

Programação: Fatores de Qualidade

Qualidade de Software (2011.0)

Prof. Me. José Ricardo Mello Viana

Conteúdo

1. Introdução
2. Gap semântico
3. Paradigmas de programação
4. Combinando paradigmas
5. Tratando a complexidade
6. Importância dos diagramas
7. Documentação e Implementação
8. Geradores de código
9. Visão geral sobre métodos formais

Introdução

- Deve-se planejar desde a escolha de ferramentas até o preparo de cronogramas e administração de mudanças
- Sucesso do projeto depende a realização correta das atividades que compõem o ciclo de vida
- Programação deve ser a continuação natural da análise e do projeto e não a simples confecção de código

Introdução

- Programadores praticam de maneira intuitiva
- Sabem o que o software “deve fazer”
- Dificilmente garante a qualidade dos resultados
- Pode ser totalmente inadequada em ambientes organizados
- Veremos as conexões entre a programação e as fases que lhe antecedem

Gap semântico

- Desentendimento entre as pessoas pode gerar defeitos
- Distância semântica ou gap semântico
 - Falta de coincidência entre informação e interpretação
 - Distância entre conhecimento sobre o domínio da aplicação e o formalismo para representá-lo
 - Diferença entre representação computacional e mundo real
- Diferença entre dois conceitos que deveriam ser o mesmo
- Ex:
 - Matemática $x = \frac{y}{z}$
 - Linguagem C: $x = y / z$
 - Linguagem C++: $x = y / z$
- Parecidos ou iguais? Vejamos

Gap semântico

- Semelhança apenas visual
- Matemática guarda seu sentido tradicional
- Nas linguagens é preciso analisar o contexto (variáveis)
- Em C, seria válido:
 - `int x, y, z;`
 - `char x, y, z;`
 - `void* x; char y; int z;`
 - Compilador mostra avisos (warnings), mas compila
- Em C++, fazendo sobrecarga, podemos fazer qualquer coisa:
 - ```
int *C::operator / (C a) {
 • return (int*) (n + a.n)
}
```
  - `int* x; C y, z;`
  - `x = y / z;`
  - Contraria o senso comum, trará dúvidas e possivelmente defeitos

# Gap semântico

- Influência da linguagem
  - Ex1: Fibonacci
    - $Fib_0 = 0, Fib_1 = 1, Fib_n = Fib_{n-1} + Fib_{n-2}$
  - Em C ou Pascal devemos acrescentar diversos elementos
  - Grande problema: sintaxe
    - Toma tempo e não contribui para a solução
  - Se usar vetor deve se preocupar com indexação
  - Se trocar valores deve ser preocupar com a ordem correta
  - Em linguagem funcional Haskell ou Clean fica:
    - $Fib\ 0 = 0$
    - $Fib\ 1 = 1$
    - $Fib\ n = Fib(n - 1) + Fib(n - 2)$
  - Quase nenhum elemento é adicionado

# Gap semântico

- Influência da linguagem
  - Ex2: Árvore genealógica
    - Supõe-se definição de predicados como
      - pai(antonio, roberto), irmao(roberto, danilo)
    - Deseja-se obter relações como avô e tio
    - Em lógica:
      - $\text{avo}(X, Y) \Rightarrow \text{pai}(X, T) \wedge \text{pai}(T, Y)$
      - $\text{tio}(X, Y) \Rightarrow \text{irmao}(X, T) \wedge \text{pai}(T, Y)$
    - Na linguagem Prolog ficaria
      - $\text{avo}(X, Y) \text{ :- pai}(X, T), \text{pai}(T, Y)$
      - $\text{tio}(X, Y) \text{ :- irmao}(X, T), \text{pai}(T, Y)$
    - A escolha da linguagem reduziu o esforço
    - Se fosse usado C ou Java iria requerer um trabalho significativo de definição de estruturas de dados

