Laboratório de Linguagens de Programação

Impacto da Programação Orientada por Objetos no Projeto de Software de Boa Qualidade.

Apresentação de Resultados finais de POC II

Por:

Alexander Thiago de Assis O. Carvalho

Orientadora:

Profa. Mariza Andrade da Silva Bigonha.

UFMG

Sumário

- Problema.
- Objetivos.
- Metodologia.
- Resultados.
- Conclusões.
- Benchmarks.
- Bibliografia.

Problema:

- Engenharia de Software X Orientação por Objetos no desenvolvimento de software.
  - Uso indiscriminado do paradigma da Orientação por Objetos.
- Identificar, para cada aplicação a ser desenvolvida, a linguagem Orientada por Objetos mais adequada.

Objetivos:

O objetivo deste projeto é apresentar uma análise dos pontos fortes e fracos de diversas linguagens Orientadas por Objetos e sua influência para o Projeto de Software de Qualidade.

Para a avaliação foram considerados alguns dos principais recursos das linguagens Orientadas por Objetos como: **classe, herança, encapsulação, passagem de mensagem e polimorfismo.**

Metodologia 1/3

• Explicitar a relação da Engenharia de Software e da Orientação por Objetos

•Principais fatores externos:

Correção, Robustez, Reusabilidade e Extendibilidade.

• Principais fatores internos: **Confiabilidade e Modularidade.**

• Linguagens Escolhidas.

Eiffel (Várias Refrências)
Smalltalk (Linguagem OO pura)
C++ (Muito usada)
Java (Muito Usada

Metodologia 2/3

- Identificação das Métricas que foram utilizadas juntamente com os aspectos que foram focados.
  - **Número de métodos por classe.**
  - **Profundidade da árvore de herança.**
  - **Número de filhos.**
  - **Acoplamento entre as classes.**
  - **Resposta de uma classe.**
  - **Coesão entre os métodos de uma classe.**
- Estudo detalhado sobre cada métrica
- Avaliação de cada uma das linguagens
  - **Estudo Orientado pela métricas, evitando assim fugir do escopo da OO**

Metodologia 3/3

- Quadro comparativo das linguagens
- Benchmark para mostrar a legibilidade e como conseqüência a extensibilidade do software.
- Conclusão final

Resultados

	Smalltalk		Eiffel	
	P	N	P	N
MPC	• Herança Simples • Classes Abstratas		• Herança Múltipla + Controle de visibilidade: *feature* • Herança Simples	• Herança Múltipla
PAH	• Classes Abstratas	• Herança Simples	• Herança Múltipla • Controle de visibilidade: *feature* • Recurso: *ANY*	
NDF	• Herança Simples		• Controle de visibilidade: *feature* • Recurso: *ANY* • Herança Simples	• Herança Múltipla
AEC	• Herança Simples	• Baseada em Mensagens	• Controle de visibilidade: *feature* • Herança Simples	• Herança Múltipla
RDC	• Herança Simples	• Baseada em Mensagens	• Herança Simples	• Dependente do Implementador
FCM	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador

Fig. 12: Quadro comparativo I

	C++		Java	
	P	N	P	N
MPC	• Herança Simples	• Classe Orfã • Herança Múltipla • Modos de Acesso • Utilização de Código C • Classes Amigas	• Herança Simples • Recurso: *final*	• Interface (simulação de herança múltipla)
PAH	• Herança Múltipla • Classes Amigas	• Classe Orfã • Herança Simples • Modos de Acesso	• Interface (simulação de herança múltipla) • Recurso: *final*	• Herança Simples
NF	• Herança Simples	• Classe Orfã • Herança Múltipla • Modos de Acesso	• Herança Simples • Recurso: *final*	• Interface (simulação de herança múltipla)
AEC	• Herança Simples	• Herança Múltipla • Modo de Acesso • Classes Amigas	• Herança Simples • Recurso: *final* • Controle de Acesso Simples • Recurso: *Pacotes*	• Interface (simulação de herança múltipla)
RDC	• Herança Simples	• Modo de Acesso • Classes Amigas	• Herança Simples • Recurso: *final* • Controle de Acesso Simples • Recurso: *Pacotes*	
FCM	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador

