facilitar a reutilização do módulo de análise que contribua para o uso em diversos sistemas;
- Falta de mecanismo que facilite a manutenção e expansão do módulo para análise de linhas com maior simplicidade;
- Necessidade de técnicas computacionais que proporcionem o desenvolvimento de um módulo de análise único, para auxiliar na manutenção dos sistemas.

Outro fator que motiva o desenvolvimento deste trabalho está intrinsecamente relacionado à capacidade de expansão do módulo de análise, tornando-o uma plataforma básica para o desenvolvimento de pesquisas e projetos, de modo que os resultados desses trabalhos possam ser aproveitados por outros alunos de graduação e pós-graduação. Dessa forma, novas implementações não impactam em novas tarefas de manutenção das aplicações, que recebem as melhorias de forma natural, seguindo o exemplo de sucesso do FEMOOP (Martha; Parente Jr., 2002).

1.2 Soluções

O problema de reutilização de programas computacionais não é uma necessidade atual. Desde as primeiras linguagens de computação, o reuso de trechos de código eram obtidos através de desvios condicionais. Na década de 1970 o reuso era alcançado com a utilização da programação baseada em módulos e sub-rotinas. Já na década de 1980, com o surgimento da programação orientada a objetos, foi introduzido o reuso por herança. Recentemente, na década de 1990, o reuso de programas começou a ser feito por meio

de análise de domínio, componentes, padrões de projeto e *frameworks*, persistindo até os dias de hoje (Braga, 2004).

Segundo Guimarães (2000), os *frameworks* promovem o reuso de código devido ao conjunto de classes que eles fornecem a determinada linguagem e sistema. Mais ainda, possibilitam a reutilização de análise e projeto, uma vez que as classes dos *frameworks* representam o projeto de um sistema abstrato. Além disso, Guimarães afirma que o reuso de análise e projeto é mais importante do que a reutilização de código. Ele justifica dizendo: “gasta-se mais tempo na análise e projeto de um sistema do que em sua codificação”.

Além dessa questão da reutilização, os *frameworks* orientados a objetos possibilitam uma boa manutenção, organização e expansão de código. Isso é possível devido à utilização da filosofia orientada a objetos, que possui mecanismos como, por exemplo, a herança, que é capaz de facilitar a expansão do sistema de forma ordenada. Os conceitos sobre tais mecanismos são apresentados na seção 1.5.

Fundamentado no que foi exposto neste tópico, a proposta central deste trabalho consiste no desenvolvimento de um *framework* orientado a objetos baseado no módulo de análise numérica do PREADYN. Com isso, cria-se um ambiente que facilite a manutenção e implementação de novos desenvolvimentos no *framework* gerando benefícios a todos os sistemas computacionais que fizerem uso do mesmo.

1.3 Revisão Bibliográfica

De acordo com Mattsson (2000), o primeiro *framework* largamente utilizado foi criado para interface com o usuário, da Smalltalk-80, chamado *Model-View-Controller* (MVC) (Goldberg, 1984). A Apple Inc. desenvolveu o *MacApp* (Schmucker, 1986), um outro *framework* de interface com o usuário, que foi projetado para suportar implementações de aplicações da Macintosh. No entanto, os *frameworks* só despertaram maior interesse quando foram desenvolvidos o *InterView* (Linton *et al.*, 1989) e o *ET++* (Weinand *et al.*, 1988), que são *frameworks* de interface com o usuário, e foram disponibilizados no mercado. Sendo assim, era possível a interação com outras linguagem de programação. Existem também muitos *frameworks* comerciais de interface com o usuário, por exemplo, *zApp*, *OpenStep* e *Microsoft Foundation Class* (MFC).

Viljamaa (2001) diz que o sucesso e o grande número de *frameworks* de interface com o usuário causou a falsa idéia de que *frameworks* são limitados somente a interfaces com o

usuário. Todavia, existem muitos *frameworks* para diferentes domínios de aplicação, por exemplo, para sistemas de hipermídia (Meyrowitz, 1986), sistemas psicofisiológico (Foote, 1988), editores de desenho (Vlissides; Linton, 1989), sistemas operacionais (Russo, 1990), compiladores (Järnvall *et al.*, 1995), programas de protocolo de internet (Hüni *et al.*, 1995), sistemas de alarmes contra incêndio (Molin; Ohlsson, 1998), entre outros.

Contudo, fazendo uma varredura na literatura técnica, não é possível identificar *frameworks* com as mesmas características do apresentado neste trabalho, tais como: análise dinâmica de cabos, algoritmos de integração direta, método dos elementos finitos, dentre outras características. Porém, é possível encontrar *frameworks* com características semelhantes, por exemplo, *frameworks* para simulação numérica (Beall; Shephard, 1999), mecânica computacional (Sahu *et al.*, 1999), análise numérica interativa (Bettig; Han, 1999) etc.

1.4 Objetivos

Este trabalho tem como objetivo geral desenvolver um *framework* para a análise dinâmica de linhas de ancoragem e de *risers*, com base no módulo de análise numérica do PREADYN, para a utilização em sistemas de análise de estruturas *offshore*. Além disso, este trabalho tem como objetivos específicos:

- Modelar o *framework* empregando uma linguagem de modelagem gráfica;
- Implementar o *framework* utilizando o paradigma da programação orientada a objetos, através da linguagem C++;
- Validar o *framework* a partir da incorporação dele em sistemas existentes, por usuários distintos;
- Contribuir para o melhoramento da modelagem do problema físico, como a criação de um novo modelo constitutivo;
- Tornar o *framework* um sistema computacional que sirva de base para o desenvolvimento de outros sistemas e pesquisas na área de estruturas *offshore*.

1.5 Definição de Conceitos

Para um melhor entendimento deste trabalho faz-se necessário definir alguns conceitos que são utilizados no decorrer do texto. Portanto, são apresentados neste capítulo conceitos sobre orientação a objetos, *framework* e UML.

1.5.1 Orientação a Objetos

O termo *orientação a objetos* refere-se a uma organização de um programa computacional em termos de coleção de objetos discretos, que incorporam estrutura e comportamento próprios. Essa abordagem de organização difere daquela existente no desenvolvimento tradicional de programas computacionais, onde estruturas de dados e rotinas são desenvolvidas de forma apenas fracamente acopladas. As definições dos termos associados a orientação a objetos descritos abaixo são em parte baseados no trabalho de Ricarte (1995).

Pode-se dizer que um **objeto** é uma entidade do mundo real que tem uma identidade. Ele pode representar entidades concretas (um carro, uma bola, um arquivo de computador etc.) e entidades conceituais (uma estratégia de jogo, uma política de escalonamento em um sistema operacional etc). A identidade de cada objeto é importante para distingui-los, mesmo que eles tenham as mesmas características.

A estrutura de um objeto é representada em termos de **atributos**. O comportamento dele é representado por um conjunto de **operações** que podem ser executadas sobre os atributos. Os objetos que têm a mesma estrutura e comportamento são agrupados em **classes**. Uma classe é uma abstração que descreve propriedades importantes para uma aplicação. Cada classe descreve um conjunto de objetos, onde cada objeto é dito ser uma instância da classe.

Um atributo é um valor de dado assumido pelos objetos de uma classe. Cada atributo tem um valor para cada instância da classe. Uma operação ou método é uma função ou transformação que pode ser aplicada a uma classe ou a um objeto dela. Todas as instâncias de uma classe compartilham das mesmas operações.

Com o objetivo de separar os aspectos externos dos objetos (que são acessíveis a outros objetos) dos detalhes internos da implementação dos mesmos (que permanece isolado dos outros objetos), define-se **encapsulamento**. O encapsulamento é uma ferramenta que evita que o programa torne-se tão interdependente que uma pequena

mudança tenha grandes efeitos colaterais. O uso do encapsulamento permite que um objeto seja modificado, em termos de implementação, sem afetar os programas que o utilizam.

Um diferencial da linguagem de programação orientada a objetos em relação a outros tipos de linguagens está no conceito de herança. **Herança** é um mecanismo que compartilha semelhanças entre classes, preservando as diferenças entre elas. Ela relaciona uma classe e suas versões especializadas. A tal classe (generalizada) é chamada de superclasse ou classe base e as versões especializadas dela são chamadas de subclasses ou classes derivadas. Quando os atributos e operações são comuns a um grupo de subclasses eles passam a pertencer a superclasse. A classe derivada herda as características de sua classe base, bem como as características de todos seus ancestrais.

Geralmente, imagina-se que as classes têm definições completas. Entretanto, existem situações onde definições incompletas são úteis. Sendo assim, classes que apresentam definições incompletas são igualmente úteis. As **classes abstratas** são classes que incorporam conceitos coesivos e coerentes, porém, incompletos. Por sua vez, as características dessas classes estão disponíveis nas versões especializadas delas, através da herança. Não se pode criar instâncias de classes abstratas, porém pode-se fazer uso de suas características individuais que estão disponíveis nas suas versões especializadas (subclasses). Classe abstrata também pode ser chamada de *tipo parcial* e *superclasse abstrata* (Berard, 1998). Dessa forma, tem-se a definição de **classe concreta** por exclusão, sendo considerada classe concreta aquela que não é denominada de classe abstrata.

Um outro termo importante da orientação a objetos é o **polimorfismo**, que é uma palavra de origem grega que quer dizer: muitas formas. No entanto, no contexto da orientação a objetos, é usada para denotar que um mesmo nome pode se referir a muitos métodos diferentes. Alguns autores classificam dois tipos de polimorfismo: ***overloading*** e ***overriding***. *Overloading* refere-se a capacidade de definir diversos métodos com o mesmo nome dentro de uma determinada classe, possuindo assinaturas diferentes. A assinatura de um método é definida pelo número, tipo e ordem de seus parâmetros. *Overriding* é a forma de polimorfismo mais complexa, que ocorre quando uma subclasse tem um método com o mesmo nome e assinatura de um método da superclasse. Quando isso acontece o método da subclasse cancela o método da superclasse. Então, se um objeto da subclasse estiver atribuído a uma referência da superclasse, e um método da superclasse for invocado que é *overridden* no objeto da subclasse, através do polimorfismo o método correto a ser

invocado é do objeto da subclasse. É importante mencionar que o polimorfismo por *overriding* só ocorre dentro de uma hierarquia de classe (Hanna, 2005).

1.5.2 *Framework* Orientado a Objetos

Muitos autores aceitam que um *framework* orientado a objetos é um programa de arquitetura reutilizável incluindo projeto e código. Contudo, essa definição geralmente não é aceita e constitui, apenas, uma parte dela (Mattsson, 2000). Provavelmente a definição com maior referência foi apresentada por Johnson & Foote (1988), diz que:

Framework é um conjunto de classes que incorpora um projeto abstrato para soluções de uma família de problemas associados.

Então, um *framework* consiste de um conjunto de classes (podendo não ser abstrata), cujas instâncias trabalham em conjunto. É esperado que ele seja expansível, isto é, reutilizável (projeto abstrato). Não é obrigatório que ele faça uso completo do domínio de aplicação, permitindo a composição de *frameworks*. Além disso, *frameworks* são expressos em uma linguagem de programação, provendo reuso de código e projeto (Mattsson, 2000).

Em outras palavras, *framework* é um projeto e uma implementação parcial de uma aplicação para um dado domínio do problema. Quando se discute conceitos de *framework*, normalmente surgem dificuldades terminológicas devido ao fato de não existir uma definição comum de *framework* (Mattsson, 2000).

Framework é construído para abstrair diversas aplicações de um determinado domínio, por exemplo, um *framework* para abstrair diversos editores de texto. Obtém-se o que esses editores têm em comum e produz-se um sistema abstrato, genérico, admitindo-se que todas as classes desse sistema são abstratas. Contudo, não é possível criar objetos de classes abstratas e sendo assim esse conjunto de classes abstratas (*framework* - editor de texto) não pode ser caracterizado como uma aplicação, mas pode ser transformado em uma aplicação criando-se subclasses, compondo-se classes existentes e criando-se classes auxiliares. Como no exemplo em questão o *framework* é uma abstração de um editor de texto genérico, ele é suscetível a mudanças. Diante disso, vários editores diferentes podem ser derivados dele apenas criando-se classes, subclasses etc. Quando um *framework* é transformado em uma aplicação concreta diz-se que foi feita a instanciação do *framework*. O programador responsável por isso recebe o nome de instanciador ou usuário instanciador (Guimarães, 2000).

1.5.2.1 Benefícios de um *framework*

De acordo com Fayad & Schmidt (1997), os benefícios básicos dos *frameworks* orientados a objetos originam-se da modularidade, reusabilidade e extensibilidade que eles provêm ao desenvolvedor (usuário instanciador), como descrito abaixo:

- **Modularidade:** os *frameworks* melhoram a modularidade pelo encapsulamento de detalhes temporários da implementação atrás das interfaces estáveis. A modularidade ajuda o *framework* a melhorar a qualidade do programa localizando mais facilmente o impacto de mudanças no projeto e na implementação. Essa localização reduz o esforço necessário para compreender e manter programas existentes;
- **Reusabilidade:** as interfaces estáveis, fornecidas pelos *frameworks*, melhoram a reusabilidade através da definição de componentes genéricos, que podem ser reaplicáveis para criar novas aplicações. A reusabilidade de *frameworks* influencia no conhecimento do domínio e no esforço prévio de desenvolvedores experientes, a fim de evitar que soluções comuns sejam recriadas e re-validadas. O reuso de componentes de *frameworks* pode render melhorias substanciais na produtividade do usuário instanciador, bem como melhorias na qualidade, no desempenho, na confiabilidade e na interoperabilidade do programa;
- **Extensibilidade:** os *frameworks* melhoram a extensibilidade devido ao fornecimento de métodos explícitos, permitindo que as aplicações estendam as interfaces estáveis deles. Esses métodos desacoplam as interfaces estáveis e os comportamentos de um domínio de aplicação das variações requeridas pelas instanciações de uma aplicação em um contexto particular. A extensibilidade do *framework* é essencial para assegurar personalização oportuna de serviços e de características de uma nova aplicação.

1.5.2.2 Diferença entre um *framework* e uma biblioteca

Bibliotecas de classe contêm classes separadas que podem ser usadas independentemente umas das outras: o usuário instancia as classes e chama os métodos dela. O uso dessas bibliotecas são muito semelhantes ao uso de módulos e bibliotecas de subrotina. Bibliotecas de classe são principalmente focadas em reuso de código, raramente em análise e projeto. Muitas delas oferecem serviços gerais, tais como estruturas de dados e

stream IO. Por outro lado, os *frameworks* são mais dependentes do domínio de aplicação. Eles consistem de classes relacionadas cujos os objetos interagem uns com os outros. Quando se usa *frameworks*, a idéia é reutilizar toda a arquitetura do sistema e não somente classes individuais. Dessa forma, o uso de *framework* reduz mais a quantidade de código de uma aplicação específica do que usando bibliotecas da classe (Viljamaa, 2001).

O princípio *Hollywood* (*Não nos chame, nós chamaremos você*) é tradicionalmente considerado como um dos fatores que diferenciam *frameworks* de bibliotecas de classe. Quando se usa bibliotecas, o código da aplicação é responsável pelo controle de fluxo. Já nos *frameworks*, o método principal está contido dentro dele e ele chama o código da aplicação e “vice-versa” (Viljamaa, 2001).

1.5.3 UML

Segundo Rumbaugh *et al.* (2000), UML (*Unified Modeling Language*) é uma linguagem para especificar, visualizar e construir os artefatos de sistemas de *software*. Essa linguagem é um sistema de notação voltado à modelagem de sistemas computacionais, que utiliza conceitos de orientação a objetos (Oliveira; Pessoa, 2001).

A UML é um padrão emergente, que está sendo aceito pelos profissionais da área para modelar sistemas orientado a objetos. Ela teve início com o empenho conjunto de Grady Booch e Jim Rumbaugh em 1994, agrupando seus respectivos métodos populares, os métodos Booch e OMT (*Object Modeling Technique*). Depois, Ivar Jacobson, o criador do método OOSE (*Object Oriented Software Engineering*), juntou-se a eles. Em 1997, a UML foi submetida como candidata a uma entidade de padronização, a OMG (*Object Management Group*). A OMG aprovou a submissão da UML, que também recebeu a aceitação pelos profissionais da área. Muitas organizações de desenvolvimento de *software* adotaram a UML na modelagem de seus sistemas (Oliveira; Pessoa, 2001).

Com a UML é possível modelar elementos, relacionamentos, mecanismos de extensibilidade e diagramas. Os diagramas são utilizados para representar modelos. Um modelo é uma descrição completa do sistema em uma determinada perspectiva. Um diagrama pode ser representado de várias formas, dependendo de quem irá interpretá-lo (Neto, 2005). Um dos diagramas mais conhecidos dessa linguagem é o diagrama de classes, o qual pode ser observado na Figura 3.

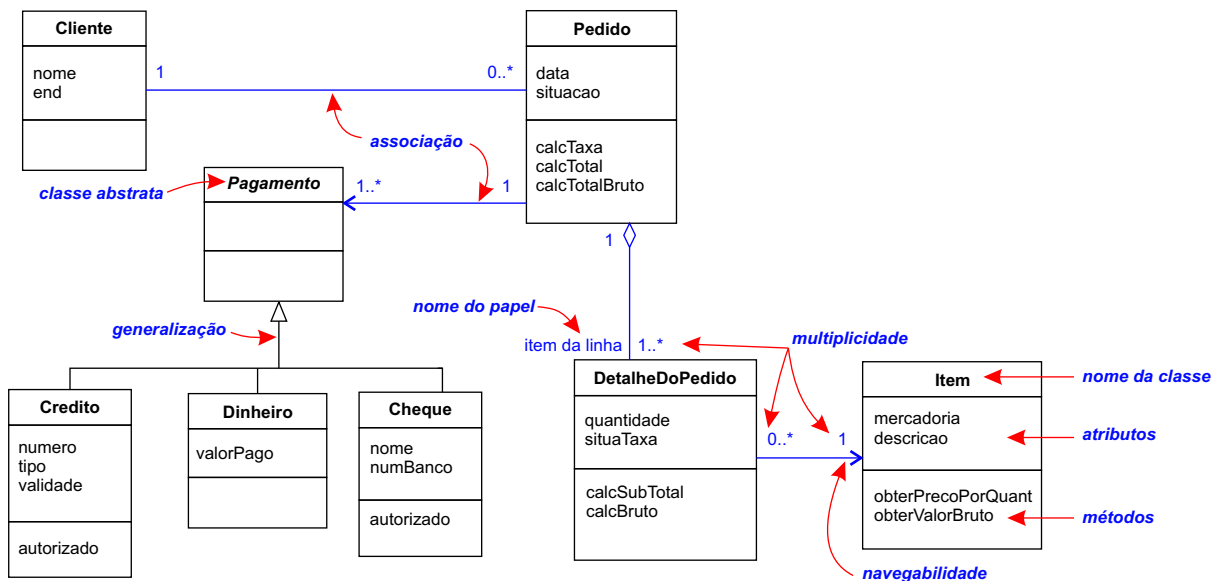


Figura 3: Exemplo de um diagrama de classes da linguagem UML (Neto, 2005).

Com base na Figura 3, pode-se observar algumas entidades e relacionamentos pertencentes ao diagrama de classes da linguagem UML, tais como:

- Entidades:

- Classe concreta: é formada por um retângulo, contendo duas linhas que dividem três partes. A primeira parte contém o nome da classe, a segunda e a terceira contém, respectivamente, os atributos e os métodos da classe;
- Classe abstrata: é semelhante à representação da classe concreta, exceto pelo fato de o nome da classe ser em estilo itálico.

- Relacionamentos:

- Generalização ou herança: é representada por uma flecha com um triângulo vazado na ponta, como acontece na representação da hierarquia formada pela classe **Pagamento**;
- Associação: é representada por uma linha. Quando a associação possui uma seta (navegabilidade) indica a classe que é responsável pela outra. Por exemplo, a classe **DetalheDoPedido** tem “responsabilidade” sobre a classe **Item**;
- Papel: é um rótulo que descreve o relacionamento entre as classes. Por exemplo, o papel “item da linha” definido entre a classe **Pedido** e **DetalheDoPedido**;

- Multiplicidade: são índices que informam a quantidade de instâncias que podem ser criadas. Por exemplo, na associação entre as classes `Cliente` e `Pedido`, uma instância de `Cliente` pode estar associada a nenhuma ou várias instâncias de `Pedido`. Por outro lado, uma instância de `Pedido` só pode estar associada a uma instância de `Cliente`.

1.6 Organização do Trabalho

- Capítulo 2: este capítulo aborda questões relacionadas à modelagem física e numérica do problema, apresentando definições importantes, como o equacionamento do problema, as questões referentes ao problema de dinâmica estrutural e os métodos de solução classicamente utilizados nesse tipo de problema;
- Capítulo 3: apresentado o problema físico e seus métodos de solução, este capítulo descreve a modelagem computacional do *framework*, mostrando uma descrição detalhada das classes propostas, os relacionamentos entre elas e o papel que elas assumem no *framework*. Este capítulo mostra alguns detalhes sobre a implementação computacional do *framework* bem como sua forma de utilização;
- Capítulo 4: apresenta um exemplo e três aplicações do *framework*. No exemplo é construído um exemplo básico e nas aplicações é apresentada a incorporação do *framework* a elas;
- Capítulo 5: são abordadas as considerações finais ressaltando os objetivos alcançados e as sugestões para trabalhos futuros relacionados a este trabalho.

2 *Formulação do Problema*

Este capítulo tem por objetivo descrever as equações de movimento para problemas de dinâmica estrutural, além de apresentar os métodos de integração direta utilizados na análise do tipo *time-history*. A formulação matemática exposta neste capítulo baseia-se na tese de Silveira (2001).

Segundo Cook *et al.* (1989), os problemas de dinâmica podem ser divididos em duas categorias: problemas de propagação de onda e problemas de dinâmica estrutural. Os problemas de propagação de onda estão associados a problema de impacto. As excitações e as conseqüentes respostas estruturais são ricas em altas frequências. Nesse tipo de problema o interesse maior está relacionado aos efeitos de ondas de tensão. Desse modo, o tempo de duração da análise é geralmente curto e é normalmente da ordem do tempo de travessia da onda na estrutura. Um problema que não é de propagação de onda, para o qual a inércia é importante, é chamado de problema de dinâmica estrutural. Nessa categoria, a frequência de excitação é geralmente da mesma ordem das frequências naturais de vibração da estrutura.

2.1 **Equação de Movimento**

Matematicamente, o comportamento dinâmico de sistemas estruturais contínuos pode ser representado através de um Problema de Valor Contorno e um Problema de Valor Inicial, sendo definido por um sistema de Equações Diferenciais Parciais (EDPs), que para o problema tratado neste trabalho correspondem às equações de movimento, e por um conjunto de condições de contorno e de condições iniciais. Para solucionar esse tipo de problema, normalmente são utilizados os métodos numéricos baseados na discretização das equações de movimento de forma independente no espaço e no tempo (Jacob, 1990, ex. ref.).

