Reúso de Software

THAINÁ MARIANI, 2016.

Introdução

- Reúso é utilizado no dia-a-dia em diferentes contextos;
- Relativo ao aproveitamento de soluções para problemas similares.
- No contexto de Engenharia de Software, refere-se a criação de software a partir de componentes, ideias ou processos já existentes.
- O problema consiste em encontrar uma maneira sistemática e formal de realizar o reúso.

Definição

- Reúso de software é o processo de incorporar produtos existentes em um novo produto.
- Exemplos:
  - Código;
  - Especificações de Requisitos e Projeto;
  - Planos de Teste;
  - Conhecimento.

Benefícios

- Aumento da Produtividade;
- Diminuição do tempo de desenvolvimento e validação → Redução de custo;
- Qualidade dos Produtos;
- Flexibilidade na estrutura do software;
- Manutenibilidade;
- Familiaridade com o uso de padrões leva a menos erros.

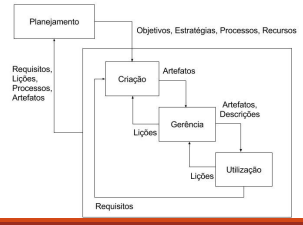
Dificuldades

- Identificação e compreensão dos artefatos;
- Qualidade dos artefatos;
- Modificação dos artefatos;
- Falta de confiança nos "artefatos dos outros",
  Mito: "not invented here"
- Ferramentas de apoio;
- Aspectos legais e econômicos;
- Falta de incentivo.

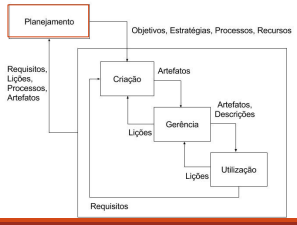
Requisitos

- Catalogação, documentação e certificação completa do artefato a ser reutilizado, de modo a ser possível:
  - Encontrar o artefato a ser reutilizado;
  - Compreender o artefato para adaptá-lo ao novo contexto;
  - Garantir que o artefato se comportará conforme especificado.

Modelo de Processo de Reúso



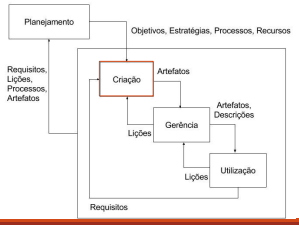
Planejamento do Reúso



Planejamento do Reúso

- Atividades
  - Coordenar o processo de reúso;
  - Definir prioridades e cronogramas para os artefatos a serem construídos;
  - Resolver conflitos quando artefatos necessários não estão mais disponíveis;
  - Integrar o reúso no processo de desenvolvimento.

Criação dos Artefatos



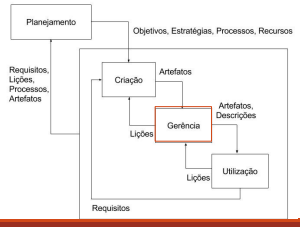
Criação dos Artefatos

- Atividades:
  - Pesquisar os artefatos;
  - Avaliação das necessidades do utilizador;
  - Tecnologias e novidades do mercado;
  - **Engenharia de Domínio.**

Engenharia de Domínio

- Domínio:
  - Uma coleção de problemas reais
  - Uma coleção de aplicações que compartilham características comuns
- Identificar, construir, catalogar e disseminar um conjunto de artefatos que podem ser utilizados em softwares de um domínio específico.
- Com a Engenharia de Domínio é possível definir modelos de domínios e arquiteturas comuns à uma família de aplicações.
- Formalizar teorias específicas a um domínio.

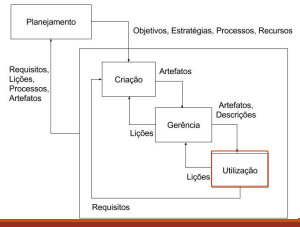
Gerência dos Artefatos



Gerência dos Artefatos

- Atividades:
  - Certificar novos artefatos;
  - Classificar novos artefatos no repositório;
  - Suporte ao processo de utilização;
  - Checar as necessidades do processo de utilização com os artefatos existentes;
  - Coletar feedback e reportar erros.