	Smalltalk	ISE Eiffel	C++	Java
Projeto por Contrato	Não possui	Projeto por Contrato	Não possui	Não possui
Verificação de Tipo	Tipada dinamicamente	Tipada estaticamente	Estaticamente tipada, embora a linguagem dê suporte à C-Style o qual permite violação das regras de tipo.	Na maioria das vezes tipada estaticamente, mas tipificação dinâmica é necessária para estruturas de "conteiner" genéricas.
Compilação	Historicamente baseada em interpretação, mas atualmente possui uma mistura de interpretação e compilação.	Combinação de compilação e interpretação no mesmo ambiente	Compilada	Mistura de interpretação e compilação "on-the-fly"
Eficiência do Código Gerado	Executáveis lentos, pois requer "Smalltalk Image"	Gera executáveis rápidos, mas não tão eficientes como C++	Gera executáveis rápidos	Possui muitos problemas de desempenho
Tratamento de Exceção	Possui	Possui	Possui	Possui
Herança	Herança simples	Herança múltipla	Herança múltipla, mas com vários problemas	Herança simples, embora a herança múltipla possa ser simulada através das "interfaces"
Facilidade de Uso	No geral possui uma sintaxe "bizarra"	Clara, simples e sintaxe legível	Sintaxe complexa	Sintaxe parecida com C++
Padronização de estilo de biblioteca	Ênfase na consistência	Bibliotecas com vocabulário padronizado	Ampla variação	?
Gerência de Memória	Possui coletor de lixo	Coletor de lixo/Gerência automática de memória	Não possui coletor de lixo	Possui coletor de lixo
Escopo no ciclo de vida do software	A maior parte está focada na implementação	Ambiente de desenvolvimento visual sem emenda	Somente implementação de endereçamento	Focado na implementação
Software matemático	Bibliotecas disponíveis	Bibliotecas disponíveis	Bibliotecas disponíveis	Bibliotecas disponíveis ou em construção
Interoperabilidade	Sem padronização de interface para a linguagem C	Linguagem e ambiente padronizados para interação com C e C++	Boa interoperabilidade com C	Métodos nativos
Grau de OO	Puramente OO	Puramente OO	Híbrida	Apresenta um alto grau de OO, mas não é puramente OO

UFMG

Conclusões

- **Smalltalk:**
  - Produção de Softwares que simulam o mundo real
  - Não indicada para softwares científicos ou que possuam grandes processamentos.
  - Possui muitos recursos para atingir a qualidade do software
  - Sintaxe complexa
  - Executáveis de pouca eficiência
  - Polimorfismo pouco eficaz.

Conclusões

- **Eiffel:**
  - A melhor de todas as linguagens avaliada.
  - Recursos claros e elegantes.
  - Sintaxe simples e de fácil entendimento.
  - Indicada para qualquer área de aplicação.
  - Executáveis eficientes, não tanto quanto C++.
  - Fácil de ser aprendida por quem está em início de careira
  - Extremamente voltada para atingir a qualidade do Software
  - Possui o melhor recurso de polimorfismo.

Conclusões

- **C++:**
  - Produz os executáveis mais eficientes.
  - A linguagem que menos oferece recursos para se atingir a qualidade.
  - Pode ser usada em qualquer aplicação onde a eficiência é o principal alvo e não a qualidade.
  - Por ser uma linguagem híbrida os objetivos da OO não são tão desenvolvidos.
  - Permite um grau de liberdade muito amplo para o programador. O que torna a linguagem complexa de ser entendida.

Conclusões

- **Java:**
  - Semelhante a C++, mas possui um grau bem maior de OO.
  - Sintaxe simples e de fácil entendimento.
  - Destaca-se pelo seu poderoso recurso de pacotes, o que dá um eficiente suporte à encapsulação.
  - Indicada para aplicações WEB, devido sua portabilidade.

Conclusões

- Nem toda linguagem OO atinge a qualidade.
- Os objetivos finais em se desenvolver um sistema devem ser levantados.
- O tipo de aplicação é importante.
- Bons desenvolvedores aliados com uma linguagem adequada produzem softwares muito melhores.
- Este trabalho serve como um guia e um conjunto de critérios para se determinar qual linguagem OO deve ser usada em um determinado contexto.
- Através desses levantamento, critérios e regras é possível produzir softwares mais eficientes, mais baratos e com um maior grau de qualidade.