Discretizando essas equações no espaço (semi-discretização) o problema é reduzido a

um sistema de Equações Diferenciais Ordinárias (EDOs) de segunda ordem no tempo, que devem ser integradas para obtenção da solução do sistema. O método numérico mais utilizado para discretização espacial desse tipo de problema é o Método dos Elementos Finitos (Bathe, 1996).

De posse das equações discretizadas no espaço, ou seja, das equações semi-discretizadas, o passo seguinte consiste na discretização das EDOs ao longo do tempo. Os algoritmos de integração no tempo levam a solução aproximada para as EDOs geradas na semi-discretização. Em cada passo de tempo são obtidas as acelerações, velocidades e deslocamentos para cada nó existente na estrutura discretizada. Existem diversos tipos de algoritmos de integração que podem ser utilizados para solucionar esse tipo de problema.

Em problemas de dinâmica estrutural, as equações de movimento semi-discretizadas são obtidas a partir da consideração do equilíbrio em determinado instante de tempo t , incluindo os efeitos das forças inerciais, das forças de amortecimento, forças externas e dos esforços internos. Assim, a equação geral de movimento pode ser escrita como:

$$F_I(t) + F_{int}(t) = F_{ext}(t), \quad (2.1)$$

onde $F_I(t)$ corresponde às forças inerciais, $F_{int}(t)$ aos esforços internos da estrutura e $F_{ext}(t)$ às forças externas aplicadas à estrutura.

O esforço interno pode ser expresso como:

$$F_{int}(t) = F_{amort}(t) + F_{elast}(t), \quad (2.2)$$

onde $F_{amort}(t)$ refere-se às forças de amortecimento e $F_{elast}(t)$ às forças elásticas. Considera-se que as forças elásticas incorporam os efeitos devido às tensões iniciais. Observa-se que todas as parcelas da equação geral de movimento (2.1) são dependentes do tempo. Normalmente, essa equação é expressa em função dos deslocamentos $U(t)$, velocidades $\dot{U}(t)$ e acelerações $\ddot{U}(t)$, como é apresentado nas próximas seções.

2.1.1 Problemas Lineares

No caso dos problema lineares as parcelas da equação de movimento semi-discretizada (2.1) podem ser escrita como:

$$F_I(t) = M\ddot{U}(t) \quad (2.3)$$

e

$$F_{int}(t) = F_{amort}(t) + F_{elast}(t) = C\dot{U}(t) + KU(t). \quad (2.4)$$

Assim, a equação de movimento que governa a resposta do problema linear de dinâmica semi-discretizado pode ser expressa como:

$$M\ddot{U}(t) + C\dot{U}(t) + KU(t) = F_{ext}(t), \quad (2.5)$$

onde M é a matriz de massa da estrutura, C é a matriz de amortecimento e K é a matriz de rigidez. No âmbito do método dos elementos finitos, os vetores $U(t)$, $\dot{U}(t)$ e $\ddot{U}(t)$ são os vetores formados por componentes nodais dos deslocamentos, velocidades e acelerações, respectivamente. Para o caso do problema linear as matrizes C e K são constantes ao longo do tempo.

As condições iniciais que compõem o problema de valor inicial da Equação 2.5 são fornecidas por:

$$U(0) = U_0 \quad e \quad \dot{U}(0) = \dot{U}_0, \quad (2.6)$$

onde U_0 e \dot{U}_0 são valores iniciais para os deslocamentos e velocidades, respectivamente. Caso não sejam conhecidas as acelerações iniciais \ddot{U}_0 , elas podem ser determinadas a partir da expressão:

$$\ddot{U}_0 = M^{-1}(F_{ext}(0) - C\dot{U}_0 - KU_0). \quad (2.7)$$

2.1.2 Problemas Não Lineares

Para problemas não lineares as parcelas da equação geral de movimento (2.1), discretizada apenas no espaço, podem ser escritas como:

$$F_I(t) = M\ddot{U}(t), \quad (2.8)$$

$$F_{int}(t) = R(U(t), \dot{U}(t)) \quad (2.9)$$

e

$$F_{ext}(t) = F(U(t), t). \quad (2.10)$$

Dessa forma, para problemas não lineares, a equação de movimento semi-discretizada é expressa por:

$$M\ddot{U}(t) + R(U(t), \dot{U}(t)) = F(U(t), t), \quad (2.11)$$

onde $R(U(t), \dot{U}(t))$ corresponde ao vetor dos esforços internos nodais para um estado de tensões na configuração do instante t e $F(U(t), t)$ ao vetor de carregamentos externos aplicados aos pontos nodais na configuração relativa ao instante t . O vetor $R(U(t), \dot{U}(t))$ está associado a mudanças nas propriedades dos materiais, como plasticidade (não linearidade física) e/ou mudanças na configuração, como em grandes deflexões de uma viga elástica delgada (não linearidade geométrica). Por sua vez, o vetor $F(U(t), t)$ considera a não linearidade devido à variação das cargas externas com a geometria, caracterizando carregamentos não conservativos (Silveira, 2001).

As condições iniciais $U(0)$, $\dot{U}(0)$ e $\ddot{U}(0)$ que complementam a Equação 2.11 podem ser descritas de forma semelhante ao problema linear, apresentado na seção anterior.

2.2 Métodos de Integração Direta

Para muitos dos problema de propagação de onda e de dinâmica estrutural, incluindo os de natureza não linear, os métodos de integração direta são os mais convenientes. Muitos desses métodos são populares e a escolha do método é fortemente dependente do problema (Cook *et al.*, 1989).

Os métodos de integração direta são mecanismos que, passo a passo, fornecem funções do tempo que são a solução para as EDPs que regem o movimento. Esses métodos podem ser divididos, basicamente, em duas categorias de métodos, que são abordados nos próximos tópicos.

2.2.1 Métodos Explícitos de Integração Direta

Os métodos explícitos têm a seguinte forma geral:

$$X_{n+1} = f(X_n, \dot{X}_n, \ddot{X}_n, X_{n-1}, \dots) \quad (2.12)$$

e permite, conseqüentemente, que X_{n+1} seja determinado em termo do histórico completo das informações baseadas nos deslocamentos e suas derivadas no tempo, anteriores ao passo corrente (Cook *et al.*, 1989).

Os métodos explícitos são considerados sempre condicionalmente estáveis, ou seja, possuem restrições quanto ao tamanho do incremento de tempo (Δt) utilizado para que seja mantida a condição de estabilidade do método. Normalmente, os incrementos de tempo requeridos por esses métodos, para que seja mantida a estabilidade, são muito

pequenos. O tamanho desses incrementos de tempo, em geral, é ditado pela estabilidade e não pela precisão, uma vez que o incremento requerido para a estabilidade é tão pequeno que se torna suficiente para que se obtenha uma resposta precisa.

Para cada instante de tempo, os métodos explícitos resolvem um sistema do tipo $Ax = b$. Se matriz A for diagonal, ou seja, se as matrizes de massa (M) e de amortecimento (C) forem diagonais, implica na solução imediata do sistema, obtendo o termo x_i pela simples divisão entre b_i e a_{ii} . Isso acontece devido ao desacoplamento do sistema, que para um mesmo intervalo de tempo os deslocamentos de um nó são afetados apenas pelos seus vizinhos.

2.2.2 Métodos Implícitos de Integração Direta

Os métodos implícitos têm a seguinte forma geral:

$$X_{n+1} = f(\dot{X}_{n+1}, \ddot{X}_{n+1}, X_n, \dots) \quad (2.13)$$

e o cálculo de X_{n+1} requer o conhecimento das derivadas no tempo de X_{n+1} , além do histórico completo das informações dos deslocamentos e suas derivadas no tempo, anteriores ao passo corrente.

Para problemas lineares, os métodos implícitos são considerados incondicionalmente estáveis, ou seja, não apresentam restrições em relação ao tamanho do incremento de tempo para efeito da estabilidade do método. Sendo assim, o tamanho do incremento de tempo utilizado é função da precisão desejada e não pela condição de estabilidade do método.

É importante ressaltar que para os métodos implícitos, mesmo que a matriz de massa (M) e de amortecimento (C) sejam diagonais, as equações do sistema resultante ainda serão acopladas. Isso ocorre porque a resolução desse sistema acopla os deslocamentos, velocidades e acelerações de todos os nós da malha dentro de um mesmo intervalo de tempo.

3 *Modelagem do Framework e Descrição de suas Classes*

Este capítulo aborda a modelagem computacional do *framework*. Entenda-se por modelagem computacional, no contexto deste trabalho, como sendo a organização de classes, bem como a relação entre elas e entre suas instâncias. Sendo assim, são apresentados detalhes dessa modelagem, buscando sempre associar a essas classes as definições e conceitos relacionados à formulação do problema físico, apresentada no Capítulo 2.

É apresentada ainda neste capítulo a descrição detalhada das classes, mostrando seus atributos e métodos, a funcionalidade no contexto do *framework*, bem como a forma de utilização do *framework*.

3.1 Modelagem da Solução

Esta seção tem por objetivo fazer a transição entre a modelagem do problema físico, apresentado no capítulo anterior, com a modelagem computacional desse mesmo problema. A Figura 4 mostra um desenho esquemático bidimensional do problema físico, ilustrando alguns componentes que são mencionados no decorrer deste capítulo.

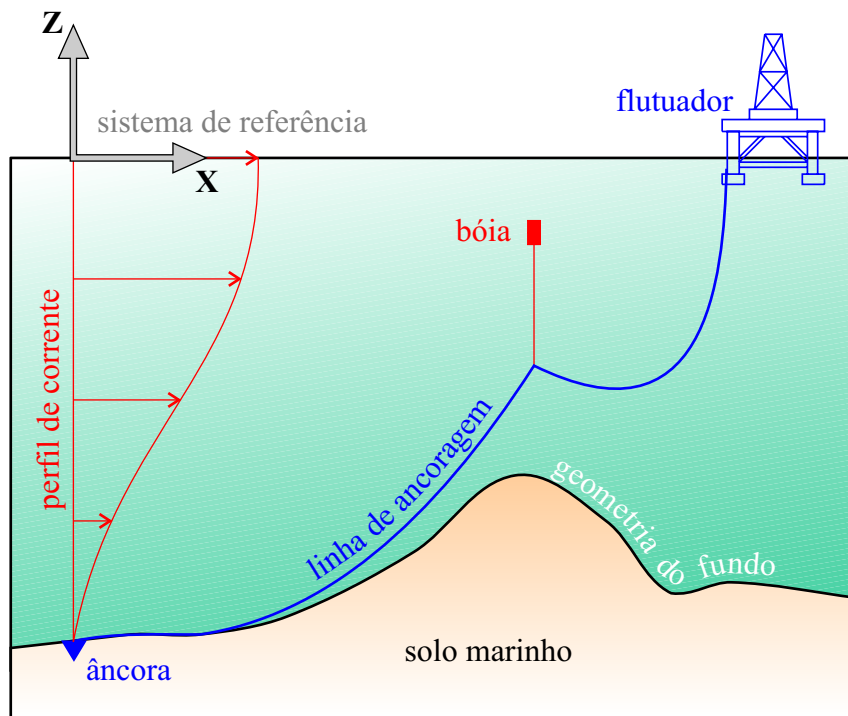


Figura 4: Ilustração do modelo físico.

O modelo computacional (Figura 5) deve ser capaz de representar, a partir de suas abstrações, todos os componentes envolvidos no problema físico, tais como:

- Estrutura/Domínio (linhas): representado pela malha de elementos finitos (nós e elementos);
- Ambiente/Condição de contorno (solo, correnteza, ar etc): representado pelo solo marinho e pelo perfil de velocidade da correnteza;
- Carregamento/Condição de contorno (bóia, poita, deslocamento prescrito etc): representado pela bóia e por ações impostas pelo flutuante (navio ou plataforma).

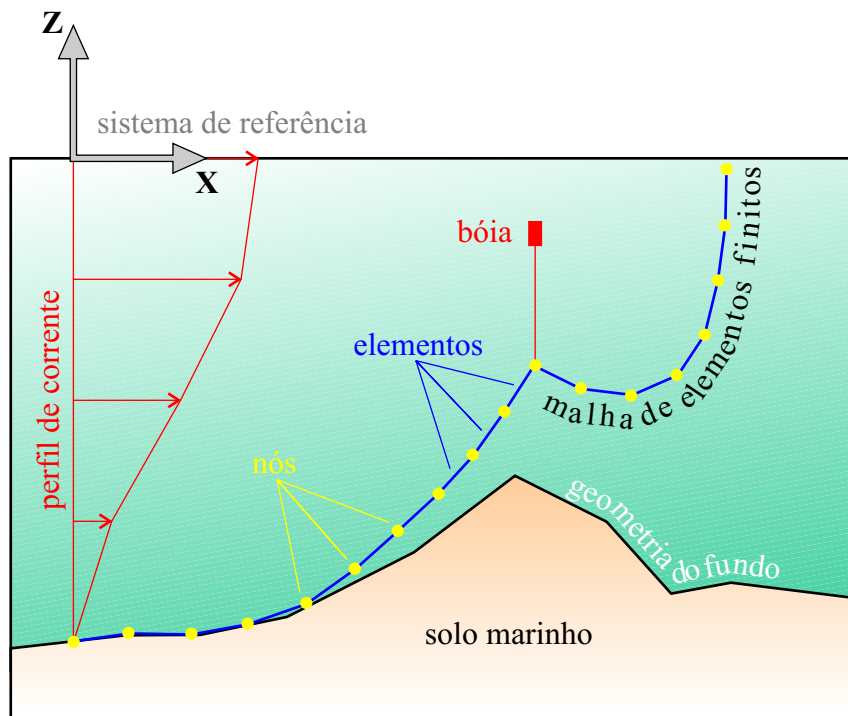


Figura 5: Ilustração do modelo computacional.

Em função do conhecimento do problema físico, das equações envolvidas, dos métodos de solução utilizados e dos componentes que foram mencionados, o *framework* é modelado de acordo com o que está ilustrado na Figura 6, utilizando o padrão de modelagem gráfica UML.

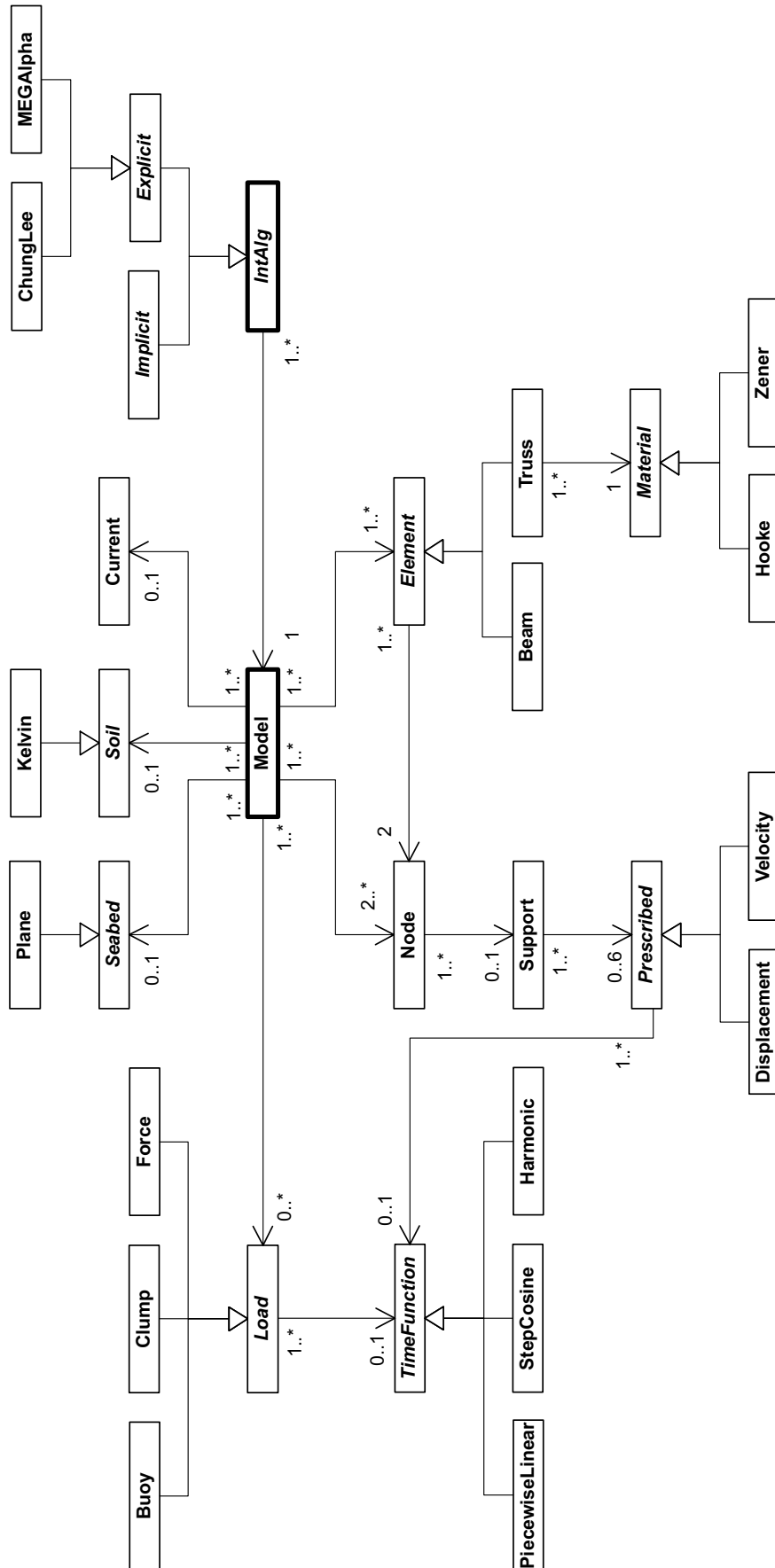


Figura 6: Diagrama de classes do *framework* desenvolvido.

O diagrama de classes apresentado na Figura 6, mostra como é feita a associação de classes do *framework*, o que possibilita ao usuário instaciador ter um conhecimento global do *framework*.

3.2 Descrição das Classes

Nesta seção são descritas as classes que compõem o *framework* desenvolvido (Figura 6). Os tópicos descritos abaixo mostram os principais atributos e métodos da classe, através de um diagrama de classes simplificado. É apresentado também o papel que cada classe assume no *framework* e a abstração relacionada ao problema físico que ela representa. Além disso, ao final da descrição de cada classe concreta, faz-se uma pergunta relacionada à associação dela a outra classe, de acordo com o diagrama da Figura 6. Essa descrição tem o objetivo de mostrar ao usuário instaciador como o *framework* deve ser utilizado.

A utilização do *framework* baseia-se na instanciação de algumas classes. Uma das principais instâncias que deve ser criada é a da classe **Model**, que representa o modelo computacional. Essa instância precisa ser associada a outras instâncias para a construção do modelo, tais como as das classes:

- **Soil**: representa as propriedades físicas do solo marinho;
- **Current**: está relacionado ao perfil de corrente existente ao longo da profundidade;
- **Seabed**: refere-se à geometria do fundo do mar;
- **Load**: representa os carregamentos pontuais que podem atuar sobre a estrutura;
- **Element**: trata-se dos elementos gerados na discretização espacial do modelo (malha de elementos finitos);
- **Node**: semelhante à classe **Element**, trata-se dos nós da discretização espacial do modelo.

Como pode ser visto no diagrama da Figura 6, uma instância da classe **Node** pode estar associada a uma instância da classe **Support**, que representa a condição de apoio. Por sua vez, uma instância dessa classe pode estar associada a uma instância da classe **Prescribed**, que representa alguns tipos de prescrições (deslocamento, velocidade e

aceleração). Além disso, uma instância da classe **Truss** deve estar associada a uma instância da classe **Material**, que representa os modelos constitutivos.

Por fim, é criada uma instância da classe **IntAlg**, que abstrai os diversos tipos de algoritmos de integração. Em seguida, é feita a associação dela ao modelo computacional (instância da classe **Model**). Dessa forma, o *framework* está preparado para iniciar a simulação.

As classes do *framework* estão implementadas na linguagem computacional C++, de acordo com o diagrama da Figura 6 e com o papel que cada classe exerce nele. Essa linguagem é empregada por possuir suporte a programação orientada a objetos e por ser bastante utilizada, o que pode possibilitar uma maior utilização do *framework*.

3.2.1 Classe Model

Esta classe representa uma abstração do modelo físico a ser analisado, relacionado a um problema de análise de linhas de ancoragem e de *risers*. A partir da classe **Model** é possível modelar computacionalmente problemas dessa natureza. Essa classe contém toda a estrutura relacionada ao modelo que se deseja analisar.

Quando se cria uma instância de **Model** deve-se necessariamente definir uma estrutura de nós e elementos através da criação de instâncias das classes **Node** e **Element**, respectivamente. A partir disso, tem-se a configuração básica do modelo definida apenas com a malha de elementos finitos (nós, elementos e conectividades). Feito isso, podem ser definidos os carregamentos que irão atuar sobre a estrutura como, por exemplo, bóias, poitas, forças aplicadas, deslocamento prescritos etc. São apresentados, nos próximos tópicos, como tais carregamentos são criados e associados aos nós. Além disso, é possível associar ao modelo como um todo a ação da correnteza marítima e do solo marinho, através da criação de instâncias das classes **Soil** e **Current**, respectivamente.

A classe **Model** gerencia toda a modelagem que é feita através do *framework*, o que a torna uma das principais classes desenvolvidas. Na Figura 7 pode ser visto o diagrama de classe com os principais métodos e atributos da classe **Model**. Esses métodos referem-se ao cálculo da matriz de massa, do vetor de forças internas e de forças externas. Já os atributos principais são relativos à malha (nó e elemento), ao solo, à geometria do fundo e ao perfil de corrente.

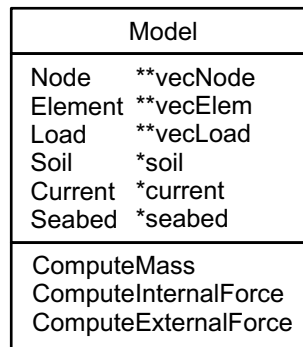


Figura 7: Diagrama de classe de `Model`.

A versão atual do *framework* contempla a solução de problemas de dinâmica estrutural com a utilização de um modelo de massa concentrada e de algoritmos explícitos de integração no tempo. Para implementação computacional de outros algoritmos de integração no tempo, talvez seja necessária a implementação de um modelo que utilize matriz de massa consistente e também a matriz de rigidez do sistema.