Utilização dos Artefatos



Utilização dos Artefatos

- Atividades:
  - Análise dos requisitos do produto a ser construído;
  - Especificação do produto a ser construído;
  - Identificar os artefatos a serem reutilizados;
  - Compreensão dos artefatos;
  - Avaliação dos artefatos;
  - Adaptação dos artefatos;
  - Integração dos artefatos.

Uma boa técnica de reúso deve

Garantir adaptação e adequação a um novo contexto:

- Abstração
- Seleção;
- Especialização;
- Integração.

Técnicas para Reúso

- Bibliotecas;
- Frameworks;
- Componentes;
- Padrões de Software
- Linhas de Produto de Software

Bibliotecas de Classes

Bibliotecas de Classes

- Classes de uso genérico podem ser disponibilizadas para reúso e importadas em múltiplas aplicações
- Em geral são incorporadas ao código final da aplicação, ou seja, são compiladas juntamente com o restante do código.

Linha de Produto de Software

Definição

- É um conjunto de produtos (software) que compartilham características em comum para satisfazer um determinado segmento de mercado;
- Linhas de produtos são comuns em outras áreas;
- Carros são exemplos comuns.

Benefícios

- Melhora a produtividade;
- Melhora a qualidade;
- Satisfação do cliente;
- Reduz os custos;
- Rápida disponibilidade no mercado;
- Customização em massa.

Riscos

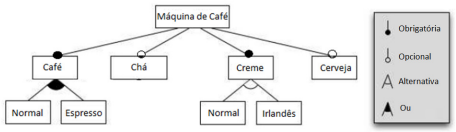
- Adotar uma estratégia nova;
- Estratégias organizacionais;
- Estratégias gerenciais.

Características (*Features*)

- Funcionalidades relevantes do sistema e visíveis ao usuário;
- Características podem ser obrigatórias, alternativas ou opcionais;
- Diagrama de Características (*Feature Model)*.

Características (*Features*)

- Funcionalidades relevantes do sistema e visíveis ao usuário;
- Características podem ser obrigatórias, alternativas ou opcionais;
- Diagrama de Características (*Feature Model)*.



Gerenciando Variabilidades

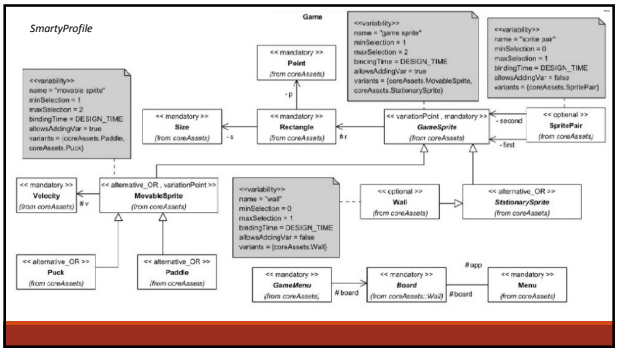
- Similaridades: Características comuns a todos os produtos;
- Variabilidades: Características que diferenciam os produtos;
- Pontos de Variação;
- Variantes.
- O núcleo em comum da família é reutilizado cada vez que uma nova aplicação é desejada.

Gerenciando Variabilidades

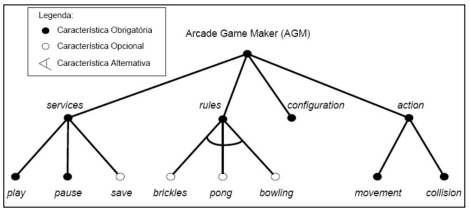
- Representação das variabilidades
  - São geralmente definidas em diagramas, como diagramas de classes, casos de uso, colaboração e atividades;
  - Podem ser representadas por estereótipos em classes;
  - Existem diferentes notações para representar variabilidades.

Arquitetura de LPS

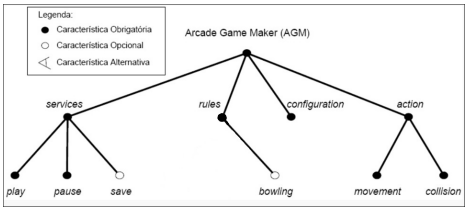
- Contém todas as similaridades e variabilidades da LPS;
- É utilizada para derivar os produtos;
- É o artefato mais importante da LPS.



