

# Interfaces de Programação de Aplicações (Application Programming Interfaces)

Cleudson R. B. Souza  
Departamento de Informática  
Universidade Federal do Pará



## Roteiro

- Definição
- Vantagens
- Exemplos
- Utilização na Indústria
- Considerações de Projeto
- Evolução

## Conceitos Básicos

- Modularidade: a divisão de um sistema em partes para que possamos tratar estas partes isoladamente. Um mecanismo para lidar com a complexidade no desenvolvimento de produtos, não apenas software.
  - Um sistema deve ser dividido em módulos para facilitar sua construção
- Parnas\* propôs o princípio de ocultamento da informação como um critério indicando como os módulos devem ser separados.
  - Parnas, D. L. (1972). "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM 15(12): 1053-1058.

## Conceitos Básicos

- Módulos não devem expor as suas partes que podem mudar, seus detalhes de implementação, para que eles não afetem seus clientes.
- Encapsulamento:
  - O código cliente pode usar apenas a interface para a operação.
  - A implementação do objeto pode mudar, para corrigir erros, aumentar performance, etc sem que seja necessário modificar o código do cliente.
  - A manutenção é mais fácil e menos custosa.
  - Cria um programa legível e bem estruturado

## APIs

---

- Uma interface de programação de aplicações (em inglês API) é definida pelo Software Engineering Institute como:
  - API é uma tecnologia que facilita a troca de mensagens ou dados entre duas ou mais aplicações. Uma API é uma interface virtual entre duas funções do software como um processador e uma planilha... Uma API é o software que é usado a suportar a integração de vários produtos de software “off-the-shelf” ou aplicações recém-desenvolvidas com aplicações novas ou existentes.

## APIs

---

- Na prática, uma API significa ou indica uma interface bem-definida que define um conjunto de serviços que um componente, módulo, ou aplicação fornece a outros elementos de software.
- des Rivieres:
  - Uma API é uma interface bem-definida que permite que um componente de software acesse através de seu código outro componente, e isto é suportado pela linguagem de programação.

## APIs

---

- A palavra interface é usada em APIs para explicitamente indicar que esta é uma construção que existe entre dois “mundos”:
  - Interface com o usuário;
  - Interface de programação de aplicações;
- No caso de APIs, os “mundos” são dois elementos de software (componentes, classes, módulos) ou até mesmo duas aplicações.

## Exemplos de APIs

---

- Sistemas Operacionais:
  - Win32
  - Mac OS
  - J2SE
- IDEs
  - Eclipse
  - NetBeans
- APIs bem projetadas são duradouras;
- Elas não quebram os clientes existentes - evolução com compatibilidade;
- Elas deixam o código binário continuar funcionando - compatibilidade binária;

## APIs em Java

- Não confundir APIs com o conceito de interfaces em Java!
- Tipicamente, em uma linguagem como Java, uma API corresponde a um conjunto de:
  - classes + interfaces + métodos públicos + a documentação associada.

## APIs na indústria

- APIs são adotadas pela indústria porque elas suportam a separação entre a interface e a implementação de componentes.
- Pode-se separar elementos de software entre uma parte pública (a API) e a parte privada (a implementação), de tal maneira que mudanças na parte privada podem ser feitas sem afetar a parte pública, consequentemente diminuindo a dependência (acoplamento, coupling) entre as o usuário da API e a API.

## APIs na indústria

- APIs e interfaces também facilitam a coordenação das atividades dos engenheiros de software. Se dois engenheiros (João e Maria) concordam na API que integra seus componentes, eles podem trabalhar em paralelo. João não precisa saber dos detalhes do componente que Maria está implementando e vice-versa, desde que os dois “honrem” o que foi definido na API.