Os métodos para calcular a matriz de massa consistente, matriz de amortecimento e matriz de rigidez ainda não foram criados, por isso esses métodos não estão representados no diagrama da Figura 7. Os algoritmos de integração que estão implementados no *framework* são algoritmos explícitos, que por sua vez não necessitam de tais matrizes. Contudo, a criação desses métodos não irão prejudicar, de nenhuma forma, a organização atual do *framework* e nem haverá a necessidade de modificar o que já existe nele.

► *Como construir o modelo?*

Como a classe `Model` é responsável pela abstração de um modelo físico, faz-se necessário que o usuário instanciador crie uma instância dessa classe. Para isso, é preciso definir a quantidade de nós e elementos da malha, como ilustra o trecho de código da Figura 8. Com a instância da classe `Model` criada pode ser iniciada a construção do modelo.

```
double numNode; // Número de nós da malha
double numElem; // Número de elementos da malha
...
Model *model = new Model(numNode, numElem);
```

Figura 8: Criação de uma instância de `Model`.

3.2.2 Classe Soil

Esta é uma classe abstrata que representa a interação entre o solo marinho e a estrutura (linha de ancoragem e *riser*). Existem diversas metodologias de análise para se representar esse tipo de interação. Uma dessas metodologias é apresentada no trabalho de Lages *et al.* (2002). Nesse trabalho a interação solo-estrutura é representada através de uma distribuição de forças nas direções transversais e longitudinais à linha, como ilustra a Figura 9. No próximo tópico é apresentada a subclasse de `Soil`, que representa a interação solo-estrutura no *framework*.

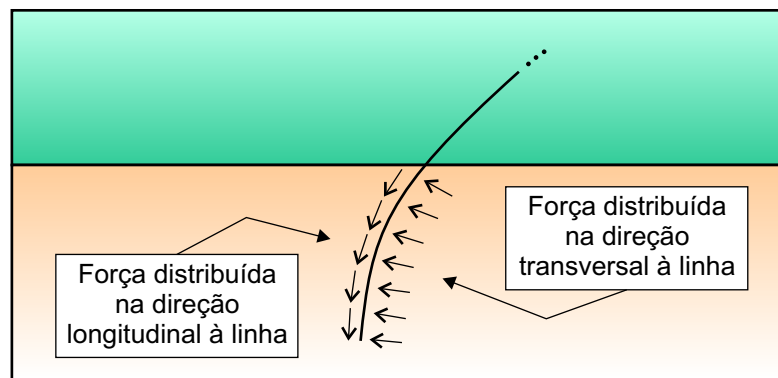


Figura 9: Linha-solo.

É possível ter diversas implementações no *framework* para representar a interação solo-estrutura. Para isso, o usuário instanciador deve criar classes concretas derivadas da classe `Soil`. Para a criação dessas classes é necessário implementar o método `ComputeForce`, pois ele está definido, na classe `Soil`, como virtual puro. Isso significa dizer que as classes derivadas de `Soil` precisam definir obrigatoriamente esse método. Isso pode ser observado na classe `Kelvin`, que é uma classe derivada de `Soil` descrita na próxima seção. O `ComputeForce` calcula a força de reação do solo quando a estrutura toca o fundo do mar. Na Figura 10 é possível observar o diagrama de classe de `Soil`.

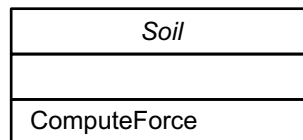


Figura 10: Diagrama de classe de `Soil`.

3.2.3 Classe Kelvin

A classe `Kelvin` representa uma abstração da interação solo-estrutura, considerando um modelo visco-elástico para o solo que é representado por um conjunto de molas e

amortecedores (Figura 11), formando um sistema dinâmico simplificado. A principal função desta classe é referente ao cálculo da força de reação do solo (Silveira, 2001) ou força restauradora do contato (interação solo-estrutura).

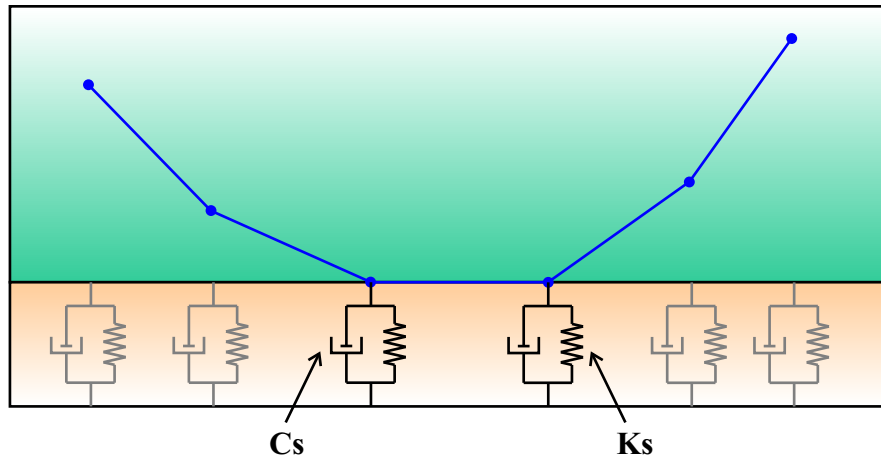


Figura 11: Modelagem do solo marinho como uma base viscoelástica.

A Figura 12 contém o diagrama de classe com os principais atributos e métodos desta classe. Como pode ser visto, os principais atributos são referentes as constantes K_s e C_s que representam a rigidez e o amortecimento do solo, respectivamente. O principal método desta classe é utilizado para o cálculo da força restauradora do contato.

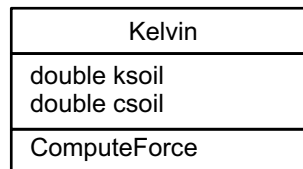


Figura 12: Diagrama de classe de Kelvin.

Para o modelo em questão a força restauradora é obtida através da Equação 3.1

$$F_{cs} = K_s \delta + C_s \nu, \quad (3.1)$$

onde:

K_s = constante elástica do modelo;

δ = profundidade de penetração no solo;

C_s = constante de amortecimento do modelo;

ν = velocidade de penetração no solo.

► **Como associar esta classe ao modelo?**

Primeiramente, é preciso criar uma instância da classe Kelvin, como mostra o trecho de código abaixo (Figura 13). Para isso é necessário definir o valor de `ksoil` e `csoil`. Em seguida, é feita a associação do solo ao modelo, através do método `AddSoil` da classe `Model`.

```
double ksoil; // Constante de rigidez do solo
double csoil; // Constante de amortecimento do solo
...
Soil *soil = new Kelvin(ksoil, csoil);
model->AddSoil(soil);
```

Figura 13: Criação e associação de uma instância de Kelvin.

3.2.4 Classe Current

Trata-se de uma abstração para a modelagem da corrente marítima. Em geral, as informações que se têm relacionadas à corrente marítima são perfis de velocidade de corrente ao longo da profundidade. A partir desses perfis de velocidade, faz-se necessária a transformação dessas velocidades em forças atuantes na linha, para que o efeito da corrente seja incorporado ao modelo. Essa transformação é feita a partir da classe `Current`, que tem a função de calcular a força resultante da interação fluido-estrutura, partindo de um perfil de velocidade de corrente conhecido. Essa força é calculada utilizando a formulação de Morison, dada por:

$$F_m = \frac{1}{2}C_{dt}\rho_w D\dot{u}_{rn}|\dot{u}_{rn}| + \frac{1}{2}C_{dl}\rho_w D\dot{u}_{rt}|\dot{u}_{rt}| + \rho_w \frac{\pi D^2}{4}C_m\ddot{u}_w n - \rho_w \frac{\pi D^2}{4}(C_m - 1)\ddot{u}_{pn} \quad (3.2)$$

onde:

- C_{dt} = coeficiente de arraste transversal;
- ρ_w = densidade do fluido;
- D = diâmetro hidrodinâmico do elemento;
- \dot{u}_{rn} = velocidade do fluido - estrutura normal ao elemento;
- C_{dl} = coeficiente de arraste longitudinal;
- \dot{u}_{rt} = velocidade do fluido - estrutura tangencial ao elemento;
- C_m = coeficiente de inércia;
- \ddot{u}_{wn} = aceleração do fluido na direção normal ao elemento;
- \ddot{u}_{pn} = aceleração da estrutura na direção normal ao elemento.

Nos trabalhos de Silveira (2001) e Pereira (2002) pode ser encontrada essa formulação direcionada ao problema de linhas ancoragem e de *risers*, bem como uma explanação maior sobre o assunto.

No *framework*, o perfil de corrente (classe **Current**) é definido através de uma seqüência de pontos, onde é informado, para cada ponto, a profundidade, a intensidade do vetor velocidade e o ângulo entre o vetor velocidade e o eixo x-positivo, como pode ser visto no diagrama da Figura 14. Além desses dados é necessário informar a densidade da água do mar (ρ_w). O principal método da classe **Current** refere-se ao cálculo da força resultante da interação fluido-estrutura.

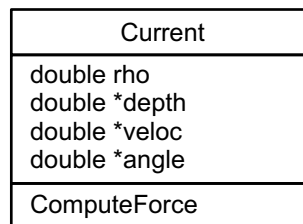


Figura 14: Diagrama de classe de **Current**.

É importante ressaltar que o perfil de corrente é definido sempre ao longo do eixo Z negativo, como ilustra a Figura 15.

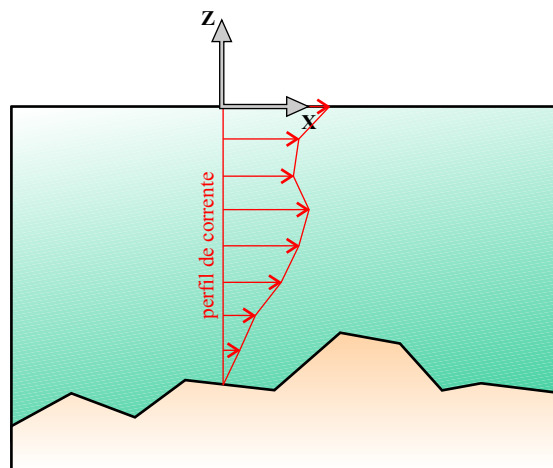


Figura 15: Ilustração do perfil de corrente.

► **Como associar esta classe ao modelo?**

Como pode ser observado no trecho de código da Figura 16, é necessário definir alguns dados para poder criar uma instância de `Current`. O primeiro dado é o número de pontos (`np`) utilizados para construir o perfil de corrente. Os próximos três dados são vetores que armazenam o módulo da velocidade (`veloc`) e a direção (`angle`) dela, em relação ao eixo x-positivo, para cada profundidade (`depth`). É importante mencionar que o perfil de corrente é construído a partir da profundidade nula até a máxima profundidade. Por fim, define-se a densidade da água do mar (`rho`). Então, com os dados todos definidos, o passo seguinte é associar o perfil de corrente (`current`) ao modelo, através do método `AddCurrent` da classe `Model`.

```
int    num;    // Número de pontos
double *depth; // Vetor de profundidades
double *veloc; // Vetor de módulos do vetor velocidade
double *angle; // Vetor de ângulos do vetor velocidade
double rho;    // Densidade da água do mar
...
Current *current = new Current(num, depth, veloc, angle, rho);
model->AddCurrent(current);
```

Figura 16: Criação e associação de uma instância de `Current`.

3.2.5 Classe Load

Esta é a superclasse dos carregamentos que podem atuar pontualmente sobre a estrutura. A principal função da classe `Load` consiste em calcular as forças resultantes

desses carregamentos. Tais carregamentos normalmente são associados a uma função de tempo, caso contrário o carregamento é aplicado instantaneamente à estrutura, atuando como uma carga de impacto.

A classe `Load` possui três subclasses previamente definidas que são as classes `Buoy`, `Clump` e `Force`. Caso o usuário instanciador tenha a necessidade de utilizar outro tipo de carregamento, ele pode fazer isso com relativa facilidade. Para tanto, é necessário criar uma classe derivada de `Load` e em seguida definir o método virtual puro `ComputeLoad` nessa nova classe, semelhante ao que se faz com as classes `Buoy`, `Clump` e `Force`. Na Figura 17 é possível observar os atributos e métodos da classe `Load`, através de um diagrama de classe.

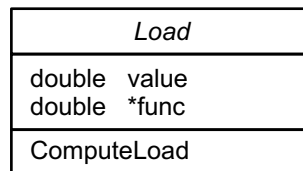


Figura 17: Diagrama de classe de `Load`.

3.2.5.1 Classe `Buoy`

Esta classe abstrai de maneira simples as bóias utilizadas em estruturas *offshore*. A Figura 18 mostra os principais atributos e métodos da classe `Buoy`. Os principais atributos são referentes ao empuxo da bóia, à altura dela, ao peso do pendente e ao comprimento do pendente. O método principal está relacionado ao cálculo da força resultante atuante sobre o nó em que a bóia está fixada.

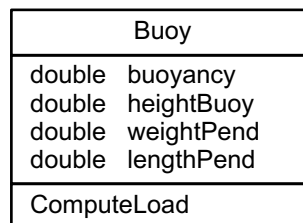


Figura 18: Diagrama de classe de `Buoy`.

Para esse cálculo são consideradas três situações: na primeira, caso a bóia esteja totalmente submersa, é considerado todo o empuxo dela menos o peso do pendente; na segunda, caso ela esteja parcialmente submersa, é considera apenas o empuxo da parte submersa subtraído pelo peso do pendente; na última, caso a bóia não esteja submersa,

não é considerada a ação dela. Na Figura 19 são ilustradas essas três situações, onde a parte submersa da bóia (B) indica o volume que é considerado o empuxo.

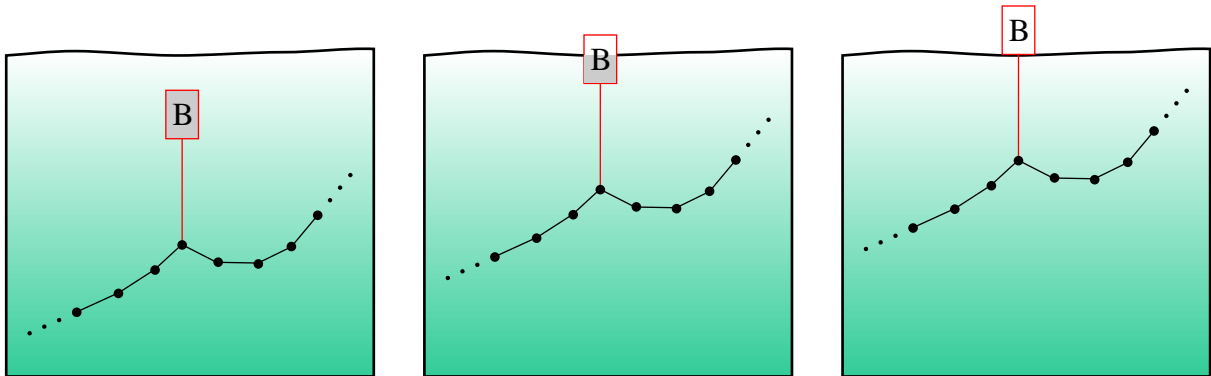


Figura 19: Situações possíveis que a bóia pode se encontrar.

► **Como associar esta classe ao modelo?**

Para se criar uma instância de Buoy faz-se necessário definir alguns parâmetros, como mostra o código da Figura 20. Depois de criada essa instância é feita a associação dela ao modelo, por meio do método *AddLoad* da classe *Model*.

```
double buoyancy; // Empuxo da bóia
double heightbuoy; // Altura da bóia
double weightpend; // Peso do pendente
double lengthpend; // Comprimento do pendente
Function *func; // Função de tempo - opcional
...
Load *buoy = Buoy(buoyancy, heightbuoy, weightpend, lengthpend, func);
Node *node; // Nó que será inserida a bóia
...
model->AddLoad(buoy, node);
```

Figura 20: Criação e associação de uma instância de Buoy.

3.2.5.2 Classe Clump

A classe *Clump* é uma abstração das poitas que podem ser utilizadas nas estruturas *offshore*, como por exemplo na operação de instalação de âncora. Ela tem a função inversa da bóia, ou seja, levar parte da estrutura para uma região mais profunda. Na Figura 21 podem ser vistos os principais atributos e métodos da classe *Clump*. Esses atributos são referentes ao peso e à altura da poita e ao peso do pendente. O método principal refere-se ao cálculo da força resultante atuante sobre o nó em que a poita está fixada.

| Clump | |
|-------------|-------------|
| double | weight |
| double | heightClump |
| double | lengthPend |
| ComputeLoad | |

Figura 21: Diagrama de classe de `Clump`.

Na Figura 22 estão exemplificadas duas situações que podem ocorrer na prática. Na primeira a poita está totalmente suspensa na linha. Nessa situação é considerado todo o peso da poita. Já no segundo caso a poita está apoiada no fundo do mar e, portanto, não é considerado o seu peso.

Para identificar a posição da poita em relação ao fundo do mar é feita uma consulta à classe `Seabed` através do método `GetZCoord`, informando a posição (X,Y) do nó onde a poita está fixada. A consulta retorna a cota Z do fundo, que é comparada à coordenada Z da poita, definindo assim a atuação ou não desta força sobre a estrutura.

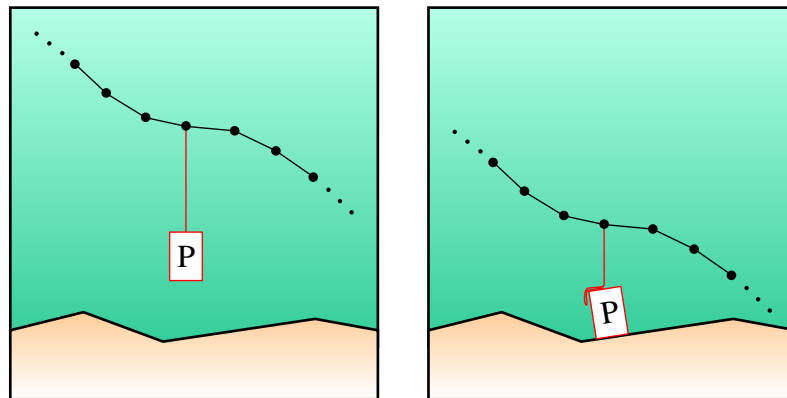


Figura 22: Situações que a poita pode se encontrar.

► ***Como associar esta classe ao modelo?***

Semelhante ao que foi apresentado na classe `Buoy`, deve-se definir alguns parâmetros para se criar uma instância de `Clump`. Em seguida, essa instância é associada ao modelo através do método `AddLoad` da classe `Model`, como mostra o código da Figura 23.

```

double weight; // Peso da poita
double heightclump; // Altura da poita
double lengthpend; // Comprimento do pendente
Function *func; // Função de tempo - opcional
...
Load *clump = Clump(weight, heightclump, lengthpend, func);
Node *node; // Nó que será inserida a poita
...
model->AddLoad(clump, node);

```

Figura 23: Criação e associação de uma instância de `Clump`.

3.2.5.3 Classe Force

Esta classe representa as forças que podem atuar sobre a linha de ancoragem. Tais forças podem ser provenientes de diversos tipos de ações, por exemplo, a ação de uma plataforma sobre a estrutura da linha. Na Figura 24 são mostrados os principais métodos da classe `Force`. O método virtual puro `ComputeLoad` tem a função de calcular a força resultante atuante sobre nó ao qual possui uma força aplicada.

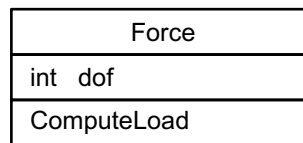


Figura 24: Diagrama de classe de `Force`.

► *Como associar esta classe ao modelo?*

A criação de uma instância de `Force` é feita definindo-se, basicamente, dois parâmetros. O primeiro é a direção da força, através do grau de liberdade. O segundo é o valor da força e do sentido, esse último é definido através do sinal dessa força. Além disso, pode-se associar à força uma função de tempo. Com os parâmetros definidos, cria-se a instância de `Force` e em seguida associa ela ao modelo, com o método `AddLoad` da classe `Model`, tal como mostra o código da Figura 25.

```

int      dof;    // Grau de liberdade
double   value; // Valor da força
Function *func; // Função de tempo - opcional
...
Load *force = Force(dof, value, func);
Node *node; // Nó que será inserida a força
...
model->AddLoad(force, node);

```

Figura 25: Criação e associação de uma instância de *Force*.

3.2.6 Classe *Seabed*

Esta é uma classe abstrata que representa uma abstração à geometria do fundo do mar. A principal funcionalidade desta classe é a obtenção da coordenada *Z* e do vetor normal para cada par ordenado (*X,Y*) dado. Na Figura 26 é mostrado o diagrama de classe de *Seabed*. Os métodos *GetZCoord* e *GetNormal* são os principais métodos dessa classe. O primeiro método é utilizado para consultar a informação referente à coordenada *Z* do fundo do mar, fornecendo um par ordenado (*X,Y*). Já o segundo corresponde ao vetor normal ao fundo do mar, fornecendo também um par ordenado (*X,Y*). Então, como esses métodos são virtuais puros, o usuário instanciador deve definir na classe concreta tais métodos. No próximo tópico, é apresentada uma classe concreta derivada da classe *Seabed*. Assim, é possível observar como os métodos *GetZCoord* e *GetNormal* podem ser implementados.

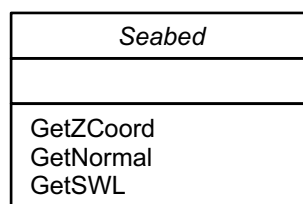


Figura 26: Diagrama da classe *Seabed*.

3.2.6.1 Classe *Plane*

A classe *Plane* representa uma abstração de um fundo do mar plano. Para definir um plano são necessários três pontos não colineares. A partir desses pontos, obtém-se a equação do plano e o vetor normal a ele. Na Figura 27 pode ser visto o diagrama de classe de *Plane*. Os atributos principais referem-se aos coeficientes da equação do plano. Já os métodos principais correspondem à obtenção da coordenada *Z* e do vetor normal.

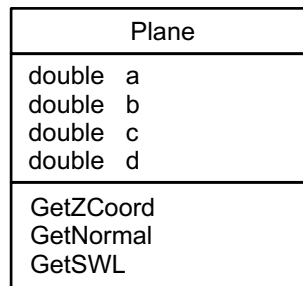


Figura 27: Diagrama de classe de `Plane`.

A equação cartesiana do plano é dada por

$$ax + by + cz + d = 0 \quad (3.3)$$

onde a , b , c e d são coeficientes que definem o plano, obtidos a partir de operações com três pontos não colineares.

Então, utilizando a Equação 3.3, define-se o método *GetZCoord* isolando a variável z dessa equação, obtendo a seguinte expressão:

$$z = \frac{-(ax + by + d)}{c}. \quad (3.4)$$

$$prof = \frac{d}{c} \quad (3.5)$$

Sabe-se da geometria analítica que o vetor normal ao plano é dado por

$$\vec{n} = a\vec{i} + b\vec{j} + c\vec{k}, \quad (3.6)$$

onde a , b e c são os coeficientes da Equação 3.3, definindo-se, assim, o método *GetNormal*.