Benchmarks

- **Smalltalk:**

```
advance
" first try next row "
        (row < 8)
ifTrue: [ row := row + 1. ↑ self findSolution ].
" cannot go further, move neighbor "
        (neighbor advance) ifFalse: [ ↑ false ].
" begin again in row 1 "
         row := 1.
         ↑ self findSolution
```

…

Benchmarks

- **Eiffel:**

```
class CHESS_BOARD
        inherit
                D2_ARRAY[CHESS_PIECE]
                rename
                        valid_coordinates as is_on_board,
                        item_at as piece_at,
                        put_at as attach
                end
        creation
                create
                        -- Must be initialised as 8 * 8 matrix
        feature
```

Benchmarks

- **C++:**

```
bool queen::canAttack (int testRow, int testColumn)
{
        if (row == testRow)
                return true;
        int columnDifference = testColumn - column;
        if ((row + columnDifference == testRow) ||
                (row - columnDifference == testRow))
                        return true;
        return neighbor && neighbor->canAttack(testRow, testColumn);
}
…
```

Benchmarks

- **Java:**

```
class Queen {
                    // data fields
        private int row;
        private int column;
        private Queen neighbor;
                    // constructor
        Queen (int c, Queen n) {
                            // initialize data fields
                row = 1;
                column = c;
                neighbor = n;
                }
…
```


Bibliografia

- A. A. Kaposi, "*Measurement Theory",* in Software Engineer's Ref. Book, J. McDermind, Ed., Oxford: Butterworth-Heinemann Ltd., 1991.
- M. Bunge, *Treatise on Basic Philosophy: Ontology I: The Furniture of the World.* Boston: Rieldel, 1977.
- M. Bunge, *Treatise on Basic Philosophy: Ontology I: The World of Systems.* Boston: Rieldel, 1979.
- J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, and N, Ballou, *"Data model issues for object oriented applications,"* ACM Trans. Office Inform. Syst., vol. 5, pp. 3-26, 1987.
- G. Booch, *Object Oriented Design with Applications.* Redwood City, CA: Benjamin/Cummings, 1991.
- Meyer, Bertrand. *Object-oriented Software Construction*, Prentice-Hall International Series in Computer Science, C.A.R. Hoare Series Editor, 2nd Edition, 1254 pág., 1997.
- Bigonha, Mariza Andrade da Silva; BIGONHA, Roberto da Silva. *Programação Orientado por Objetos*, transparências de aula. Universidade Federal de Minas Gerais, departamento de Ciência da Computação. Belo Horizonte, 2003.
- BIGONHA, Mariza Andrade da Silva; BIGONHA, Roberto da Silva. *Linguagens de Programação*, transparências de aula. Universidade Federal de Minas Gerais, departamento de Ciência da Computação. Belo Horizonte, 2003.
- Chidamber, Shyam and Kemerer, Chris, "*A Metrics Suite for Object Oriented Design*", IEEE Transactions on Software Eng, June, 1994, pp. 476-492.
- Filho, Wilson de Pádua Paula. *Engenharia de Software Fundamentos, Métodos e Padrões.* São Paulo, 2ª edição, 2003.
- Gudwin, Ricardo R., "*Notas de aula para a disciplina EA877*", Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e Computação, 1997.
- Disponível em: < http://www.aerohobby.hpg.ig.com.br/Hobbies/1/ling_prog.pdf>.
- R. E. Prather, "An axiomatic theory of software complexity measures." *Comput. J.*, vol. 27, pp 340-346. 1984.
- F. Roberts, *Encyclopedia of Mathematical and its Applications.* Reading, MA: Addison-Wesley, 1979.
- Rosenberg, Linda H, *Applying and Interpreting Object Oriented Metrics.* Utah, April 1998. Disponível em: http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html
- Sebesta, Robert W., *Linguagens de Programação*, Trad. José Carlos Barbosa dos Santos, 2000. Bookman, 4a. edição, 1999.
- I. Vessey and R. Weber, "*Research on structured programming: An empiricist's evaluation,*" IEEE Transactions on Software Eng., vol. SE-10, pp 394-407, 1984.
- E. Weyuker, "Evaluating software complexity measures.", *IEEE Transactions Software Engineering.,* vol. 14, pp 1357-1365, 1988.
- Watt, David. *Programming Language Concepts and Paradigms*, C.A.R. Hoare series editor, Prentice Hall International Series in Computer Science, 1ª edição. 1990.