## Mas, para que isso aconteça ...

- APIs “entre amigos” (dentro de uma mesma empresa)
  - Não existem clientes externos
  - Não precisa ser estável entre duas releases
  - Erros podem ser corrigidos
- APIs que se tornam públicas
  - Precisam ser estáveis porque suportam um enorme número de clientes
    - A API do Lotus Notes tem 8.000 clientes!
  - Quando uma API é publicada, não há como voltar atrás
  - Mudanças que quebram a API são uma péssima política
  - Em caso de mudanças na próxima versão, deve-se estender a API

## Projeto de APIs - Considerações

- É necessário
  - Informar os clientes de como usar a API –“API specs”
  - Escolher nomes de métodos e classes **informativos**
  - Decidir **cuidadosamente** entre usar classes ou interfaces
- Quando uma API é publicada
  - Ela tem de trabalhar de acordo com a estrutura que ela propôs
  - É necessário melhorar a API **sem quebrar os clientes existentes**
- “API specs” mal projetadas
  - Confundem os clientes
  - E quando consertadas podem piorar as coisas

## Projeto de APIs - Considerações

- Tem de existir uma fronteira clara entre o que é parte e o que não é parte da API
  - Exemplo: no Eclipse *packages* que não são da API tem o nome “internal”:  
org.eclipse.ui.internal
- Não deve haver conexão visível entre a API e a implementação da mesma
  - Detalhes de implementação não devem existir na API

## Evolução de APIs

- APIs naturalmente vão evoluir a cada nova versão
  - Changes to API could invalidate existing clients
- A evolução tem de ser feita de maneira compatível
  - Mantenha os clientes funcionando;
  - A API deve fornecer os mesmos serviços;
- Duas considerações gerais
  - Compatibilidade no contrato – honre o contrato das APIs existentes
  - Compatibilidade binária - o código tem de continuar funcionando;

## Compatibilidade no contrato

- Antes

```
/** Retorna uma lista não-vazia. */
public int[] getIndices();
```
- Depois

```
/** Retorna uma lista. Esta lista PODE ser vazia.*/
public int[] getIndices();
```
- Poderia “quebrar clientes”

```
int[] d = getIndices();
System.print(d[0]); // array index out of bounds
```
- Mas, se minha implementação faz:

```
public int[] getIndices() {
    ...; return result; // resultado não é vazio
}
```

## Compatibilidade binária

---

- Antes  
`public void register(String key);`
- Depois  
`public void register(Object key);`
- As chamadas existentes recompilam sem erro  
`register("foo");`
- Mas, o código compilado existente indicará erro  
`register("foo"); // link error`

Porquê?

## Como quebrar a compatibilidade binária?

---

- Renomear um pacote, classe, método ou atributo;
- Remover um pacote, classe, método ou atributo;
- Diminuir a visibilidade (de *public* para *private*);
- Adicionar ou remover parâmetros;
- Mudar o tipo de um parâmetro;
- Mudar o tipo de retorno de um método;
- Adicionar ou remover exceções de um método;
- Mudar o tipo de um atributo;
- Mudar o valor de uma constante;

## Como quebrar a compatibilidade binária?

---

- Mude um método de um objeto para um método da classe (ver conceito de meta-classe);
- Mude um atributo de um objeto para um atributo da classe (ver conceito de meta-classe);
- Mude uma classe para uma interface ou vice-versa;
- Faça uma classe *final*, se os clientes podem criar subclasses dela;
- Faça uma classe *abstract*, se os clientes podem criar instâncias dela;

## Como manter a compatibilidade binária?

---

- Adicionar *packages*, classes, e interfaces;
- Modificar a implementação de um método;
- Modificar elementos que não pertencem a API;
- Adicionar atributos as classes e interfaces;
- Adicionar métodos *abstract* as classes (se os clientes não criam subclasses);
- Adicionar métodos as interfaces (se os clientes não implementam as interfaces);
- Adicionar métodos não *abstract* as classes (se os clientes herdam destas classes);
- Mudar o valor de um atributo se ele não é constante;

## Como manter a compatibilidade binária?

---

- Mover um método de uma classe para sua superclasse;
- Tornar uma classe *final* em *não-final*;
- Tornar uma classe *abstract* em concreta;
- Mudar o nome de um parâmetro em um método;