# Paradigmas de programação

- Diferentes maneiras de pensar a respeito de problemas
- Diferentes algoritmos e graus variados de eficiência
  - Fibonacci em Haskell é mais simples e mais lento
- Paradigmas:
  - Imperativo: sequência de operações e estado interno. Problemas são particionados em subproblemas
  - Funcional: mapeamento entre entrada e saída composto por funções matemáticas. Puramente funcionais não tem variáveis. Usadas em aplicações de inteligência artificial
  - Lógico: cálculo de predicados
- Orientação a objetos vem do imperativo
  - Mais usado hoje
  - Surge ainda a orientação a aspectos

# Paradigmas de programação

- Orientação a objetos
  - Alternativa diferente para particionar problemas, que não se baseia apenas em algoritmos
  - Objetos:
    - Contem um estado geral, invisível ao exterior
    - Contem métodos que mudam seu estado
    - Fornecem uma interface
    - São organizados em hierarquia de classes
  - Identifica-se entidades que fazem parte do problema
    - Sistema que faz conexão via rede (criptografia, compressão)
  - Base da OO: Tipos abstratos de dados (TAD)
    - Cápsula contendo dados acessíveis apenas por seus métodos
    - Auxilia a manipulação de informações (ex: strings)

# Paradigmas de programação

- Objetos são criados a partir de classes
  - Representação de um conjunto de objetos que compartilham as mesmas características
- Vantagem: ocultação da informação
  - Definição de interfaces deve ser bem definida
- Encapsulamento aumenta a possibilidade de reuso (ex:cliente)
- Herança permite o reuso de código
- Smaltalk é um dos raros exemplos de linguagem puramente orientada a objetos
- Não existe alternativa única de paradigma que seja perfeita
  - Usar a mais adequada a cada momento
- Ex: C++, Java, .NET

# Paradigmas de programação

- Orientação a aspectos
  - Existem requisitos funcionais difíceis de mapear
  - Ex: log, cria-se uma única classe, mas todas as outras usam
  - É um requisito transversal ou aspecto
  - Outro exemplo: exigir que todas as divisões sejam protegidas
  - Aparecem de maneira difusa, em todo o código
  - Gera:
    - Entrelaçamento: mistura com o código de requisitos funcionais
    - Espalhamento: presente em todo o código e não somente em um ponto
  - Consequências:
    - Dificuldade de rastrear e compreender o código
    - Possível menor produtividade dos desenvolvedores
    - Baixo grau de reuso
    - Baixa coesão e alto acoplamento

# Paradigmas de programação

- AOP combina objetos e aspectos
  - O estado do objeto é alterado quando há chamada explícita
  - Para aspectos, o estado pode ser alterado em diferentes pontos do programa
- Definem-se pontos de junção (jointpoints)
  - Tipicamente em chamadas de métodos e tratamento de exceções
  - Conjuntos de junção: reunir informações sobre o contexto desses pontos
  - Regras de junção: códigos relativos aos requisitos transversais que devem ser executados em pontos de junção
- Pode melhorar a qualidade ao aumentar a modularização
  - Estudos ainda devem ser feitos para avaliar seus impactos

# Paradigmas de programação

- AspectJ
  - Extensão de Java para suportar AOP
  - Conjunto de junção é formado por um ou mais pontos de junção
    - pointcut `verif (int i)`:
    - `call (raiz (int i))`
  - Chamadas a rotina `raiz` serão capturadas com seu parâmetro
    - pointcut `verif2 ()`:
    - `call (public * ClassX.* (..))`
  - Captura todas as chamadas de métodos públicos da classe `ClassX`
  - AspectJ Permite a coleta de diferentes pontos
  - Execução por advices
    - `before() : verific (int i) {`
    - `if (i > 0) System.out.println("problemas")`
    - `}`