► ***Como associar esta classe ao modelo?***

A associação da geometria do fundo do mar ao modelo é feita através do método *AddSeabed* da classe `Model`. Contudo, é necessário criar uma instância da classe `Plane`. Para tal, precisa-se definir três pontos não colineares (`p1`, `p2` e `p3`), que determinam um plano no espaço. Esse procedimento é exemplificado com o trecho de código da Figura 28.

```
double p1[3]; // Ponto 1 do plano
double p2[3]; // Ponto 2 do plano
double p3[3]; // Ponto 3 do plano
...
Seabed *plane = new Plane(p1, p2, p3);
model->AddSeabed(plane);
```

Figura 28: Criação e associação de uma instância de `Plane`.

3.2.7 Classe `Element`

Refere-se à classe base dos tipos de elementos finitos. Essa classe tem como principais operações o cálculo da força interna e da matriz de massa, que dependem diretamente do tipo de elemento utilizado. Na implementação atual do *framework* existem duas classes derivadas da classe `Element`, que são: a classe `Truss` e a classe `Beam`.

A classe `Truss` representa o tipo de elemento finito treliça, que possui três graus de liberdade por nó, totalizando seis graus de liberdade por elemento. Detalhes acerca da formulação desse elemento podem ser encontrados em Silveira (2001). Nessa formulação foi implementado modelo constitutivo elástico linear. Contudo, na implementação do *framework*, modifica-se a formulação de forma para permitir a utilização de outros modelos constitutivos, por exemplo, modelos viscoelásticos.

Por sua vez, a classe `Beam` é uma abstração do elemento finito de viga. Esse elemento possui seis graus de liberdade por nó e, portanto, doze graus de liberdade por elemento. No trabalho de Lages *et al.* (1999), encontra-se a formulação para esse elemento. O modelo constitutivo do elemento, nessa formulação, é considerado elástico linear.

A discretização do modelo empregando o método dos elementos finitos utiliza funções de interpolação para gerar um campo de deslocamento sobre um determinado elemento com base nos deslocamentos nodais sofridos por esse elemento. Normalmente, essas funções também são utilizadas para determinar o vetor de carregamento nodal e as matrizes de massa e de rigidez do elemento. Portanto, quando é feito esse procedimento, diz-se que o vetor de carregamento, a matriz de massa e a matriz de rigidez são consistentes.

Contudo, vem sendo utilizado há muitos anos o chamado modelo de massa concentrada (Figura 29), também conhecido como *lumped mass model* (Cook *et al.*, 1989). Quando se utiliza o modelo de massa concentrada a matriz de massa torna-se diagonal. Cada elemento dessa diagonal corresponde à soma dos elementos de cada linha da matriz de

massa consistente. Com isso é simplificada a solução do sistema de equações, por exemplo, viabilizando computacionalmente a utilização de um algoritmo explícito.

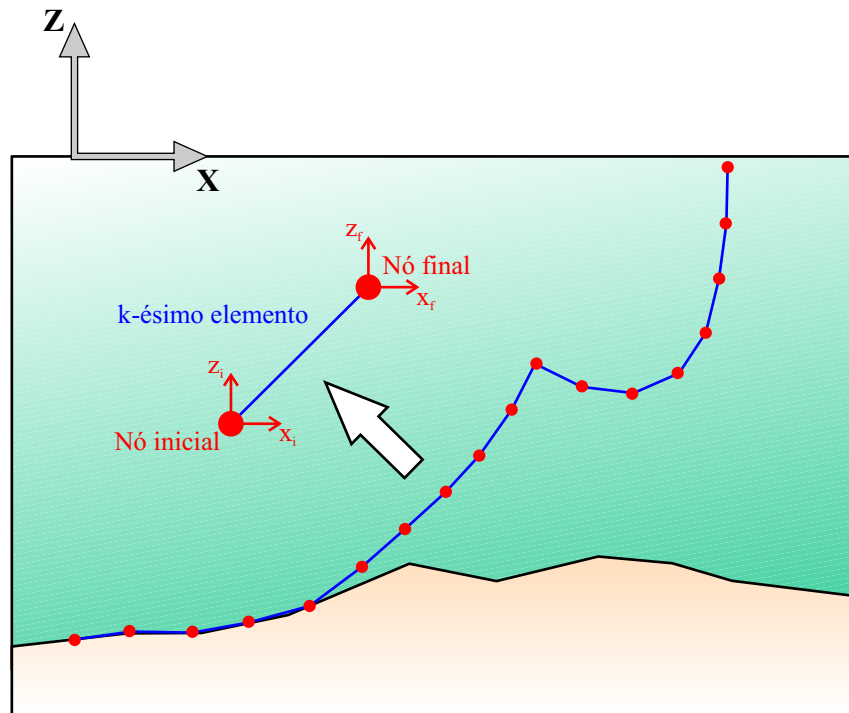


Figura 29: Modelo de massa concentrada.

► ***Como associar esta classe ao modelo?***

No trecho de código da Figura 30, observa-se a construção de instância de `Truss` e de `Beam`. Para tal é necessário definir uma série de dados, que estão discriminados no código abaixo. No caso da instância de `Truss` é preciso definir um material (uma instância da classe `Material`, que é apresentada no próximo tópico. Após a criação dessas instância o passo seguinte é associa-las ao modelo, utilizando o método `AddElement` da classe `Model`.

```

int      id;    // Identificador do elemento
Node     *ni;   // Nó inicial do elemento
Node     *nf;   // Nó final do elemento
Material *mat;  // Material do elemento para o tipo TRUSS
double   EA;   // Rigidez axial do elemento para o tipo BEAM
double   GA;   // Rigidez axial do elemento para o tipo BEAM
double   GJ;   // Rigidez axial do elemento para o tipo BEAM
double   EI;   // Rigidez à flexão do elemento para o tipo BEAM
double   rhoI; // Rigidez à flexão do elemento para o tipo BEAM
double   rhoJ; // Rigidez à flexão do elemento para o tipo BEAM
double   Lo;   // Comprimento inicial do elemento
double   Dh;   // Diâmetro hidrodinâmico do elemento
double   Ws;   // Peso na água do elemento
double   Wa;   // Peso no ar do elemento
double   Cd;   // Coeficiente de arrasto de Morison do elemento
double   Cm;   // Coeficiente de inércia de Morison do elemento
double   Fs;   // Coeficiente de atrito estático do elemento
double   Fd;   // Coeficiente de atrito dinâmico do elemento
...
// Para o elemento do tipo TRUSS
Element *truss = new Truss(id, ni, nf, mat,
                           Lo, Dh, Ws, Wa, Cd, Cm, Fs, Fd);
model->AddElement(truss);

// Para o elemento do tipo BEAM
Element *beam = new Beam(id, ni, nf, EA, EI, GA, GJ, rhoI, rhoJ,
                          Lo, Dh, Ws, Wa, Cd, Cm, Fs, Fd);
model->AddElement(beam);

```

Figura 30: Criação e associação de uma instância de Element.

3.2.8 Classe Material

Esta é a classe abstrata que representa os modelos constitutivos. Esses modelos constitutivos tentam representar o comportamento físico de determinados materiais. Na Figura 31 pode ser visto o diagrama de classe, mostrando que a classe `Material` tem como objetivo calcular o esforço interno no elemento.

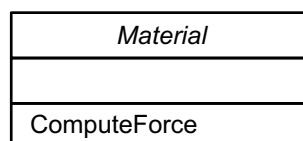


Figura 31: Diagrama de classe de Material.

No *framework* é possível implementar diversos modelos constitutivos. O usuário

instanciador pode criar tantos modelos quanto queira, sendo necessário, apenas, derivar classes de `Material` e definir o método `ComputeForce`, pois esse método é virtual puro. Nos próximos dois tópicos são apresentados alguns modelos constitutivos.

3.2.8.1 Classe Hooke

Esta classe é uma abstração do modelo constitutivo de Hooke. Esse modelo, que tem um comportamento elástico linear, é composto por apenas uma mola sendo, portanto, bastante simples a equação que representa tal modelo reológico, como pode ser visto na seguinte expressão:

$$\sigma(t) = E\varepsilon(t), \quad (3.7)$$

onde:

$$\begin{aligned} \sigma &= \text{tensão axial;} \\ E &= \text{parâmetro elástico;} \\ \varepsilon &= \text{deformação axial.} \end{aligned}$$

O principal atributo e método da classe `Hooke` é mostrado no diagrama da Figura 32. O método `ComputeForce`, que calcula o esforço interna no elemento, é implementado com base na Equação 3.7.

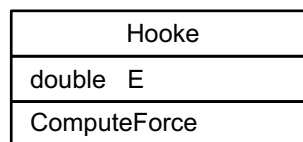


Figura 32: Diagrama de classe de Hooke.

► **Como associar esta classe ao elemento?**

No final do tópico sobre a classe `Element`, mostrou-se a criação de elemento do tipo treliça. Para a criação desse tipo de elemento, faz-se necessário uma instância da classe `Material`. Então, no trecho de código da Figura 33 está sendo mostrada a criação de uma instância de `Hooke` e em seguida a associação dela ao elemento do tipo treliça.

```
double EA; // Parâmetro de rigidez da mola
...
Material *hooke = new Hooke(EA);
Element *truss = new Truss(..., hooke,...);
```

Figura 33: Criação e associação de uma instância de Hooke.

3.2.8.2 Classe Zener

Esta classe abstrai o modelo reológico de Zener, que é um modelo que pode ser utilizado, no escopo deste trabalho, para representar matematicamente o comportamento do poliéster. O modelo de Zener tem um comportamento viscoelástico linear, com duas molas e um amortecedor, tal como mostra a Figura 34.

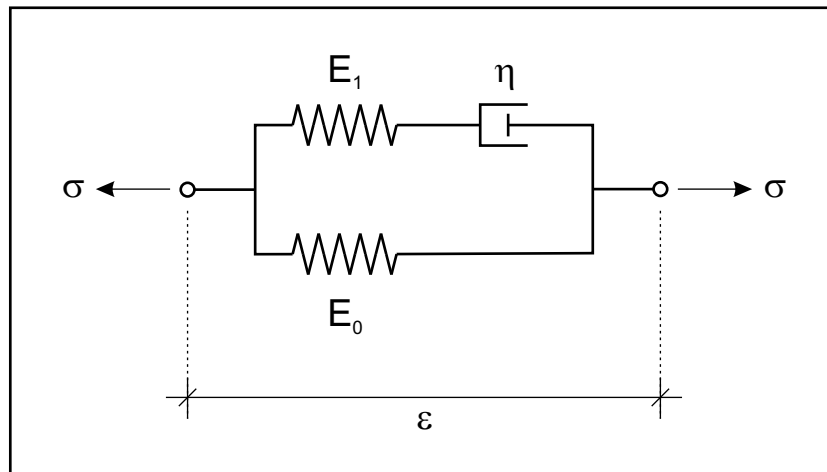


Figura 34: Modelo constitutivo de Zener.

A equação diferencial que representa esse modelo é definida pela seguinte expressão:

$$\frac{\dot{\sigma}(t)}{E_1} + \frac{\sigma(t)}{\eta} = \left(\frac{E_0}{E_1} + 1 \right) \dot{\varepsilon}(t) + \frac{E_0 \varepsilon(t)}{\eta}, \quad (3.8)$$

onde:

$$\begin{aligned} \sigma &= \text{tensão axial;} \\ E_0, E_1 &= \text{parâmetros elásticos;} \\ \eta, &= \text{parâmetro viscoso;} \\ \varepsilon &= \text{deformação axial.} \end{aligned}$$

A Figura 35 mostra o diagrama da classe **Zener** com seus atributos e método. Os atributos principais referem-se aos parâmetros do modelo e ao incremento de tempo da

simulação. Já o principal método (*ComputeForce*) está relacionado ao cálculo do esforço interno no elemento, que é feito tomando como base a Equação 3.8. Nessa equação, a derivada da tensão ($\dot{\sigma}$) é aproximada pelo Método das Diferenças Finitas. Desta forma tem-se:

$$\dot{\sigma}(t) = \frac{\sigma(t) - \sigma(t - \Delta t)}{\Delta t}, \quad (3.9)$$

onde:

σ = tensão axial;

Δt = incremento de tempo.

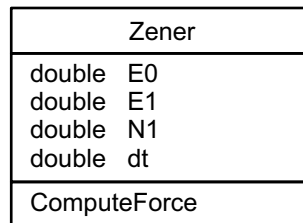


Figura 35: Diagrama de classe de *Zener*.

► *Como associar esta classe ao elemento?*

A criação de uma instância de *Zener* assemelha-se bastante a criação de uma instância de *Hooke*, sendo diferenciada, apenas, nos dados necessários para se construir cada uma delas. Isso pode ser comprovado observando o trecho de código da Figura 36.

```
double EA0; // Parâmetro de rigidez da mola 0
double EA1; // Parâmetro de rigidez da mola 1
double NA; // Parâmetro de amortecimento do amortecedor
double dt; // Incremento de tempo
...
Material *zener = new Zener(EA0, EA1, NA, dt);
Element *truss = new Truss(..., zener,...);
```

Figura 36: Criação e associação de uma instância de *Zener*.

3.2.9 Classe Node

Esta classe representa uma abstração dos nós da discretização do modelo. A principal característica da classe *Node* é o armazenamento de informações associadas à geometria, ou seja, é guardar a posição inicial do nó e a posição no instante de tempo corrente. Na Figura 37 estão apresentados os principais atributos e métodos dessa classe.

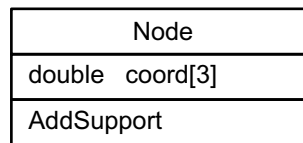


Figura 37: Diagrama de classe de Node.

► **Como associar esta classe ao modelo?**

Para criar uma instância de `Node` faz-se necessário, basicamente, das coordenadas espaciais dele. Com isso definido, cria-se uma instância de `Node` e a associa ao modelo, através do método `AddNode`. É mostrado a seguir o trecho de código (Figura 38) referente ao descrito acima.

```
int    id;        // Identificador
double coord[3]; // Coordenadas
...
Node *node = new Node(id, coord);
model->AddNode(node);
```

Figura 38: Criação e associação de uma instância de Node.

3.2.10 Classe Support

A `Support` representa as condições de suporte (restrições) dos nós. Além disso, ela associa valores prescritos aos graus de liberdades com restrição. Na Figura 39 é mostrado o diagrama de classe de `Support`, com seus atributos e métodos.

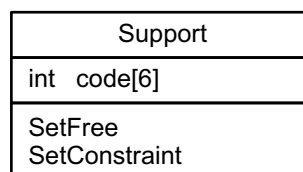


Figura 39: Diagrama de classe de Support.

► **Como associar esta classe ao nó?**

A construção de uma instância de `Support` é feita com a criação da instância sem parâmetros. Depois disso, utiliza-se os métodos `SetFree` e `SetConstraint` para liberar ou restringir o grau de liberdade, respectivamente. Com o método `SetConstraint` é possível associar um valor prescrito. Com a instância de `Support` construída, o passo seguinte é associa-la ao nó, tal com mostra o trecho de código da Figura 40.

```

int      dof;    // Grau de liberdade
Prescribed *presc; // Valor prescrito - opcional
...
Support *support = new Support();
support->SetFree(dof);
support->SetConstraint(dof, presc);
node->AddSupport(support);

```

Figura 40: Criação e associação de uma instância de `Support`.

3.2.11 Classe `Prescribed`

A classe abstrata `Prescribed` é a superclasse de valores prescritos, representados pelo deslocamento, pela velocidade e pela aceleração. Tais valores podem ser impostos sobre a estrutura e podem estar associados a uma função de tempo. Essa associação é importante para que o valor prescrito não seja aplicado instantaneamente, podendo causar um impacto à estrutura.

Alguns dos atributos e métodos da classe `Prescribed` podem ser visto na Figura 41. O método virtual puro `ComputePresc` calcula o valor prescrito associado a um determinado suporte, que por sua vez está associado a um determinado nó.

No *framework* existem duas classes derivadas de `Prescribed`, a classe `Velocity` e a `Displacement`, que podem ser utilizadas para impor, respectivamente, velocidade e deslocamento prescritos. Essas classes são apresentadas nos dois próximos tópicos.

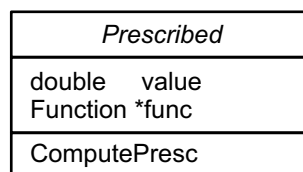


Figura 41: Diagrama de classe de `Prescribed`.

3.2.11.1 Classe `Displacement`

Esta classe representa os deslocamentos prescritos que podem ser impostos à linha de ancoragem ou ao *riser*. A aplicação do deslocamento prescrito no *framework* ocorre em um grau de liberdade restrito, através da classe `Support`. Ainda mais, o deslocamento prescrito pode estar atrelado a uma função de tempo. A Figura 42 apresenta um dos métodos desta classe.

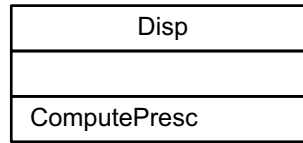


Figura 42: Diagrama de classe de `Displacement`.

O método `ComputePresc` calcula a velocidade (\dot{U}) e a aceleração (\ddot{U}) utilizando o Método das Diferenças Finitas, como mostram estas duas equações:

$$\dot{U}(t) = \frac{U(t + \Delta t) - U(t - \Delta t)}{2\Delta t} \quad (3.10)$$

$$\ddot{U}(t) = \frac{U(t - \Delta t) - 2U(t) + U(t + \Delta t)}{\Delta t^2}, \quad (3.11)$$

onde:

U = deslocamento;

Δt = incremento de tempo.

► ***Como associar esta classe ao suporte?***

A criação de uma instância de `Disp` é relativamente simples. Define-se o valor do deslocamento e, desejando-se, associa-se uma função de tempo. Com isso feito, o passo seguinte é associar a instância de `Disp` ao suporte, utilizando o método `SetConstraint` da classe `Support`. A seguir é apresentado um trecho de código (Figura 43) do procedimento descrito.

```
double value; // Valor do deslocamento prescrito
Function *func; // Função de tempo - opcional
...
Prescribed *disp = Disp(value, func);
...
support->SetConstraint(dof, disp);
```

Figura 43: Criação e associação de uma instância de `Disp`.

3.2.11.2 Classe `Velocity`

A classe `Velocity` está relacionada à aplicação de velocidade prescrita na estrutura (linha de ancoragem ou *riser*). Essa aplicação pode ocorrer nos nós da malha de elementos

finitos, sendo imposta nos graus de liberdades restritos, através da classe `Support`. Além disso, pode-se relacionar a velocidade prescrita a uma função de tempo. O diagrama de classe de `Velocity` pode ser visto na Figura 44.

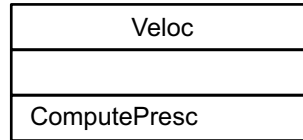


Figura 44: Diagrama de classe de `Velocity`.

O método virtual puro `ComputePresc` faz o cálculo do deslocamento (U) utilizando a seguinte expressão:

$$U(t) = \dot{U}(t)\Delta t, \quad (3.12)$$

onde:

\dot{U} = velocidade;

Δt = incremento de tempo.

► *Como associar esta classe ao suporte?*

Semelhante ao apresentado na classe `Disp`, a criação de um instância de `Veloc` é feita definindo-se o valor da velocidade e, desejando-se, associa-se uma função de tempo. Depois disso, realiza-se a associação dessa instância ao suporte, através do método `SetConstraint` da classe `Support`. No trecho de código apresentado na Figura 45 é possível observar tal procedimento.

```

double value; // Valor da velocidade prescrita
Function *func; // Função de tempo - opcional
...
Prescribed *veloc = Veloc(value, func);
...
support->SetConstraint(dof, veloc);
  
```

Figura 45: Criação e associação de uma instância de `Veloc`.

3.2.12 Classe `Function`

Esta é a superclasse das funções de tempo. Uma das aplicações destas funções é a sua utilização para aplicação gradativa dos carregamentos atuantes sobre a estrutura, evitando

que esses carregamentos sejam impostos como cargas de impacto sobre a estrutura. Por isso, essas funções são também denominadas funções rampa. Essas funções são quase sempre necessárias na criação de instâncias de `Load` e de `Prescribed`. No entanto, se o objetivo for justamente de causar um impacto à estrutura ou julgar não ser necessário a função rampa, simplesmente deve ser omitido o argumento referente a ela, no momento da criação de instâncias de `Load` ou de `Prescribed`. Além disso, as funções de tempo podem ser aplicadas a qualquer entidade no *framework* que varie ao longo do tempo, não sendo específico a essas duas classes. A Figura 46 mostra o diagrama de classe de `Function`. O método virtual puro `ComputeFunc` é utilizado para calcular o valor da função para um determinado instante de tempo.

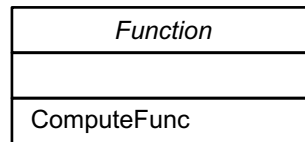


Figura 46: Diagrama de classe de `Function`.

Estão definidas no *framework* três tipos de função de tempo a *piecewise linear*, *step cosine* e *harmonic*, que são explicadas nos próximos tópicos. Contudo, é possível que o usuário instanciador crie outros tipos de funções. Para isso, é preciso derivar uma classe de `Function` e definir nessa nova classe o método `ComputeFunc`.

3.2.12.1 Classe Piecewise

Esta classe representa a função *piecewise linear*, que é uma função definida por trechos lineares, como ilustra a Figura 47.

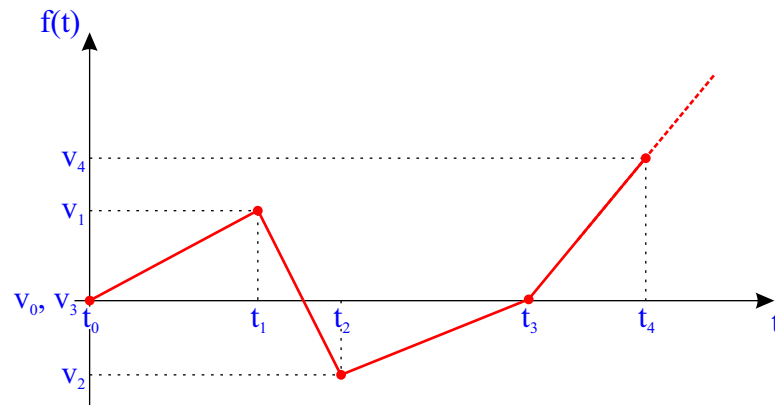


Figura 47: Exemplo da função *piecewise linear*.

O diagrama de classe é apresentado na Figura 48, onde podem ser vistos os atributos

necessários para definir uma função desse tipo, bem como o método utilizado para avaliar a função em um determinado instante de tempo, que é o *ComputeFunc*.