## Substituindo Métodos em uma API

---

- Adicione o novo método na API;
- Torne o método original "deprecate"
  - Faça um "forward" do método velho para o novo;
  - Mas, você tem de garantir que o método original continua funcionando

```
package org.eclipse.jdt.core.dom;
public class Message {
    ...
    /** ...
     * @deprecated Use getSourcePosition() instead
     */
    public int getSourcePosition() {
        return getStartPosition(); // forward method
    }

    public int getStartPosition() {
        ...
    }
}
```

## Referências

---

- <http://openide.netbeans.org/nonav/tutorial/api-design.html>
- <http://www.eclipse.org/eclipse/development/java-api-evolution.html>
- *How to Use the Eclipse API*, <http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html>
- *Requirements for Writing Java API Specifications* <http://java.sun.com/products/jdk/javadoc/writing-apispecs/index.html>

## Times

- **Análise e Interfaces**
  - alline lemos
  - alline peixoto
  - leandro
  - luiz otavio
  - paula danielle
  - priscila
- **Arquitetos**
  - adailton
  - anderson
  - marcelo
  - marcio braga
  - rafael
  - weverton
- **Implementação**
  - Cleberson
  - Dedier
  - Diego
  - Leonardo
  - Marcio Kuroki
  - Marlos
  - Tacio
- **Teste**
  - aline patricia
  - billy
  - breno
  - franklin harrison
  - joseane
  - luiz alberto
  - pedro leandro

## Responsabilidades dos Times

- **Time 1 - Análise e Interfaces**
  - Responsável por TODAS as comunicações com o cliente!
  - Especificação de requisitos
    - Use-Cases
    - Diagrama de transição de estados da interfaces
    - Usuário (fictício) típico: nome, idade, escolaridade, etc.
  - Manual da ferramenta.
- **Time 2 - Arquitetos**
  - Arquitetura em Camadas da Aplicação
  - APIs para cada camada
    - Diagramas de Classes, Arquitetura, etc.
    - Código fonte conforme discutido anteriormente;

## Responsabilidades dos Times

- **Time 3 - Implementação**
  - Código Java;
  - Documentação em Javadoc;
  - Terá auxílio do time de arquitetos;
- **Time 4 - Testes**
  - Gerência de configuração;
  - Testes unitários para cada camada;
  - Testes de integração do sistema;
    - Precisa popular um banco de dados para efetuar os testes;

## Observações

- Cada time precisa eleger um gerente.
- A única exceção é o time dos arquitetos: Adailton é o gerente.

**Dez minutos para vocês decidirem!**

## Observações

- Cada time precisa definir as responsabilidades de cada integrante. Por exemplo:
  - Analistas: quem irá contactar o cliente?
  - Arquitetos: quem usará a ferramenta Rose para os diagramas?
  - Implementação: quem instalará o banco de dados SPRING ou a ferramenta TerraLib? Configuração do servidor CVS?
  - Testes: quem instalará o banco de dados SPRING ou a ferramenta TerraLib?

## Apresentações

- Time 1 - Analistas:
  - Especificação de requisitos: 7 de fevereiro;
  - Especificação da interface com usuário: 21 de fevereiro;
  - Manual do Usuário: 04 de Abril
- Time 2 - Arquitetos:
  - Arquitetura: 21 de fevereiro;
- Time 3 - Implementação
  - Demonstração da ferramenta ANTES dos testes
  - Primeira Parte: 07 de Março, 21 de Março e 04 de Abril;
- Time 4 - Testes
  - Testes Unitários: 07 de Março;
  - Relatório de Testes Unitários: 14 de Março;
- Apresentação para o Cliente
  - 11 de Abril de 2006.

**Datas aproximadas!**

## Em resumo, na próxima aula.

- Professor: informações de como contactar o cliente.
- Alunos:
  - Reunião para distribuição de atividades;
  - Preparação de uma planilha inicial com a descrição das atividades de cada integrante;
  - Planilha deve ser entregue na aula seguinte.
- Recomendação:
  - Todos os times devem estudar aspectos da ferramenta SPRING e TerraLib.
  - Cada time deve estudar aspectos relacionados.
- Início da Aula as 3 da tarde.

## Perguntas?

## Observações

---

- Atraso na entrega e/ ou apresentação do trabalho implica no desconto de 1 pt por dia útil na pontuação total do time