# Combinando paradigmas

- Tirar proveito das características de várias linguagens
- Escrevendo módulos em diferentes linguagens e fazendo a ligação entre objetos correspondentes
- Ou usar bibliotecas que ofereçam recursos de outra linguagem
  - Ex: Amzi! Para C++, que implementa Prolog
  - FPC++, com recursos de linguagens funcionais
- Porque não criar uma linguagem universal com todos os paradigmas?
  - A introdução de muitos recursos torna a linguagem difícil de usar

# Tratando a complexidade

- Programas grandes são difíceis de projetar, construir e manter
- Não é exclusividade da área de software nem é um problema novo
  - Dijkstra, 1979: A técnica de dominar a complexidade é conhecida desde os tempos antigos: dividir para conquistar
- Na programação está sempre presente e não há solução definitiva

# Tratando a complexidade

- Técnicas estruturadas
  - Na década de 70 surgiram várias técnicas
  - Principal objetivo era a decomposição de problemas em subproblemas
  - Técnicas orientadas a objetos surgirão agregando novas possibilidades, mas não desmentindo o que já existia
  - Ficaram conhecidas pelos nomes dos criadores: Gane e Sarson, Yourdon, Tom de Marco.
  - Ferramenta para análise estruturada:
    - Diagrama de fluxo de dados (DFD)
    - Dicionário de dados (DD)

# Tratando a complexidade

- Diagrama de Fluxo de Dados
  - Representação gráfica que mostra o fluxo de dados e suas transformações à medida que se movem da entrada para a saída
  - Vários níveis de abstração
    - Nível 1: diagrama de contexto, visão geral do problema
    - Detalhado em sucessivos níveis
  - Muitos usados por sua simplicidade
  - Não representam paralelismo ou sequência lógica
  - Transformação em código não é natural

# Tratando a complexidade

- Dicionário de dados
  - Conjunto de informações contendo definições e representações de diversos itens de dados
  - Usado no contexto de sistemas de gerenciamento de banco de dados
  - Pode conter informações como:
    - Definições de elementos de dados
    - Restrições de integridade
    - Informações para auditoria
  - Norma ISO/IEC 11179 ou XML

# Tratando a complexidade

- Padrões de projeto
  - Referência mais atual de software: GoF Patterns
  - Não esgotam o assunto, existem muitos outros
- Padrões GoF
  - Creational – Criacional: Tornar o software independente de como os objetos são criados
  - Structural – Estrutural: Como agrupar objetos e classes para formar uma estrutura maior
  - Behavioral – Comportamental: Interconexão entre objetos, para atingir um objetivo

# Tratando a complexidade

- Padrões GoF

| Criacional                                                             | Estrutural                                                     | Comportamental                                                                                                                                                   |
|------------------------------------------------------------------------|----------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Factory Method<br>Abstract Method<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Proxy | Interpreter<br>Template Method<br>Chain of responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Flyweight<br>Observer<br>State<br>Strategy<br>Visitor |

# Importância dos diagramas

- Contribui para que os requisitos sejam mais facilmente compreendidos e documentados
- Notações gráficas padronizadas permitem que diferentes profissionais interpretem da mesma forma
- Principais hoje: UML e Redes de Petri

# Importância dos diagramas

- UML
  - Também surgiram diversas metodologias OO
  - Notação gráfica orientada a objetos padrão foi criada
  - Unified Modeling Language (UML)
  - Permite a modelagem em várias fases do desenvolvimento
  - Fornece múltiplas visões do sistemas, cada uma com um enfoque
  - Dois tipos:
    - Estáticos: parte do sistemas relacionada com dados, arquitetura física e de software. Ex: classe, objetos, componentes, implantação
    - Dinâmicos: parte de controle do sistema. Ex: caso de uso, sequência, colaboração, atividades e estado-transição

# Importância dos diagramas

- Exemplos
  - Diagrama de caso de uso
  - Diagrama de classe
- A partir de diagramas, ferramentas como Jude ou StarUML geram “esqueletos” de código, facilitando o trabalho do programador e aproximando o projeto da implementação