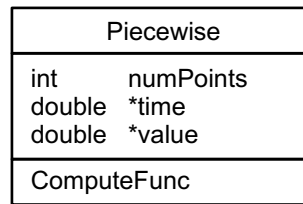


Figura 48: Diagrama de classe de Piecewise.

► **Como associar esta classe ao carregamento ou valor prescrito?**

Antes de explicar a associação de uma instância de *Piecewise*, faz-se necessário explicar como se cria uma instância desse tipo. Para isso, define-se o número de pontos que a função deve ter. Em seguida, preenche-se o vetor *time* (abscissa) e o *value* (ordenada) com os respectivos valores de cada ponto. Assim, constrói-se uma instância de *Piecewise*, podendo ser associada a uma força ou a um deslocamento prescrito, por exemplo, como mostra o trecho de código da Figura 49.

```

int    num;    // Número de pontos
double *time; // Pontos nas abscissas
double *value; // Pontos nas ordenadas
...
Function *piecewise = new Piecewise(num, time, value);

// Associando a uma força
Load *force = new Force();
force->SetData(..., piecewise);

// Associando a um deslocamento prescrito
Prescribed *disp = new Disp();
disp->SetData(..., piecewise);

```

Figura 49: Criação e associação de uma instância de Piecewise.

3.2.12.2 Classe StepCosine

Esta é a classe que representa a função *step cosine*. Essa função é semelhante à função passo unitária, sendo que o passo da função *step cosine* (Figura 50) é feito através de uma função co-seno. Na Figura 50 podem ser observados os parâmetros que são necessários para que a função seja definida.

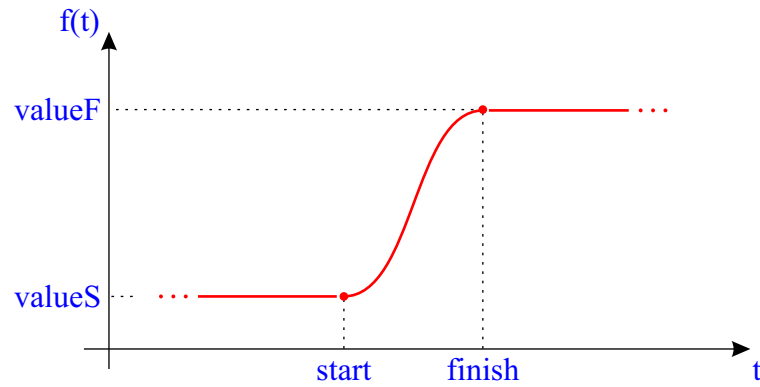


Figura 50: Exemplo da função *step cosine*.

Já na Figura 51 observa-se o diagrama de classe de `StepCosine`. Estão representados nesse diagrama alguns dos atributos e métodos dessa classe. Esses atributos referem-se aos dados necessários para que seja definida a função *step cosine*. Já o método principal realiza o cálculo do valor da função para um instante de tempo qualquer.

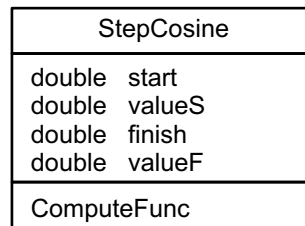


Figura 51: Diagrama de classe de `StepCosine`.

► ***Como associar esta classe ao carregamento ou valor prescrito?***

Como pode ser observado no trecho de código abaixo, a associação da instância de `StepCosine` é semelhante aquela mostrada no tópico anterior, sobre a classe `Piecewise`. Sendo que neste caso a associação é feita a uma bóia e a uma velocidade prescrita. Contudo, antes de associar, deve-se criar a instância de `StepCosine`. Para tal define-se o instante inicial e final e o valor inicial e final da função, como ilustra o código da Figura 52.

```

double start; // Instante inicial
double valueS; // Valor inicial
double finish; // Instante final
double valueF; // valor final
...
Function *stepcosine = new StepCosine(start, valueS, finish, valueF);

// Associando a uma bóia
Load *buoy = new Buoy(..., stepcosine);

// Associando a uma velocidade prescrita
Prescribed *veloc = new Veloc();
veloc->SetData(..., stepcosine);

```

Figura 52: Criação e associação de uma instância de `StepCosine`.

3.2.12.3 Classe Harmonic

Esta classe representa a função *harmonic*, que é uma função cíclica, como ilustra a Figura 53. Os parâmetros que definem essa função são a amplitude (A), a fase (ϕ), o período (T) e o deslocamento vertical (v_0), tal como pode ser observado nessa mesma figura.

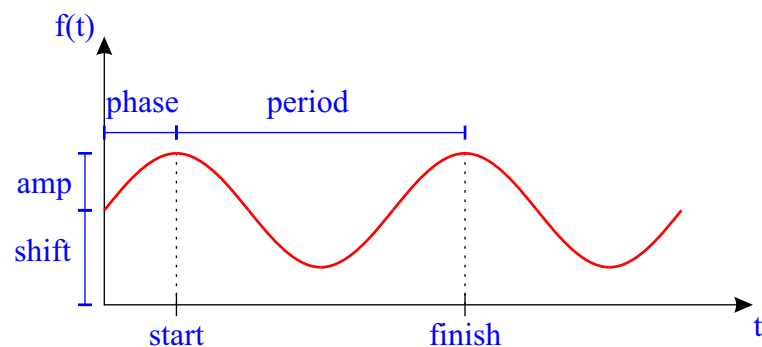


Figura 53: Exemplo da função *harmonic*.

Alguns dos atributos e métodos da classe `Harmonic` estão ilustrados na Figura 54. Os atributos estão relacionados aos dados da função *harmonic* (Figura 53). O método *ComputeFunc* avalia a função *harmonic* para um dado instante de tempo.

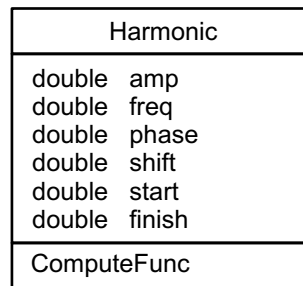


Figura 54: Diagrama de classe de `Harmonic`.

► **Como associar esta classe ao carregamento ou valor prescrito?**

Para a criação de uma instância de `Harmonic` é necessário definir alguns parâmetros como a amplitude, a frequência, a fase e o deslocamento vertical. Além disso, é possível definir o instante de início da função, através do parâmetro `start` e definir um intervalo de tempo, com o uso dos parâmetros `start` e `finish`. Então, com a instância de `Harmonic` criada, o passo seguinte é associá-la a um carregamento ou a um valor prescrito, como exemplifica o trecho de código da Figura 55.

```
double amp;    // Amplitude
double freq;  // Frequência
double phase; // Fase
double shift; // Deslocamento vertical
double start; // Instante inicial - opcional
double finish; // Instante final - opcional
...
Function *harmonic = new Harmonic(amp, freq, phase, shift,
                                  start, finish);

// Associando a uma poita
Load *clump = new Clump(..., harmonic);
```

Figura 55: Criação e associação de uma instância de `Harmonic`.

3.2.13 Classe `IntAlg`

Esta é a superclasse dos algoritmos de integração no tempo. Os algoritmos de integração são utilizados na discretização no tempo das EDOs, que regem o problema tratado. Com a utilização desses algoritmos, são obtidas as respostas do sistema estrutural ao longo do tempo. Já foi dito anteriormente que os algoritmos de integração são classicamente divididos em dois grupos: os algoritmos explícitos e os implícitos.

Portanto, derivam-se da classe `IntAlg` as classes dos algoritmos explícitos (classe `Explicit`) e dos algoritmos implícitos (classe `Implicit`), como ilustra a Figura 6.

Alguns dos atributos e métodos da classe `IntAlg` podem ser vistos na Figura 56. Os atributos principais referem-se ao vetor de forças internas e externas, à matriz de massa diagonal e ao deslocamento, velocidade e aceleração do passo anterior. Um dos métodos principais é o método `ComputeStep`, que realiza a integração em um intervalo de tempo (Δt). Esse método é disponibilizado para aplicação que estiver fazendo uso do *framework*. Essa aplicação deve definir, por exemplo, tempo inicial e tempo final de simulação e fazer chamadas ao método `ComputeStep` da forma que julgar necessária. Dessa forma, o programador da aplicação fica com o controle do processo de simulação. Além do método `ComputeStep`, existe ainda os métodos para consultar de informações em cada intervalo de tempo, tais como: `GetDisp`, `GetVeloc` e `GetAccel`.

| <i>IntAlg</i> | |
|---------------------|--------|
| double | dt |
| double | *fi |
| double | *fe |
| double | *m |
| double | *d0 |
| double | *v0 |
| double | *a0 |
| Model | *model |
| ComputeStep | |
| ComputeFextFintMass | |
| ComputeDispVelAcce | |
| GetDisp | |
| GetVeloc | |
| GetAccel | |

Figura 56: **Diagrama de classe de `IntAlg`.**

A classe `IntAlg` é normalmente a última classe a ser construída. Neste momento, o modelo em estudo já deve ter sido criado a partir de todas as associações entre classes que foram apresentadas. Então, com essa modelagem do problema concebida, chega-se a etapa de análise. Nesta etapa é utilizada uma instância de `IntAlg` para fazer a integração do modelo no tempo, que é realizado através do método `ComputeStep`.

No *framework* estão definido dois algoritmos de integração, que são apresentados nos próximos tópicos. No entanto, caso o usuário instanciador necessite de outro algoritmo de integração, deve-se criar uma classe derivada da classe `Explicit` ou `Implicit`, a depender do tipo do algoritmo, e definir os métodos virtuais puro `ComputeFextFintMass` e `ComputeDispVelAcce` nessa nova classe.

3.2.13.1 Classe ChungLee

Esta classe representa o algoritmo de integração explícito de Chung-Lee (Chung; Lee, 1996). Na Figura 57 pode ser visto o diagrama de classe de **ChungLee**. Os principais atributos dessa classe são relacionados aos parâmetros do próprio algoritmo.

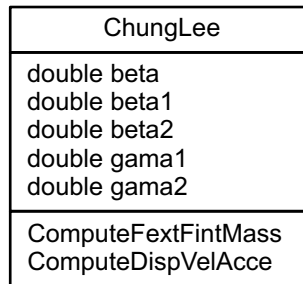


Figura 57: Diagrama de classe de **ChungLee**.

O método virtual puro *ComputeFextFintMass* realiza os cálculos referentes ao item 1 e 2 da parte A do algoritmo (Figura 58). O item 3 dessa parte é calculado no momento da criação de uma instância de **ChungLee**. O método virtual puro *ComputeDispVelAcce* realiza os cálculos da parte B do algoritmo (Figura 58).

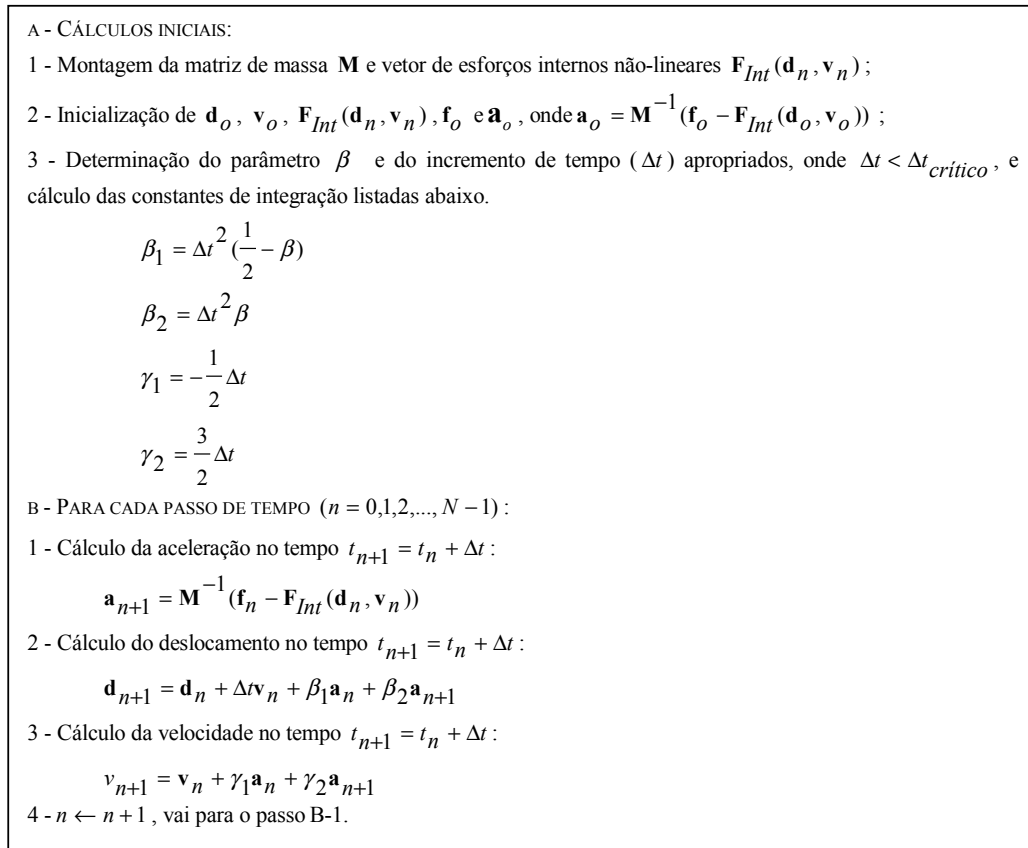


Figura 58: Algoritmo de integração de Chung-Lee para problema não linear (Silveira, 2001).

► **Como associar o modelo a esta classe?**

Com o modelo já definido, o passo seguinte é associa-lo ao algoritmo de integração. Isso é feito no instante da criação da instância de `ChungLee`, como mostra o código da Figura 59.

```
double dt;    // Intervalo de integração
Model *model; // Modelo a ser analisado
...
// Associando o modelo ao algoritmo de integração de Chung-Lee
IntAlg *chunglee = new ChungLee(dt, model);
```

Figura 59: Criação de uma instância de `ChungLee`.

3.2.13.2 Classe MEGAlpha

Esta classe representa o algoritmo de integração de Hulbert-Chung (Hulbert; Chung, 1996), conhecido como Método Explícito Generalizado- α (MEG- α). A Figura 60 mostra

o diagrama de classe de `MEGAlpha`, com seus atributos e métodos. Esses atributos são referentes, principalmente, aos parâmetros do próprio algoritmo.

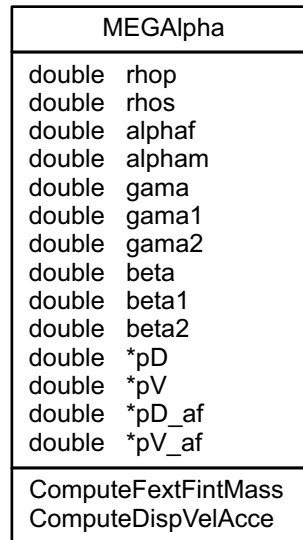


Figura 60: Diagrama de classe de `MEGAlpha`.

O método virtual puro *ComputeFextFintMass* realiza os cálculos dos três primeiros itens do algoritmo que está ilustrado na Figura 61. Já o método virtual puro *ComputeDispVelAcce* realiza os cálculos referentes aos dois últimos itens desse algoritmo (Figura 61).

1 - Predição para os valores dos deslocamentos e das velocidade no tempo t_{n+1} :

$$\mathbf{dp}_{n+1} = \mathbf{d}_n + \Delta t_{n+1} \mathbf{v}_n + \Delta t_{n+1}^2 \left(\frac{1}{2} - \beta \right) \mathbf{a}_n$$

$$\mathbf{vp}_{n+1} = \mathbf{v}_n + \Delta t_{n+1} (1 - \lambda) \mathbf{a}_n$$

2 - Estima valores para os deslocamentos e velocidades em $t_{n+1-\alpha_f}$:

$$\mathbf{d}_{n+1-\alpha_f} = (1 - \alpha_f) \mathbf{dp}_{n+1} + \alpha_f \mathbf{d}_n$$

$$\mathbf{v}_{n+1-\alpha_f} = (1 - \alpha_f) \mathbf{vp}_{n+1} + \alpha_f \mathbf{v}_n$$

2 - Aplica equação de balanço para determinar $\mathbf{a}_{n+1-\alpha_m}$:

$$\mathbf{M} \mathbf{a}_{n+1-\alpha_m} + \mathbf{C} \mathbf{v}_{n+1-\alpha_f} + \mathbf{K} \mathbf{d}_{n+1-\alpha_f} = \mathbf{F}(t_{n+1-\alpha_f})$$

3 - A partir da aceleração $\mathbf{a}_{n+1-\alpha_m}$ determina aceleração \mathbf{a}_{n+1}

$$\mathbf{a}_{n+1} = \frac{\mathbf{a}_{n+1-\alpha_m} - \alpha_m \mathbf{a}_n}{(1 - \alpha_m)}$$

4 - Obtida a aceleração \mathbf{a}_{n+1} , faz-se a correção para os valores previstos para as velocidades e deslocamentos em t_{n+1}

$$\mathbf{d}_{n+1} = \mathbf{dp}_{n+1} + \beta \Delta t_{n+1}^2 \mathbf{a}_{n+1}$$

$$\mathbf{v}_{n+1} = \mathbf{vp}_{n+1} + \gamma \Delta t_{n+1} \mathbf{a}_{n+1}$$

e os parâmetros $\alpha_m, \alpha_f, \beta$ e λ são dados por:

$$\alpha_m = \frac{2\rho_\infty - 1}{\rho_\infty + 1}, \alpha_f = \frac{\rho_\infty}{\rho_\infty + 1}, \lambda = \frac{1}{2} - \alpha_m + \alpha_f \text{ e } \beta = \frac{1}{4} \left(\frac{1}{2} + \lambda \right)^2$$

Figura 61: Algoritmo de integração de Hulbert-Chung com dissipação numérica ótima (Silveira, 2001).

► **Como associar o modelo a esta classe?**

A associação do modelo construído à instância de **MEGAlpha** é semelhante àquela que foi mostrada para a instância de **ChungLee**, tal como confirma o trecho de código da Figura 62.

```
double dt; // Intervalo de integração
Model *model; // Modelo a ser analisado
...
// Associando o modelo ao algoritmo de integração de Hulbert-Chung
IntAlg *megalpha = new MEGAlpha(dt, model);
```

Figura 62: Criação de uma instância de **MEGAlpha**.

4 *Exemplo e Aplicações*

Este capítulo apresenta um exemplo básico de utilização do *framework* e descreve também a utilização dele em três aplicações (programas computacionais) distintas, que são utilizadas para geração de modelos e para simulação dinâmica de estruturas *offshore*. Deve-se destacar que a utilização do *framework* nessas aplicações foram realizadas por usuários instanciadores distintos, o que valida os resultados do desenvolvimento deste trabalho.

No decorrer deste capítulo é utilizada a terminologia *cliente* e *serviço* para referir-se, respectivamente, à aplicação que faz uso do *framework* e ao método do *framework* que é utilizado pela aplicação.

Dentro desse contexto, a Figura 63 ilustra de forma esquemática a interação de um *cliente* com o *framework*, destacando os tipos de *serviços* disponibilizados pelo *framework*, que devem ser utilizados pelo *cliente* para que haja uma comunicação entre eles. Além disso, é evidenciada a extensão do *framework* pelo *cliente*, que mostra o duplo sentido na comunicação entre o *cliente* e o *framework*. Os serviços disponibilizados pelo *framework* estão relacionados no Apêndice A, onde se faz uma sucinta descrição dos métodos e de seus parâmetros.

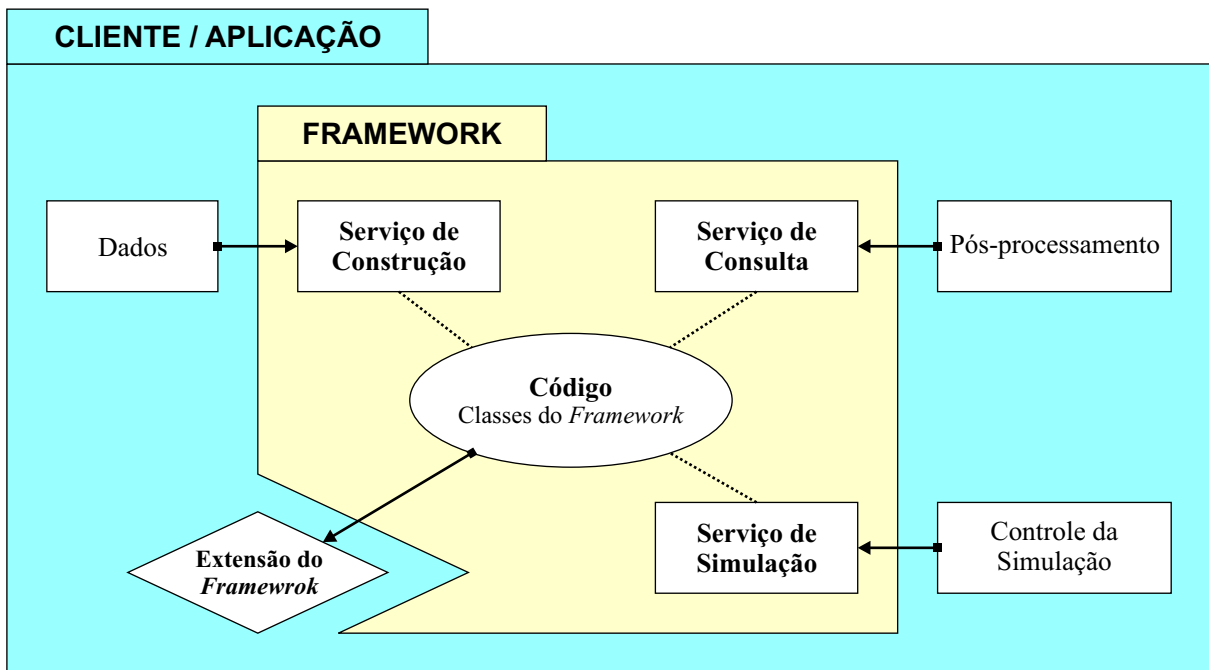


Figura 63: Interação de um *cliente* com o *framework*.

De acordo com o esquema da Figura 63, os dados necessários para o funcionamento do *framework* são fornecidos pelo *cliente* através do *serviço* de construção, que é utilizado para construir o modelo computacional. Esses dados são suficientes para que o *framework* possa realizar a simulação dinâmica de linhas, que é seu maior objetivo.

Após a construção do modelo, o *cliente* deve utilizar o *serviço* de simulação, oferecido pelo *framework*, através do método *ComputeStep* da classe *IntAlg*, para realizar a simulação numérica. Com a utilização desse *serviço*, o *cliente* tem o controle total da simulação, pois esse método analisa apenas um passo da simulação, sendo possível fazer alterações no modelo ao longo da análise.

Durante o processo de simulação ou ao final dele, existem *serviços* de consulta, que permitem ao *cliente* obter resultados parciais ou totais da simulação. Esses *serviços* também estão disponibilizados na classe *IntAlg*, através dos métodos *GetDisp*, *GetVeloc*, *GetAccel* etc. Obtendo esses resultados a partir desses *serviços*, o *cliente* é responsável pela impressão dos resultados em arquivo, pela exibição de uma animação, pelo gráfico de esforços, enfim, pelo pós-processamento dos dados.

Assim, fica evidente que o escopo do *framework* está associado ao processo de simulação dinâmica, eximindo-se da responsabilidade de pré- e pós-processamento dos dados, sendo essa responsabilidade transferida aos *clientes* que utilizam o *framework*.

Considerando os aspectos apresentados acima, opta-se por apresentar um único

exemplo de maneira detalhada e didática, uma vez que a forma de incorporação do *framework* às aplicações que são apresentadas mais adiante, seguem uma metodologia semelhante.

Naturalmente, a metodologia de incorporação do *framework* a essas aplicações segue a mesma idéia apresentada no exemplo, como já foi comentado. Em função disso, são discutidos para cada uma dessas aplicações os seguintes tópicos: breve descrição das características da aplicação, alguns detalhes específicos relacionados à incorporação e vantagens obtidas com a utilização do *framework*.

4.1 Exemplo: *Hello Framework* !

Como o próprio título desta seção sugere, este exemplo tem como objetivo ilustrar de forma didática a relação entre o *cliente* e o *framework* e a utilização, pelo *cliente*, dos *serviços* disponibilizados pelo *framework*. Além disso, apresenta-se a definição dos dados do *cliente*, a criação de instâncias das classes e a forma como essas instâncias se relacionam. É importante destacar que este exemplo não tem caráter genérico no aspecto relacionado ao problema físico, sendo específico ao problema ilustrado na Figura 64.

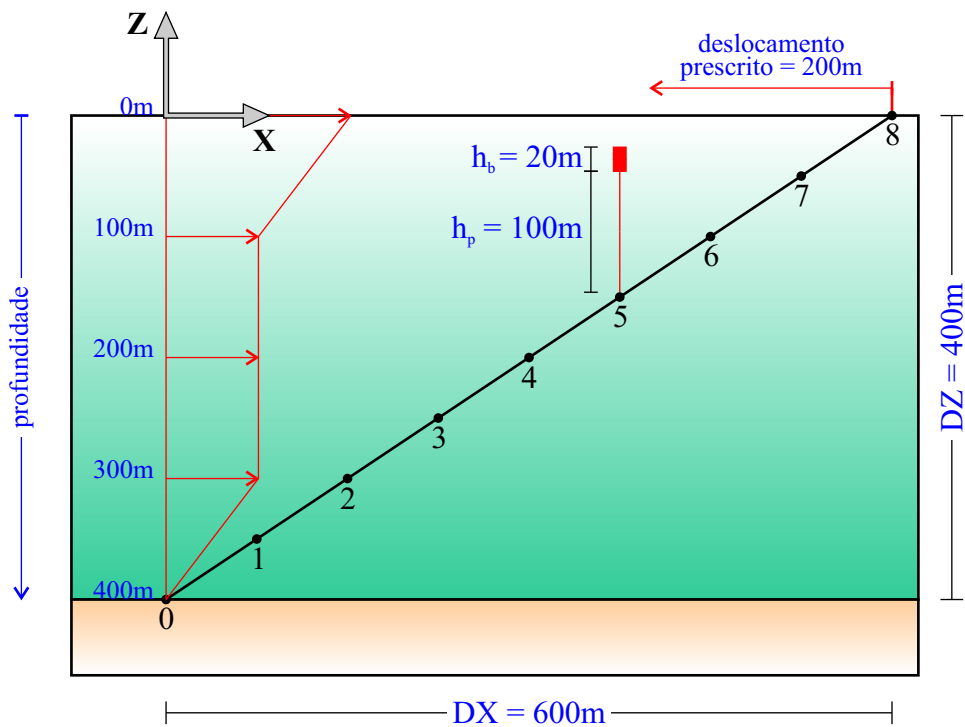


Figura 64: Esquema do problema.

Antes de apresentar a implementação feita para este exemplo, faz-se necessário definir alguns dados que são utilizados para criação das instâncias que irão compor o modelo. Os

dados referentes ao elemento finito, ao perfil de corrente e à bóia são apresentados através das Tabelas 1, 2 e 3, respectivamente. Além dessas tabelas, existem dados definidos na própria figura do problema (Figura 64) e no decorrer do texto.

Tabela 1: Propriedades físicas e geométrica do elemento.

| Propriedade | Símbolo | Unidade | Valor |
|---------------------------------|---------|---------|-----------|
| Comprimento inicial | Lo | m | 90,14 |
| Diâmetro hidrodinâmico | Dh | m | 0,25 |
| Rigidez axial | EA | kN | 300000,00 |
| Peso submerso | Ws | kN/m | 0,50 |
| Peso no ar | Wa | kN/m | 0,50 |
| Coef. de <i>drag</i> de Morison | Cd | - | 1,20 |
| Coef. de inércia de Morison | Cm | - | 2,00 |
| Coef. de atrito estático | Fs | - | 0,70 |
| Coef. de atrito dinâmico | Fd | - | 0,50 |

Tabela 2: Dados do perfil de corrente marítima.

| Profundidade(m) | Velocidade(m/s) | Direção(°) |
|-----------------|-----------------|------------|
| 0,0 | 1,50 | 0,00 |
| 100,0 | 0,75 | 45,00 |
| 200,0 | 0,75 | 45,00 |
| 300,0 | 0,75 | 45,00 |
| 400,0 | 0,00 | 0,00 |

Tabela 3: Dados da bóia.

| Dados | Unidade | Valor |
|-----------------------------------|---------|--------|
| Empuxo da bóia | kN | 200,00 |
| Altura da bóia (h_b) | m | 20,00 |
| Peso do pendente | kN | 5,00 |
| Comprimento do pendente (h_p) | m | 100,00 |

A partir desses dados, pode-se dar início à construção da aplicação que implementa esse problema. Tendo o objetivo de facilitar o entendimento, são construídas funções em C++ (pseudocódigo) para cada item a ser discutido. Na Figura 65 é mostrada a função principal *main* da aplicação implementada, podendo ser observadas as funções que são chamadas na *main*, onde os nomes delas sugerem as tarefas que realizam.

```

void main(void)
{
    Model *model = CriarModelo();

    CriarSolo(model);
    CriarPerfilDeCorrente(model);
    CriarFundoDoMar(model);
    CriarNos(model);
    CriarElementos(model);
    CriarCarregamento(model);
    CriarSuportes(model);

    IntAlg *intalg = CriarAlgoritmoDeIntegracao(model);

    AnalisarModelo(intalg);
}

```

Figura 65: Função *main* da aplicação.

A função *CriarModelo*, como mostra a Figura 66, define o número de nós e de elementos da malha, necessários para se criar uma instância da classe *Model*. É necessária para qualquer aplicação cliente a criação de pelo menos uma instância desta classe para utilização do *framework*, uma vez que ela é responsável pela criação do modelo a ser analisado.

```

Model* CriarModelo(void)
{
    int numNode = 9; // Número de nós
    int numElem = 8; // Número de elementos

    Model *model = new Model(numNode, numElem);

    return model;
}

```

Figura 66: Código da função *CriarModelo*.

Com a instância de *Model* criada, pode-se definir outras instâncias e associá-las imediatamente ao modelo. Como é o caso, por exemplo, do solo marinho, através da função *CriarSolo*. Na Figura 67 é mostrado o código dessa função. Como pode ser observado é criada uma instância de *Kelvin*. Para criá-la é preciso definir as constantes de rigidez e de amortecimento do modelo de *Kelvin*. Em seguida, é feita a associação dessa instância ao modelo utilizando o *serviço* de construção *AddSoil*.

```

void CriarSolo(Model *model)
{
    double ks = 9.0; // Rigidez do solo
    double cs = 27.0; // Amortecimento do solo

    Soil *kelvin = new Kelvin(ks, cs);
    model->AddSoil(kelvin);
}

```

Figura 67: Código da função *CriarSolo*.

Na Figura 68, apresenta-se a função *CriarPerfilDeCorrente*, onde é possível observar a criação da instância de *Current*, mediante a definição de alguns parâmetros, como o número de pontos, o vetor de profundidades, o vetor de velocidades etc. Após criada a instância é feita a associação dela ao modelo por meio do método *AddCurrent* (*serviço* de construção).

```

void CriarPerfilDeCorrente(Model *model)
{
    int    num    = 5;           // Número de pontos do perfil de corrente
    double *depth = new double[num]; // Profundidade
    double *veloc = new double[num]; // Velocidade
    double *angle = new double[num]; // Ângulo que o vetor velocidade forma com o eixo X
    double rho    = 1.025;      // Peso específico da água

    depth[0] = 0.0; veloc[0] = 1.50; angle[0] = 0.0;
    depth[1] = 100.0; veloc[1] = 0.75; angle[1] = 45.0;
    depth[2] = 200.0; veloc[2] = 0.75; angle[2] = 45.0;
    depth[3] = 300.0; veloc[3] = 0.75; angle[3] = 45.0;
    depth[4] = 400.0; veloc[4] = 0.0; angle[4] = 0.0;

    Current *current = new Current(num, depth, veloc, angle, rho);
    model->AddCurrent(current);
}

```

Figura 68: Código da função *CriarPerfilDeCorrente*.

A função *CriarFundoDoMar* define um fundo plano, utilizando, para isso, três pontos no espaço não-colineares. A definição desse plano é feita através da criação da instância de *Plane*. Em seguida, faz-se a associação de tal instância ao modelo utilizando-se o método *AddSeabed* (*serviço* de construção). O procedimento descrito está ilustrado na Figura 69.

```

void CriarFundoDoMar(Model *model)
{
    double p1[3]; // Ponto 1 do plano
    double p2[3]; // Ponto 2 do plano
    double p3[3]; // Ponto 3 do plano

    p1[0] = 0.0; p1[1] = 0.0; p1[2] = -400.0;
    p2[0] = 100.0; p2[1] = 100.0; p2[2] = -400.0;
    p3[0] = -100.0; p3[1] = 100.0; p3[2] = -400.0;

    Seabed *plane = new Plane(p1, p2, p3);
    model->AddSeabed(plane);
}

```

Figura 69: Código da função *CriarFundoDoMar*.

A criação dos nós da malha de elementos finitos é realizada pela função *CriarNos*, tal como mostra a Figura 70. Para criar uma instância de *Node* são necessários, apenas, dois dados: o identificador e a posição espacial do nó. Então, com a instância de *Node* criada é feita a associação dela ao modelo por intermédio do método *AddNode* (*serviço* de construção).

```

void CriarNos(Model *model)
{
    int    numNode = model->GetNode(); // Número de nós
    double DX      = 600.0;           // Projeção horizontal da linha
    double DZ      = 400.0;           // Projeção vertical da linha
    double dx      = DX/(numNode-1); // Projeção horizontal de um elemento
    double dz      = DZ/(numNode-1); // Projeção vertical de um elemento
    double coord[3];                  // Coordenadas espaciais do nó

    for(int i=0; i < numNode; i++)
    {
        coord[0] = dx*i;
        coord[1] = 0.0;
        coord[2] = dz*i - DZ;

        Node *node = new Node(i, coord);
        model->AddNode(node);
    }
}

```

Figura 70: Código da função *CriarNos*.

Concluindo a definição da malha de elementos finitos, tem-se a criação dos elementos e de suas conectividades, como implementa a função *CriarElementos* (Figura 71). Os tipos de elementos disponíveis no *framework* são elementos de treliça (*Truss*) e de viga (*Beam*). Para criar uma instância de **Truss** é necessário uma série de dados, mas é importante destacar alguns deles, como o nó inicial e final e o material do elemento. O nó inicial e final são instâncias de **Node** necessárias para definir a conectividade do elemento. O material é qualquer instância de **Material**, porém, até então, só pode ser associada a uma instância de **Truss**. Com essa instância criada, faz-se a associação dela ao modelo utilizando o método *AddElement* (*serviço* de construção).

```

void CriarElementos(Model *model)
{
    int      id;           // Identificador do elemento
    Node     *ni;         // Nó inicial do elemento
    Node     *nf;         // Nó final do elemento
    Material *mat;        // Material do elemento para o tipo TRUSS
    double   Lo = sqrt(dx*dx + dz*dz); // Comprimento inicial do elemento
    double   Dh = 0.25;   // Diâmetro hidrodinâmico do elemento
    double   Ws = 0.5;    // Peso na água do elemento
    double   Wa = 0.5;    // Peso no ar do elemento
    double   Cd = 1.2;    // Coef. de arrasto de Morison do elemento
    double   Cm = 2.0;    // Coef. de inércia de Morison do elemento
    double   Fs = 0.7;    // Coef. de atrito estático do elemento
    double   Fd = 0.5;    // Coef. de atrito dinâmico do elemento
    double   EA = 300000.0; // Rigidez axial

    // Criado material ///
    mat = new Hooke(EA);
    //////////////////////////////////////

    for(i=0; i < NumElem; i++)
    {
        id = i;

        // Conectividade //////////////////////////////////////
        ni = model->GetNode(id);
        nf = model->GetNode(id+1);
        //////////////////////////////////////

        Element *truss = new Truss(id, ni, nf, mat, Lo, Dh, Ws, Wa, Cd, Cm, Fs, Fd);
        model->AddElement(truss);
    }
}

```

Figura 71: Código da função *CriarElementos*.

O único carregamento concentrado que está sendo considerado no problema em estudo é a bóia. Portanto, a função *CriarCarregamento* contém o processo para associar a bóia ao modelo, como apresenta a Figura 72. Antes da criação da instância de Buoy é criada uma função de tempo que será associada à bóia, tal como mostra a figura. Feito isso e com os dados da bóia definidos, cria-se a instância de Buoy associando-a ao modelo, por meio do método *AddLoad* (*serviço* de construção), que necessita, ainda, do nó ao qual está sendo aplicada a bóia.


```

void CriarCarregamento(Model *model)
{
    double bouyancy = 200.0; // Empuxo da bóia
    double heightBuoy = 20.0; // Altura da bóia
    double weightPend = 5.0; // Peso do pendente
    double lengthPend = 100.0; // Comprimento do pendente
    Function *piecewise; // Função de tempo

    // Criando a função de tempo //////////////////////////////////////
    int num = 3;
    double *time = new double[num];
    double *value = new double[num];
    time[0] = 0.0; value[0] = 0.0;
    time[1] = 15.0; value[1] = 1.0;
    time[2] = 16.0; value[2] = 1.0;
    piecewise = new Piecewise(num, time, value);
    //////////////////////////////////////

    Load *buoy = new Buoy(bouyancy, heightBuoy, weightPend, lengthPend, piecewise);

    Node *node = model->GetNode(5);
    model->AddLoad(buoy, node);
}

```

Figura 72: Código da função *CriarCarregamento*.

Na Figura 73 é mostrada a criação de duas restrições (suportes) associadas aos nós extremos da malha. A criação de uma instância de **Support** é feita sem a necessidade de parâmetros. Contudo, utiliza-se o método *SetConstraint* e o método *SetFree* para definir os graus de liberdades restritos e livres, respectivamente. No caso da função *CriarSuportes* (Figura 73) é utilizado apenas um desses métodos, o *SetConstraint*. Esse método possui como argumento obrigatório o identificador do grau de liberdade, além disso, tem como argumento facultativo uma instância de **Prescribed**. Na criação do segundo nó, como mostra a figura, é feito o uso do argumento facultativo, prescrevendo o deslocamento na direção do eixo X. Esse deslocamento está associado a uma função de tempo, através do uso da instância de **StepCosine**. Finalizando, com a instância de **Support** definida, realiza-se a associação dela ao nó da malha, utilizando, para isso, o método *AddSupport*.

```

void CriarSuportes(Model *model)
{
    Node    *node;
    Support *support;

    // Aplicando o suporte ao primeiro nó da malha ////////////
    support = new Support();
    support->SetConstraint(dof_x);
    support->SetConstraint(dof_y);
    support->SetConstraint(dof_z);

    node = model->GetNode(0);
    node->AddSupport(support);
    //////////////////////////////////////

    // Aplicando o suporte ao último nó da malha ////////////
    double start = 0.0; // Instante inicial
    double valueS = 0.0; // Valor inicial
    double finish = 10.0; // Instante final
    double valueF = 1.0; // valor final
    Function *stepcos = new StepCosine(start, valueS, finish, valueF);
    Prescribed *disp = new Disp(-200, stepcos);

    support = new Support();
    support->SetConstraint(dof_x, disp);
    support->SetConstraint(dof_y);
    support->SetConstraint(dof_z);

    int numNode = model->GetNode();
    node = model->GetNode(numNode-1);
    node->AddSupport(support);
    //////////////////////////////////////
}

```

Figura 73: Código da função *CriarSuportes*.

Neste momento, com o modelo criado, procede-se a etapa de simulação. Contudo, é preciso definir o algoritmo de integração que será utilizado na simulação. Na Figura 74 é mostrado o código da função *CriarAlgoritmoDeIntegracao*, que apresenta a criação de uma instância de *ChungLee*. A criação de tal instância é feita definindo-se o intervalo de tempo de integração e o modelo a ser analisado.

```

IntAlg* CriarAlgoritmoDeIntegracao(Model *model)
{
    double dt = 0.001; // Passo de tempo

    IntAlg *chunglee = new ChungLee(dt, model);

    return chunglee;
}

```

Figura 74: Código da função *CriarAlgoritmoDeIntegracao*.

Por sua vez, a Figura 75 apresenta a implementação da função *AnalisarModelo*, que utiliza o método *ComputeStep* (serviço de simulação) para fazer a integração de um passo de tempo, necessitando, apenas, do tempo corrente da simulação. A integração de um passo de tempo é importante para que o usuário instanciador tenha total controle da análise que está sendo realizada. Dessa forma, ele pode criar estratégias para alterar o modelo em análise e, assim, ter uma maior flexibilidade no uso do *framework*.

```

void AnalisarModelo(IntAlg *intalg)
{
    double T = 100.0; // Tempo total de análise
    double t = 0.0;   // Tempo corrente

    while (t <= T) // Loop no tempo
    {
        intalg->ComputeStep(t); // Calcula um passo de tempo de integração
        t += dt; // Atualiza o tempo corrente
    }
}

```

Figura 75: Código da função *AnalisarModelo*.

Em cada passo da simulação é possível utilizar *serviços* de consulta para fazer o pós-processamento, como por exemplo os métodos *GetDisp*, *GetIntForce* e *GetExtForce* da classe *IntAlg*. Esses métodos obtêm respectivamente o vetor de deslocamento, de força interna e de força externa. Além desses métodos, existem outros métodos utilizados para consulta de dados listados no Apêndice A.

4.2 Aplicação 1: PREA3D

O PREA3D é um pré-processador dotado de uma série de recursos que permitem ao usuário proceder análises quase-estáticas de linhas de ancoragem e *risers*, desenvolvido pelo Tecgraf/PUC-Rio em convênio com o CENPES/PETROBRAS. Esse pré-processador tem por objetivo gerar um modelo completo de uma estrutura *offshore*, contendo todos os dados relacionados ao casco do navio ou plataforma, às linhas de ancoragem e *risers*, às condições ambientais etc. Na Figura 76 é mostrada a janela principal da interface gráfica desse sistema.

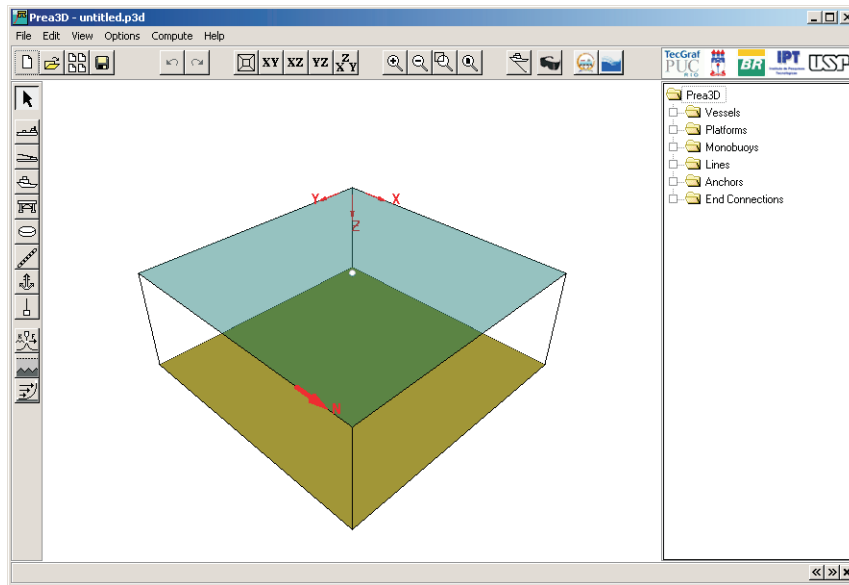


Figura 76: Interface gráfica do PREA3D.

Antes da incorporação do *framework* ao PREA3D, a aplicação limitava-se a análises quase-estáticas de linhas e *risers*. A incorporação do *framework* viabilizou a análise dinâmica dessas estruturas dentro do mesmo ambiente. Para essa incorporação foram necessárias algumas implementações na aplicação *cliente* (PREA3D). A seguir, relacionam-se as implementações realizadas na aplicação em pauta a fim de ilustrar a relação entre o *framework* e uma aplicação cliente:

- **Criar um opção para análise dinâmica de linhas na interface principal da aplicação**

Conforme mencionado anteriormente, a aplicação disponibilizava apenas a possibilidade de análise estática para as linhas do modelo. Com a incorporação do *framework*, a aplicação permite a análise dinâmica de linhas. Para isso, foi criada a opção *Dynamic Equilibrium* no *menu* da linha, como mostra a Figura 77.

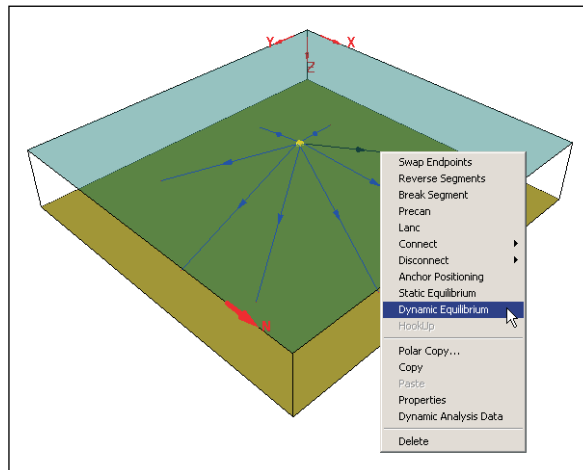


Figura 77: Opção de *Dynamic Equilibrium* da linha ou *riser* selecionado.