# Importância dos diagramas

- Redes de Petri
  - Permite a representação de sistemas a eventos discretos, caracterizados por mudarem de estado
  - Pode representar vários eventos paralelos
  - Modelar disponibilidade de recursos
  - Verificar formalmente diversas propriedades (existência de deadlocks e de estados não alcançáveis)
  - Exemplo

# Documentação e Implementação

- Os próprios desenvolvedores complicam o entendimento do trabalho.
  - Ex: Falta de padrão de documentação
- Problemas de comunicação e entendimento no trabalho em equipe levam a dificuldades de implementação, compreensão e testes
- Problemas dos programadores que inexistem realmente:
  - Números não sofrem aproximações por truncagem
  - Quando uma pessoa sai de uma fila não há ponteiros perdidos
  - Documentos em uma mesa não somem quando falta energia
- O uso de modelos auxilia

# Documentação e Implementação

- Efeito das falhas de comunicação
  - Na universidade dificilmente se percebe
  - Separação geográfica entre as pessoas agrava o problema
  - Funções que podem acabar caindo em falhas
    - `int set_log_file(char *s)`
- Documentação e código
  - Importância da documentação
    - São acessíveis aos membros da equipe
    - Linguagens como UML são padronizadas
    - Documentos não esquecem nem tem opiniões próprias
  - Além de descrever o projeto servem para preparar a implementação

# Documentação e Implementação

- Programação literal
  - Pessoas deveriam escrever códigos para outras pessoas e não para computadores
  - Observe:
    - $f = 1 : 1$ : zipWith (+) f (tail f)
    - Produz a série Fibonacci em Haskell de forma mais eficiente
  - Comentários explicam o código-fonte
    - Programadores tendem a comentar somente partes complexas
  - Donald Knuth propões inverter a proporção código/comentário
    - Programador fica mais atento ao funcionamento
    - Mais lento? (ilusão)
      - Documentação de qualidade superior e pronta junto com o programa
    - Haskell possui suporte
  - Demanda tempo e investimento incompatível com as pressões
  - Debug ou rastreio não deveria existir
    - Correção dentro de um documento??

# Geradores de código

- Novas linguagens e ferramentas facilitam a vida o desenvolvedor
- Como se programava no ENIAC?
- Costuma-se dividir em gerações
  - 1: Configuração de circuitos (0s e 1s)
  - 2: Assembly
  - 3: Alto nível (Pascal, C, Fortran, Java)
  - 4: Predefinidos a partir de relatórios, telas e ferramentas CASE
- Não tem relação perfeita com o tempo
- Programação visual
  - Ferramentas de desenho de telas e relatórios. Delphi!!
  - Clarion: Programa completo a partir de DER
  - Ruby on Rails!!!

# Geradores de Código

- Vantagens:
  - Há pouca diferença entre projeto e produto (telas e relatórios)
  - Não há sintaxe com o que se preocupar
  - Grande reaproveitamento de código
  - Reaproveitamento em nível de componentes

# Visão geral sobre métodos formais

- Possibilidade de verificar propriedades do sistema
- Garantir que durante seu funcionamento não sejam violadas
  - Permite que o desenvolvedor se concentre nas propriedades do sistema e deduza conclusões sobre seu funcionamento
- Associa-se com a ideia de refinamentos sucessivos
  - Especificação geral e refinamentos
  - A última é o próprio programa
- Comparação com linguagens

| Linguagens de Programação                                               | Métodos formais                                                                   |
|-------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Nível de abstração baixo                                                | Nível de abstração alto                                                           |
| Ênfase em implementação                                                 | Ênfase em propriedades                                                            |
| Descrever <b>como</b> funcionar                                         | Descrever <b>o que</b> fazer                                                      |
| Orientado a operação de computador e detalhes como alocação de recursos | Obedecem a princípios lógicos e matemáticos e oferecem mecanismos de demonstração |