- Desenvolver um diálogo (interface gráfica) para obtenção dos dados específicos ao *framework*

Através deste diálogo (Figura 78) o usuário define alguns parâmetros, específicos da análise dinâmica, que até então não faziam parte da estrutura de dados do PREA3D. Destaca-se que a maior parte dos dados necessários já faziam parte da estrutura do PREA3D, como os dados referentes às linhas, ao plano de fundo, às bóias etc. Através desse diálogo o usuário fornece dados específico para a análise dinâmica da linha. Esse diálogo aparece quando o usuário aciona a opção *Dynamic Equilibrium* do *menu* da linha.

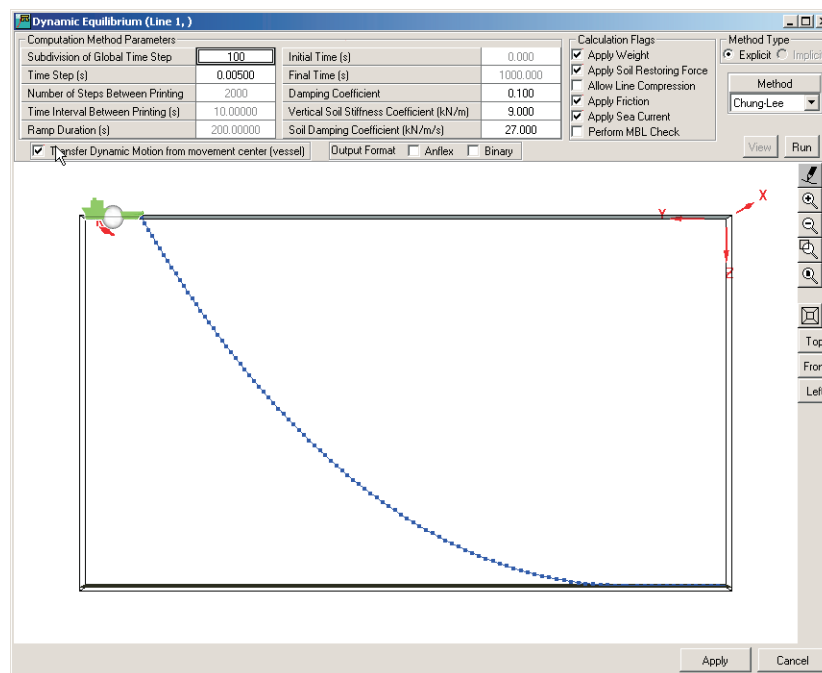


Figura 78: Janela *Dynamic Equilibrium*.

- Criar um mecanismo para definir o modelo do *framework* e para controlar a simulação dinâmica, semelhante ao exemplo apresentado

Quando o usuário do sistema iniciar a simulação, através do botão *run* (Figura 78), é acessado o método *run* da classe *PreadynData*, definida pelo usuário instaciador. Essa classe é utilizada por esse usuário para gerenciar a construção e simulação do *framework*. Em seguida, o método *run* da classe *PreadynData* acessa o método *createData* dessa mesma classe, que constrói o modelo do *framework*. Esse modelo é implementado com dados pré-processados pelo PREA3D e com dados específicos da simulação definidos no diálogo. Com o modelo construído, através do método *createData*, faz-se logo em seguida o controle da simulação utilizando o *serviço* de simulação, disponibilizado no *framework*, por intermédio do método *ComputeStep* da classe *IntAlg*. Na Figura 79 é ilustrado o procedimento descrito acima, através de um pseudocódigo do método *run*, que é acionado pelo botão *run* (Figura 78).

ACESSAR o método *createData* da classe *PreadynData*

CONSTRUIR uma instância da classe *IntAlg*

FAZER um *loop* no tempo

ACESSAR o método *ComputeStep* da classe *IntAlg*

Figura 79: Pseudocódigo do método *run* da classe *PreadynData*.

- Utilizar os *serviços* de consulta de resultados do *framework*

Com a utilização pelo *cliente* dos *serviços* de consulta de resultados é possível fazer o pós-processamento dos dados, através da exibição de gráficos e de uma animação da simulação. Esses *serviços* são utilizados pela aplicação, que neste caso exibe gráficos e uma animação tridimensional da linha a partir da consulta feita a partir desses *serviços*. No caso específico desta aplicação os resultados são exibidos a partir do próprio diálogo da Figura 78.

Dentre as vantagens obtidas com a incorporação do *framework* ao PREA3D, podem ser destacadas:

- Fácil incorporação dos recursos associados à análise dinâmica de linhas;

- Incorporação do *framework* mantendo a independência entre os códigos da aplicação e do *framework*;
- Possibilidade de incorporação automática de novos modelos e algoritmos implementados no *framework*, relacionados à análise dinâmica de linhas, sendo suficiente para a aplicação atualizar a versão do *framework* para obter as novas funcionalidades implementadas.

4.3 Aplicação 2: TPN (Tanque de Provas Numérico)

O Tanque de Provas Numérico (TPN) é um laboratório pioneiro em hidrodinâmica aplicada e fruto de uma colaboração entre a PETROBRAS e algumas instituições de pesquisa do país como a USP, a PUC-Rio, a COPPE/UFRJ e a UFAL.

O TPN é um simulador dinâmico de sistemas *offshore* com unidades flutuantes acopladas às linhas de ancoragem e aos *risers*, onde esse sistema pode estar associado a diversas condições ambientais mesmo em arranjos complexos desse sistema (Nishimoto *et al.*, 2004). Esta aplicação, por ser um programa de simulação, tem características diferentes do PREA3D que é um pré-processador. No entanto, o PREA3D é utilizado como pré-processador para geração dos dados para o TPN. Apesar dessa diferença, o *framework* desenvolvido neste trabalho foi incorporado às duas aplicações, de maneira independente e com fins específicos diferentes.

O TPN foi um dos primeiros sistemas a utilizar a simulação acoplada (casco + linhas). O acoplamento entre as linhas (de ancoragem e *risers*) e o flutuador (navio, plataforma entre outros) é empregado no TPN utilizando a metodologia descrita abaixo, onde para cada passo da simulação:

- A aplicação informa ao módulo de linhas qual a nova posição do navio;
- Essa nova posição é convertida para deslocamento prescrito atuante no topo da linha;
- O módulo de linhas simula o comportamento dinâmico da linha ou do *riser*, deslocando a posição atual do topo da linha até a nova posição do *fairlead*, ponto de conexão no flutuante;
- Ao chegar na nova posição, o módulo de linhas informa à aplicação quais os esforços da linha agindo sobre o flutuante;

- Essa informação de força é utilizada pela aplicação para o cálculo da nova posição do navio.

Esse processo é repetido para cada passo da análise do flutuador até o tempo total requerido de simulação. Na Figura 80 é ilustrado o processo descrito acima.

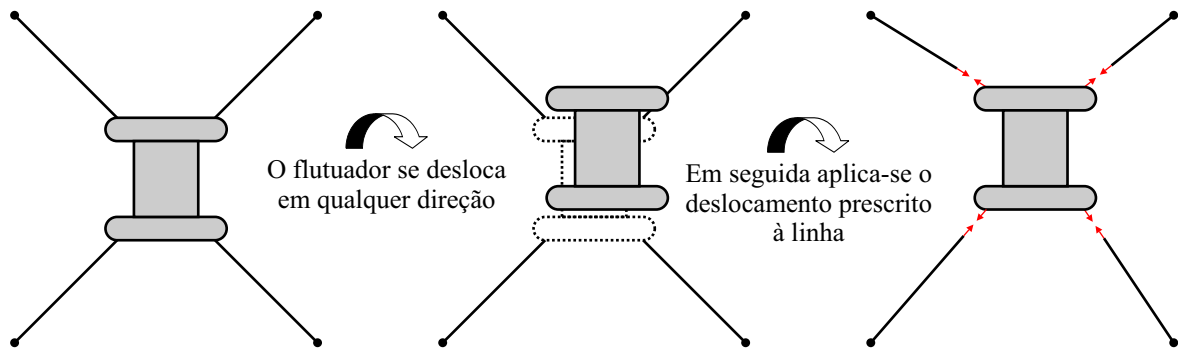


Figura 80: Metodologia da análise acoplada adotada no TPN.

A incorporação do *framework* ao TPN demandou da aplicação as seguintes implementações:

- **Criar um método no *cliente* para definir o modelo do *framework*, utilizando metodologia semelhante ao exemplo**

Para cada linha contida no modelo a ser analisado pelo TPN é criada uma instância da classe *cPreadyn*, que é uma classe definida pelo *cliente*. Então, é construído nessa classe um método para definir o modelo computacional do *framework*, com base nos dados armazenados no TPN.

- **Criar um método para fazer o controle da simulação dinâmica, através do *serviço* de análise, provido pela classe *IntAlg* do *framework***

Quando a simulação inicia-se de fato, realiza-se o procedimento de acoplamento já discutido. Então, o flutuador informa à linha o deslocamento sofrido por ele, através do método *ComputeLineForce* da classe *cPreadyn*. Por sua vez, esse método acessa o método *UpdateDisp* da mesma classe, que converte essa informação em um deslocamento prescrito na extremidade. Em seguida, é feita uma simulação equivalente a um intervalo de tempo da análise do flutuador, utilizando o *serviço* de simulação do *framework*.

ACESSAR o método *UpdateDisp* da classe `cPreadyn`

FAZER um *loop* no tempo (equivalente a um passo de análise do flutuador)
 ACESSAR o *serviço* de simulação, através do método
ComputeStep da classe `IntAlg`

Figura 81: Pseudocódigo do método *ComputeLineForce* da classe `cPreadyn`.

- **Utilizar os *serviços* de consulta de resultados do *framework***

Ao final da simulação de um passo de tempo do flutuador é fechado o ciclo da análise acoplada. Nesse momento, o *cliente* pode utilizar o *serviço* de consulta do *framework* para obter as forças nas extremidades da linha. Além disso, é possível consultar resultados para impressão de arquivos de saída da aplicação.

As vantagens mais relevantes referentes à incorporação do *framework* ao TPN são as mesmas vantagens obtidas com a incorporação do *framework* ao PREA3D: facilidade na incorporação da análise dinâmica de linhas, facilidade na incorporação de novas implementações realizadas sobre o *framework*, independência entre os códigos da aplicação cliente e do *framework*, facilitando sobretudo a manutenção da aplicação.

4.4 Aplicação 3: DYNASIM

O DYNASIM é um simulador de sistemas *offshore*, que considera quase todas as forças hidrodinâmicas e aerodinâmicas que atuam em estruturas oceânicas. Isso inclui, claro, a ação das linhas de ancoragem e dos *risers* nos flutuadores, utilizando, então, a metodologia de acoplamento já discutida na seção anterior.

A PETROBRAS tem utilizado o DYNASIM com grande sucesso já há alguns anos. A construção de uma interface gráfica e a implementação de recursos de visualização permitiu aos usuários a criação de modelos complexos em um curto espaço de tempo. Com esse simulador o usuário pode visualizar, interativamente e em três dimensões, o movimento sincronizado dos flutuadores e ondas bi-direcionais, ventos e correntes marítimas, e, ao mesmo tempo, séries do movimento do flutuador e tensões nas linhas (Coelho *et al.*, 2001).

O DYNASIM é um sistema que incorpora as etapas de pré-processamento, análise e pós-processamento. Um dos componentes mais importantes da implementação do

DYNASIM é o pré-processador, Predyna (Figura 82). O Predyna tem uma metodologia de exibição dos parâmetros de entrada que possibilita uma visualização em três e duas dimensões dos flutuadores, das linhas e das condições ambientais.

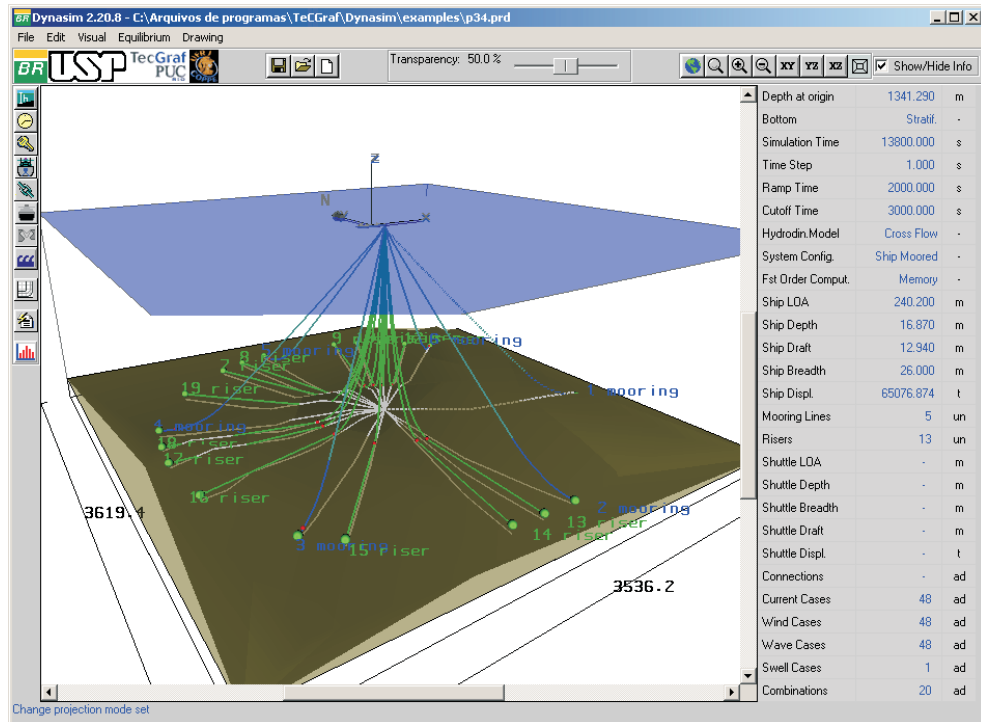


Figura 82: Interface gráfica do Predyna, exibindo um modelo com plataforma e linhas.

Em função das características do DYNASIM, o *framework* desenvolvido foi incorporado aos módulos de pré-processamento e análise dinâmica acoplada (casco + linhas). No entanto, a incorporação do *framework* ao módulo de análise do DYNASIM procedeu-se de forma análoga ao realizado no TPN, seguindo a mesma metodologia. Em função disto, será apresentado aqui apenas detalhes referentes à incorporação do *framework* ao módulo de pré-processamento, o Predyna.

Existe no Predyna a possibilidade do usuário, no momento de criação de uma linha de ancoragem ou *riser*, fazer uma análise estática isolada dessa linha para obtenção da configuração de equilíbrio, com base em alguns parâmetros. Atualmente, essa análise é feita através de um módulo que utiliza as equações da catenária. Entretanto, esse módulo possui restrições para obtenção de algumas configurações de equilíbrio, devido às limitações das equações da catenária.

São exemplos de algumas dessas restrições: o caso em que uma linha inicia a simulação pré-tensionada, neste caso o usuário define a linha de forma que o comprimento dela é menor do que a distância entre a âncora e o *fairlead* (ponto de conexão do flutuador)

(Figura 83(a)); o caso em que o comprimento da linha é maior do que a soma da distância horizontal e vertical entre a âncora e o *fairlead* (Figura 83(b)); e por fim, o caso em que o fundo do mar é estratificado, podendo ou não estar associado a esses dois outros casos.

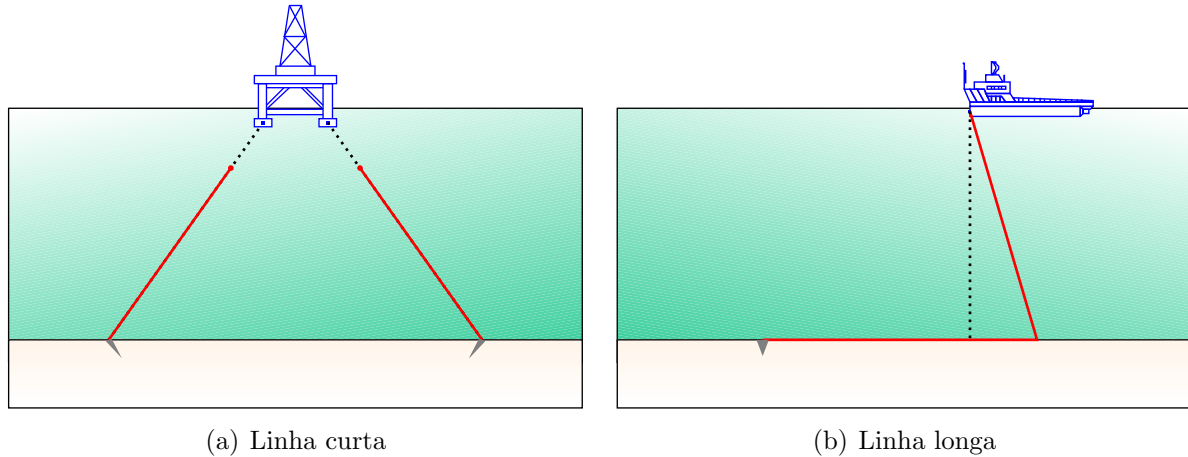


Figura 83: Alguns casos em que não é possível obter a configuração de equilíbrio da linha, com o módulo atual.

No Predyna, para a obtenção da configuração de equilíbrio individual da linha é necessário que seja fornecido pelo menos um dos parâmetros que se seguem: tensão horizontal no topo da linha, tensão total no topo, ângulo no topo ou a distância horizontal entre a âncora e o *fairlead*. Neste caso, o *framework* está sendo utilizado em substituição às equações da catenária para geração das configurações iniciais supracitadas.

O procedimento de incorporação do *framework* ao Predyna segue, parcialmente, a metodologia utilizada nas aplicações apresentadas. Inicialmente é criado um dispositivo para construir o modelo computacional a partir de dados pré-existentes no *cliente*. No entanto, a metodologia implementada na etapa de controle da simulação desta aplicação, diferencia-se das demais. Portanto, apresenta-se a seguir alguns detalhes que diferenciam a metodologia utilizada para esta aplicação.

Considerando um caso em que o fundo do mar é constituído por um plano horizontal e a linha possui um comprimento maior ou igual ao nível da água do mar, a metodologia adotada para a obtenção da configuração de equilíbrio segue os passos abaixo:

- Primeiramente, calcula-se a posição da âncora fictícia, para a qual a linha fica esticada, sem tracioná-la (Figura 86(a));
- Em seguida, gera-se uma malha uniforme ao longo da direção formada pela âncora fictícia e o *fairlead* do flutuador (Figura 86(a));

- Depois disso, aplica-se um deslocamento prescrito, em direção ao *fairlead*, no nó conectado à âncora fictícia, de maneira que será obtida uma linha “relaxada” (Figura 86(b)). Contudo, se o objetivo é obter uma linha “tracionada”, deve-se, então, aplicar o deslocamento prescrito na direção contrária ao *fairlead* do flutuador;
- À medida que esse deslocamento é aplicado verifica-se a condição de parada, que corresponde ao parâmetro de referência definido previamente, descritos abaixo:

– Ângulo no topo

No caso do ângulo no topo da linha ser o parâmetro considerado para obter a configuração de equilíbrio. É implementado no *cliente* um dispositivo para controlar a simulação, fazendo o cálculo do ângulo no topo para cada passo de tempo da análise e verificando se a tensão no topo atual é aproximadamente igual à tensão no topo desejada para a configuração inicial da linha. Caso isso aconteça, a análise é finalizada, uma que vez que o objetivo foi alcançado. No entanto, caso isso não ocorra, é requisitado ao *serviço* de análise do *framework* (método *ComputeStep* da classe *IntAlg*) até que as tensões sejam aproximadamente iguais. Na Figura 84 é mostrado um pseudocódigo que representa isso que foi discutido.

```

ACESSAR o método para calcular o ângulo no topo atual

SE ângulo no topo atual = ângulo no topo desejado

    RETORNAR informando que a análise foi concluída

SENÃO

    ACESSAR o serviço de simulação, através do método
    ComputeStep da classe IntAlg, para dar continuidade ao
    processo de análise dinâmica
  
```

Figura 84: Pseudocódigo do controle da simulação através do ângulo no topo.

– Tensão no topo

Sendo a tensão no topo o critério para obter a configuração de equilíbrio da linha, constrói-se no *cliente* um mecanismo para consultar a tensão no topo atual, através do *serviço* de consulta do *framework*, e para verificar se a tensão no topo atual é igual a tensão no topo desejada, de forma aproximada. Considerando isso verdadeiro, a análise é interrompida, pois foi atingido o

critério de parada. Caso contrário, é acionado o *serviço* de análise do *framework* até que as tensões sejam aproximadamente iguais. Um pseudocódigo desse mecanismo é apresentado na Figura 85.

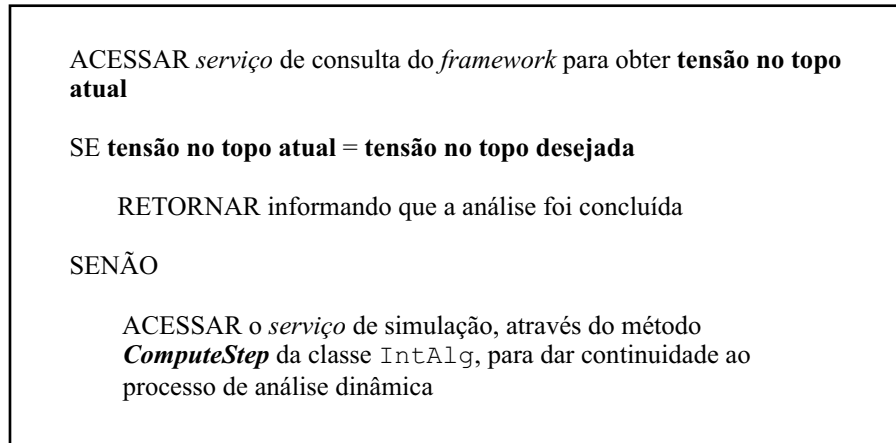


Figura 85: Pseudocódigo do controle da simulação através da tensão no topo.

– Demais parâmetros

Os critérios tensão horizontal no topo e distância horizontal seguem uma metodologia semelhante aos critérios apresentados acima, apenas diferenciando pela verificação da condição de parada de cada um.

- No instante que esse parâmetro for atingido é finalizada a aplicação do deslocamento prescrito, obtendo-se, então, a configuração de equilíbrio da linha.

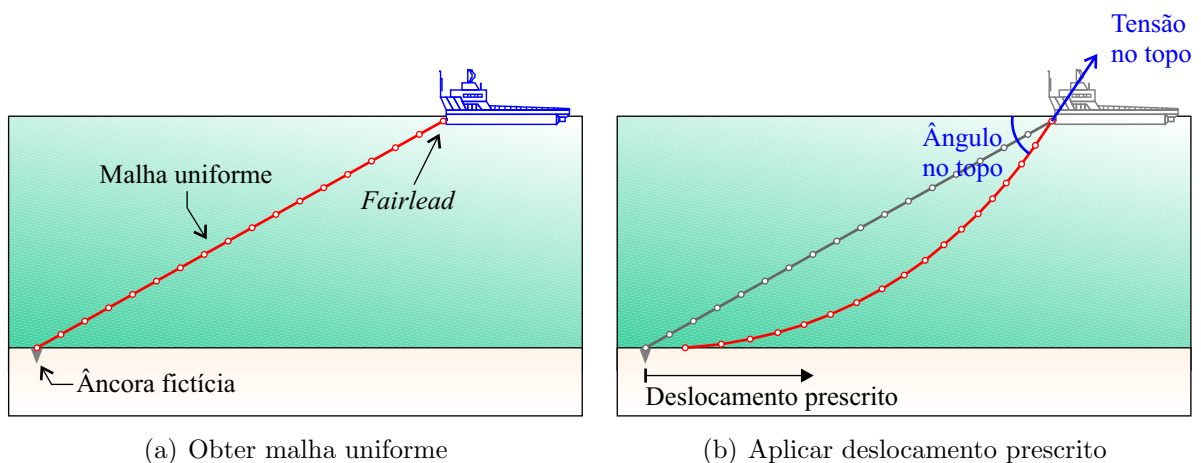


Figura 86: Metodologia adotada no DYNASIM para obter a configuração inicial da linha.

Além de todas as vantagens obtidas com a utilização do *framework* pelas outras aplicações apresentadas, esta aplicação utilizou o *framework* com uma finalidade específica

não convencional, que é a obtenção de configurações iniciais de linhas. Para isso é utilizado todos os *serviços* disponibilizados pelo *framework*. Esta aplicação, além de validar o *framework* desenvolvido, mostra a sua versatilidade quando é utilizado por diferentes aplicações e com fins específicos diferentes.

5 *Considerações Finais*

A crescente demanda feita pela indústria do petróleo por sistemas computacionais complexos que sejam capazes de simular o comportamento físico de estruturas *offshore* e desenvolvimento de pesquisas relacionadas a este tema motivaram o desenvolvimento deste trabalho.

Os sistemas computacionais existentes para análise dinâmica de linhas de ancoragem e de *risers* apresentam algumas limitações no que diz respeito à introdução de novas implementações. Não que a extensão desses códigos não seja possível, mas muitas vezes a incorporação de novas implementações a esses programas demandam além de um grande esforço inicial para o entendimento do seu código, um trabalho muito dispendioso para implementação de novas funcionalidades em função da forma como estão estruturados os códigos computacionais destes sistemas.

O *framework* que foi resultado do desenvolvimento deste trabalho possibilita o aumento da eficiência no desenvolvimento desses sistemas, facilitando seu entendimento, reuso, extensão e manutenção dos códigos (da aplicação cliente e do próprio *framework*), tornando muito mais fácil a implementação de novos modelos numéricos para representação de fenômenos físicos atrelados ao problema, novas metodologias de solução, enfim, novas implementações computacionais no *framework*.

Além dessas facilidades, o programador da aplicação cliente não precisa ter nenhum conhecimento acerca da modelagem numérica relacionada à análise dinâmica de linhas de ancoragem e *risers* sendo simplesmente usuário do *framework* utilizando os serviços oferecidos por eles através dos seus métodos.

A aplicação que utiliza o *framework* herdará automaticamente todas as novas implementações feitas a partir do desenvolvimento de novos projetos, dissertações de mestrado e teses de doutorado etc, uma vez que estes desenvolvimentos utilizarão como plataforma de trabalho o próprio *framework* e caberá às aplicações que o utilizarem simplesmente atualizar a versão do *framework* para incorporar os novos recursos e

funcionalidades implementadas.

5.1 Principais Contribuições

As principais contribuições deste trabalho estão diretamente relacionadas aos objetivos delineados no início desta dissertação tais como:

- **Modelar o *framework* empregando uma linguagem de modelagem gráfica**

Com o uso da linguagem UML (padrão de modelagem gráfica), através do diagrama de classes, foi possível modelar o *framework* com maior facilidade. Com o diagrama de classes construído fica evidenciado a organização das classes, bem como as relações entre elas, facilitando o entendimento global do *framework*;

- **Implementar o *framework* utilizando o paradigma da programação orientada a objetos, através da linguagem C++**

A utilização do paradigma de programação orientada a objetos foi utilizado na implementação do *framework*. As vantagens advindas da utilização deste paradigma estão intrinsecamente relacionadas a alguns dos objetivos principais deste trabalho que são reuso, extensão e manutenção do código. Para isso foi utilizada a linguagem de programação C++ que possibilita à ferramenta computacional desenvolvida uma maior capacidade de modularidade, que facilita na localização do impacto de mudanças na implementação, de reusabilidade, que proporciona melhorias substanciais na produtividade, e de extensibilidade, que possibilita uma personalização de serviços e de características.

- **Validar o *framework* a partir da incorporação em sistemas existentes por usuários distintos**

Um outro aspecto importante, que consolida os resultados obtidos com o desenvolvimento deste trabalho, foi a disponibilização da ferramenta gerada para utilização da mesma por outros sistemas computacionais com o objetivo de validar o *framework*. Com o exemplo e as aplicações apresentadas, destaca-se como aspecto importante que a implementação nas aplicações clientes foram feitas por programadores que são os usuários dos resultados deste trabalho. Isso foi importante para o aprimoramento do *framework* e validação do seu desenvolvimento. Pode-se também destacar que o código das aplicações ficaram, de fato, independentes do

framework. Algumas das melhorias implementadas sobre o *framework* surgiram em função de sugestões e críticas feitas por seus usuários;

- **Contribuir para o melhoramento da modelagem do problema físico, como criação de um novo modelo constitutivo**

Além das contribuições relacionadas a aspectos computacionais, foram dadas algumas contribuições relevantes referentes à modelagem do problema físico. Entre elas estão a inclusão de um modelo viscoelástico linear, que pode ser utilizado para representar o poliéster (material muito utilizado em sistemas de ancoragem), a consideração simplificada de bóia e poita, e a incorporação do elemento finito de viga, que permite uma melhor representação da análise global de *risers*;

- **Tornar o *framework* um sistema computacional que sirva de base para o desenvolvimento de outros sistemas e pesquisas na área de estruturas *offshore***

Como o *framework* foi modelado para ter uma facilidade de extensão de novos recursos, tem-se a partir disso a possibilidade de desenvolver trabalhos de partes específicas do *framework*. Isso pode ser feito sem a necessidade de se conhecer o funcionamento global do *framework*, sendo suficiente conhecer a interface da classe base que será estendida. Com isto, cria-se uma plataforma de desenvolvimento ligada a uma importante linha de pesquisa. Esta plataforma aumentará bastante a produtividade de outros desenvolvedores e pesquisadores que vierem a utilizá-la dando grande impulso a novos desenvolvimentos nesta área;

- **Aplicação imediata do *framework* em análise e projetos de estruturas *offshore* por empresa do setor**

Entre as aplicações que foram apresentadas e que utilizam o *framework* estão o TPN e o DYNASIM que são alguns dos sistemas que vêm sendo utilizados pela PETROBRAS nas etapas de análise e projetos de navio e plataformas da empresa, o que caracteriza uma contribuição dos resultados gerados neste trabalho mesmo antes da sua conclusão.

5.2 Sugestões para Trabalhos Futuros

Apresenta-se nesta seção uma relação de sugestões para trabalhos futuros que podem acrescentar muito a este trabalho, estando alguns dos itens relacionados já em andamento.

- Incorporação de rotinas clássicas de catenária, análise estática e de relaxação dinâmica aumentando sobremaneira a potencialidade do *framework*, facilitando a geração de configurações iniciais complexas;
- Implementação de um algoritmo implícito, onde pode ser utilizada as características de reusabilidade e extensibilidade oferecidas pelo *framework*. Para isso faz-se uma implementação em uma classe derivada da classe `IntAlg`;
- Implementação de um modelo de interação solo estrutura descrito no trabalho de Lages *et al.* (2002), porém é preciso expandir esse modelo para três dimensões, pois o *framework* está em três dimensões. Com isso concluído, deve-se criar uma nova subclasse de `Soil` implementando esse novo modelo em três dimensões;
- Estudar e implementar recursos de computação paralela no *framework* de forma a permitir ao usuário instanciador a utilização em ambientes de computação de alto desempenho. Assim, a execução em paralelo dar-se de forma automática abstraindo essas implementações dos usuários instanciadores;
- Finalmente, elaborar uma documentação técnica, onde seja possível consultar informações sobre qualquer classe e seus respectivos atributos e métodos, além do diagrama de classes para mostrar a relação entre as classe facilitando o entendimento global do *framework*.

Referências Bibliográficas

- BATHE, K. J. M. *Finite Elements Procedures*. New Jersey: Prentice-Hall, 1996.
- BEALL, M. W.; SHEPHARD, M. S. An object-oriented framework for reliable numerical simulations. *Eng. Comput. (Lond.)*, v. 15, n. 1, p. 61–72, 1999.
- BERARD, E. V. Basic object-oriented concepts. Software Engineering Articles. 1998. Disponível em: <<http://www.toa.com/pub/oobasics/oobasics.htm>>. Acesso em: 11 de julho de 2005.
- BETTIG, B. P.; HAN, R. P. S. An object-oriented framework for interactive numerical analysis in a graphical user interface environment. *International Journal for Numerical Methods in Engineering*, v. 39, n. 17, p. 2945–2971, 1999.
- BRAGA, R. T. V. Linguagens de padrões de domínio específico: Como criá-las e utilizá-las na construção e instanciação de frameworks orientados a objetos. Palestra realizada em Fortaleza/CE. Agosto 2004.
- CHUNG, J.; LEE, J. M. A new family of explicit time integration methods for linear and non-linear structural dynamics. *International Journal for Numerical Methods in Engineering*, v. 37, p. 3961–3976, 1996.
- COELHO, L. C. G.; NISHIMOTO, K.; MASETTI, I. Q. Dynamic simulation of anchoring systems using computer graphics. In: OMAE CONFERENCE. Rio de Janeiro, RJ, Brazil: ASME, 2001.
- COOK, R. D.; MALKUS, D. S.; PLESHA, M. E. *Concepts and Applications of Finite Element Analysis*. Madison: John Wiley & Sons, 1989.
- FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Special Issue on Object-Oriented Application Frameworks*, v. 40, n. 10, p. 32–38, 1997.
- FOOTE, B. *Designing to Facilitate Change with Object-Oriented Frameworks*. Dissertação (Mestrado) — University of Illinois at Urbana-Champaign, 1988.
- FUCATU, C. H.; NISHIMOTO, K. Implementação de modelos hidrodinâmicos de navios no sistema prosim. Departamento de Engenharia Naval e Oceânica, USP. 2003.
- GOLDBERG, A. *Smalltalk-80: The Interactive Programming Environment*. Boston: Addison-Wesley, 1984. ISBN 0-201-11372-4.
- GUIMARÃES, J. O. Padrões de projeto e *Frameworks*. Curso realizado em São Carlos/SP. Novembro 2000.

- HANNA, P. Principles of object oriented programming - part 2: Inheritance and polymorphism. Notas de aula da disciplina *Computer Programming II (CSC735)*. 2005. Disponível em: <<http://www.cs.qub.ac.uk/~P.Hanna>>. Acesso em: 11 de julho de 2005.
- HULBERT, G. M.; CHUNG, J. Explicit time integration algorithms for structural dynamics with optimal numerical dissipation. *Computer Methods Applied Mechanics Engineering*, v. 137, p. 175–188, 1996.
- HÜNI, H.; JOHNSON, R.; ENGEL, R. A framework for network protocol software. In: OOPSLA'95 (PROCEEDINGS OF THE 10th ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS). Austin, Texas: ACM Press, 1995.
- JACOB, B. P. *Estratégias Computacionais para Análise Não-Linear Dinâmica de Estruturas Complacentes para Águas Profundas*. Tese (Doutorado) — COPPE/UFRJ, Programa de Engenharia Civil, Rio de Janeiro/RJ, 1990.
- JACOB, B. P.; MASETTI, I. Q. *PROSIM, Simulação Numérica do Comportamento de Unidades Flutuantes Ancoradas*. [S.l.], 1998.
- JÄRNVALL, E.; KOSKIMIES, K.; NIITTYMÄKI, M. Object-oriented language engineering with tale. *Object-Oriented Systems*, n. 2, p. 77–98, 1995.
- JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming*, v. 1, n. 2, p. 23–35, 1988.
- KAROUMI, R. Modeling of cable stayed bridges for analysis of traffic induce vibrations. In: IMAC-XVIII CONFERENCE ON STRUCTURAL DYNAMICS. San Antonio, Texas, 2000.
- LAGES, E. N.; PAULINO, G. H.; MENEZES, I. F. M.; SILVA, R. R. Nonlinear finite element analysis using an object-oriented philosophy - application to beam elements and to the cosserat continuum. *Engineering with Computers*, v. 15, p. 73–89, 1999.
- LAGES, E. N.; RAMOS JR, A. S.; MENEZES, D. P. S.; SILVEIRA, E. S. S.; FERREIRA, F. M. G.; VIEIRA, L. C. L. M.; RAMOS, V. C. L. *Desenvolvimento e Implementações de Modelos Numéricos para Análise da Interação Linha-Solo-Estacas Torpedo e Caracterização do Solo Marinho*. [S.l.], 2002.
- LINTON, M. A.; VLISSIDES, J. M.; CALDER, P. R. Composing user interfaces with interviews. *Computer*, v. 22, n. 2, p. 8–22, 1989.
- MARTHA, L. F.; PARENTE JR., E. An object-oriented framework for finite element programming. In: WCCM V (FIFTH WORLD CONGRESS ON COMPUTATIONAL MECHANICS). Vienna, Austria: H.A. Mang, F.G. Rammerstorfer, J. Eberhardsteiner, 2002.
- MATTSSON, M. *Evolution and Composition of Object-Oriented Frameworks*. Tese (Doutorado) — University of Karlskrona, Ronneby, 2000.
- MCS. 2005. Disponível em: <<http://www.mcs.com/site.php?page=15>>. Acesso em: 02 de novembro de 2005.

MCS. 2005. Disponível em: <<http://www.mcs.com/site.php?page=6>>. Acesso em: 14 de novembro de 2005.

MEYROWITZ, N. Intermedia: The architecture and construction of an object-oriented hypermedia system and application framework. In: OOPSLA'86 (PROCEEDINGS OF THE 1st ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS). Portland, Oregon: ACM Press, 1986.

MOLIN, P.; OHLSSON, L. The points and deviation pattern language of fire alarm systems. In: MARTIN, R.; RIEHLE, D.; BUSCHMANN, F. (Ed.). *Pattern Languages of Program Design 3*. [S.l.]: Addison-Wesley, 1998. p. 431–445.

MOURELLE, M. M.; GONZALEZ, E. C.; JACOB, B. T. Anflex: Computational system for flexible and rigid risers analysis. In: PROCEEDINGS OF THE 9TH INTERNATIONAL SYMPOSIUM ON OFFSHORE ENGINEERING. Rio de Janeiro, RJ, Brazil, 1995.

MOURELLE, M. M.; GONZALEZ, E. C.; SIQUEIRA, M. Q. *Manual Teórico do Programa ANFLEX*. [S.l.], 2001.

NETO, E. R. Uml - unified modeling language. Notas de aula da disciplina: Sistemas de Informação II. 2005. Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos-map/html/uml/index.htm>>. Acesso em: 11 de julho de 2005.

NISHIMOTO, K.; FERREIRA, M. D.; MASETTI, I. Q.; SILVEIRA, E. S. S.; MENEZES, I. F. M.; RUSSO, A. A.; FUCATU, C. H.; TANURI, E. A. Development of numerical offshore tank for ultra deep water oil production systems with multi-bodies. In: CILAMCE'2004 (XXV IBERIAN LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING). Recife, PE, Brazil: UFPE/ABMEC, 2004.

OLIVEIRA, K. R.; PESSOA, M. S. P. Implementando a linguagem uml como uma ferramenta de definição de processo do workflow. In: VIII SIMPEP (SIMPÓSIO DE ENGENHARIA DE PRODUÇÃO). Bauru, São Paulo, 2001.

ORCINA. 2005. Disponível em: <<http://www.orcina.com/index.htm>>. Acesso em: 02 de novembro de 2005.

PEREIRA, J. P. A. *Análise Dinâmica de Cabos para Linhas de Ancoragem*. Monografia (Trabalho de Conclusão de Curso) — Universidade Federal de Alagoas, Alagoas/AL, 2002.

RICARTE, I. L. M. Programação orientada a objetos com c++. Notas de apoio a cursos - FEEC/UNICAMP. 1995. Disponível em: <<http://www.dca.fee.unicamp.br/~ricarte/>>. Acesso em: 11 de julho de 2005.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *UML: Guia do Usuário*. Rio de Janeiro: Campus, 2000. ISBN 8-535-20562-4.

RUSSO, V. F. *An Object-Oriented Operating System*. Tese (Doutorado) — University of Illinois at Urbana-Champaign, 1990.

SAHU, R.; PANTHAKI, M. J.; GERSTLE, W. H. An object-oriented framework for multidisciplinary, multi-physics, computational mechanics. *Eng. Comput. (Lond.)*, v. 15, n. 1, p. 105–125, 1999.

SCHMUCKER, K. J. *Object-Oriented Programming for the Macintosh*. Carmel: Hayden Books, 1986. ISBN 0-810-46565-5.

SILVEIRA, E. S. S. *Análise Dinâmica de Linhas de Ancoragem com Adaptação no Tempo e Subciclagem*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro/RJ, 2001.

VILJAMAA, A. *Pattern-Based Framework Annotation and Adaptation: A systematic approach*. Tese (Doutorado) — University of Helsinki, Helsinki, 2001.

VLISSIDES, J. M.; LINTON, M. A. Unidraw: A framework for building domain-specific graphical editors. In: PROCEEDINGS OF THE ACM USER INTERFACE SOFTWARE AND TECHNOLOGIES. Williamsburg, Virginia: ACM Press, 1989.

WEINAND, A.; GAMMA, E.; MARTY, R. ET++: An object-oriented application framework in c++. In: OOPSLA'88 (PROCEEDINGS OF THE 3rd ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS). San Diego, California: ACM Press, 1988.

APÊNDICE A – Serviços Oferecidos pelo Framework

Neste apêndice são listados alguns métodos do *framework* que são considerados *serviços* por serem utilizados pelo *cliente*. Esses métodos são separados em três categorias de *serviços*: construção, consulta e simulação.

A.1 Serviços de Construção

void Model::AddSeabed(Seabed *seabed)

Descrição:

Adiciona a geometria do fundo do mar ao modelo.

Parâmetros:

seabed – ponteiro para uma instância da classe *Seabed*.

Retorna:

Sem retorno.

void Model::AddSoil(Soil *soil)

Descrição:

Adiciona a modelagem do solo marinho ao modelo.

Parâmetros:

soil – ponteiro para uma instância da classe *Soil*.

Retorna:

Sem retorno.

void Model::AddCurrent(Current *current)

Descrição:

Adiciona a modelagem da corrente marítima ao modelo.

Parâmetros:

current – ponteiro para uma instância da classe **Current**.

Retorna:

Sem retorno.

void Model::AddLoad(Load *load)

Descrição:

Adiciona um carregamento pontual (bóia, poita, força etc) ao modelo.

Parâmetros:

load – ponteiro para uma instância da classe **Load**.

Retorna:

Sem retorno.

void Model::AddNode(Node *node)

Descrição:

Adiciona um nó da malha de elementos finitos ao modelo.

Parâmetros:

node – ponteiro para uma instância da classe **Node**.

Retorna:

Sem retorno.

void Model::AddElement(Element *element)

Descrição:

Adiciona um elemento da malha de elementos finitos ao modelo.

Parâmetros:

element – ponteiro para uma instância da classe **Element**.

Retorna:

Sem retorno.

void Node::AddSupport(Support *support)

Descrição:

Adiciona uma condição de apoio ao nó.

Parâmetros:

support – ponteiro para uma instância da classe **Support**.

Retorna:

Sem retorno.

void Support::SetConstraint(int dof)

Descrição:

Altera a condição do grau de liberdade para restrito.

Parâmetros:

dof – inteiro que identifica o grau de liberdade (varia de 0 a 5).

Retorna:

Sem retorno.

void Support::SetFree(int dof)

Descrição:

Altera a condição do grau de liberdade para livre.

Parâmetros:

dof – inteiro que identifica o grau de liberdade (varia de 0 a 5).

Retorna:

Sem retorno.

A.2 Serviços de Consulta

double* IntAlg::GetDisp(void)

Descrição:

Obtém o vetor de deslocamento nodal da estrutura.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de deslocamento com um número de elementos igual a $6 \cdot (\text{número de nós})$.

double* IntAlg::GetVeloc(void)

Descrição:

Obtém o vetor de velocidade nodal da estrutura.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de velocidade com um número de elementos igual a $6 \cdot (\text{número de nós})$.

double* IntAlg::GetAccel(void)

Descrição:

Obtém o vetor de aceleração nodal da estrutura.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de aceleração com um número de elementos igual a $6 \cdot (\text{número de nós})$.

double* IntAlg::GetIntForce(void)

Descrição:

Obtém o vetor de força interna nodal da estrutura.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de força interna com um número de elementos igual a $6 \cdot (\text{número de nós})$.

double* IntAlg::GetExtForce(void)

Descrição:

Obtém o vetor de força externa nodal da estrutura.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de força externa com um número de elementos igual a $6 \cdot (\text{número de nós})$.

Node* Model::GetNode(int id)

Descrição:

Obtém um nó da estrutura.

Parâmetros:

id – inteiro que identifica o nó.

Retorna:

Ponteiro para uma instância da classe `Node`.

Element* Model::GetElement(int id)

Descrição:

Obtém um elemento da estrutura.

Parâmetros:

id – inteiro que identifica o elemento.

Retorna:

Ponteiro para uma instância da classe `Element`.

double* Node::GetCoord(void)

Descrição:

Obtém as coordenadas iniciais do nó.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de coordenada espacial.

double* Node::GetPosition(void)

Descrição:

Obtém as coordenadas atualizadas do nó.

Parâmetros:

Sem parâmetro.

Retorna:

Vetor de coordenada espacial.

double Element::GetForce(int dof)

Descrição:

Obtém a força interna no elemento na direção grau de liberdade *dof*.

Parâmetros:

dof – inteiro que identifica o grau de liberdade (varia de 0 a 5).

Retorna:

O valor da força interna no elemento.

A.3 Serviço de Simulação

void IntAlg::ComputeStep(double time)

Descrição:

Calcula um passo da simulação dinâmica.

Parâmetros:

time – instante de tempo corrente da simulação.

Retorna:

Sem retorno.