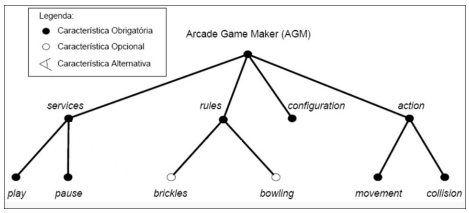
LPS - *Arcade Game Maker* (AGM)



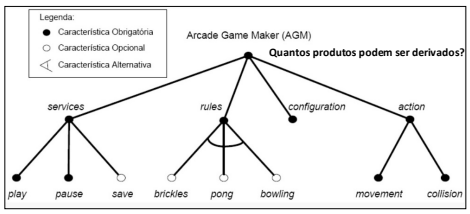
LPS - *Arcade Game Maker* (AGM)



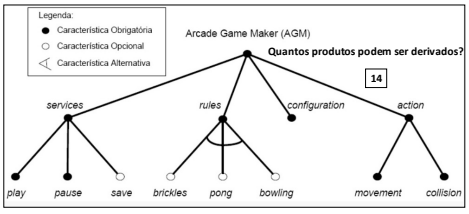
LPS - *Arcade Game Maker* (AGM)



LPS - *Arcade Game Maker* (AGM)



LPS - *Arcade Game Maker* (AGM)



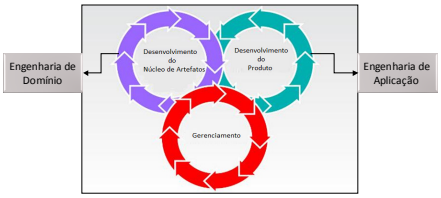
LPS - *Arcade Game Maker* (AGM)

- Define três **principais** produtos, um para cada jogo
  - Brickles: Jogo de blocos;
  - Pong: Jogo de tênis de mesa;
  - Bowling: Jogo de boliche.

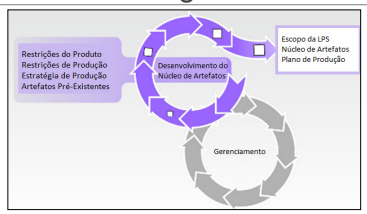
Engenharia de LPS

- Apresenta meios de construir e gerenciar uma LPS;
- Fornece um conjunto comum de artefatos.

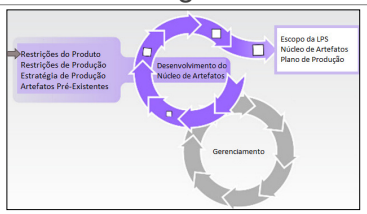
Atividades da Engenharia de LPS



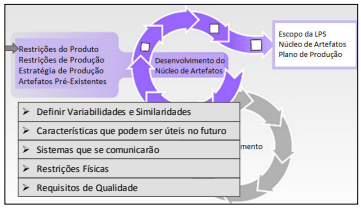
Atividades da Engenharia de LPS



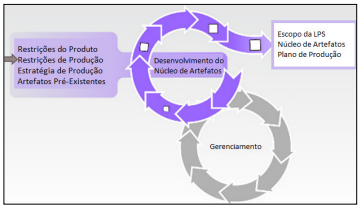
Atividades da Engenharia de LPS



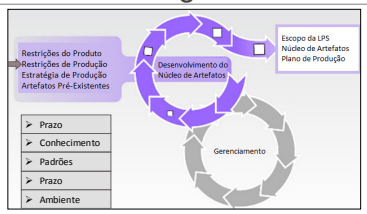
Atividades da Engenharia de LPS



Atividades da Engenharia de LPS

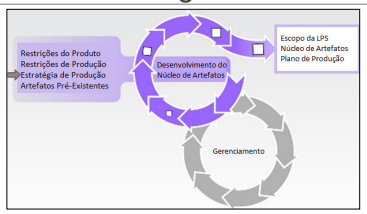


Atividades da Engenharia de LPS

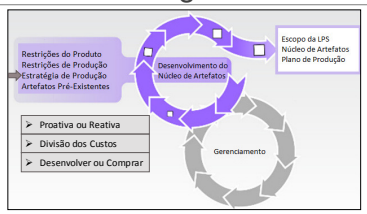


- Prazo
- Conhecimento
- Padrões
- Prazo
- Ambiente

Atividades da Engenharia de LPS

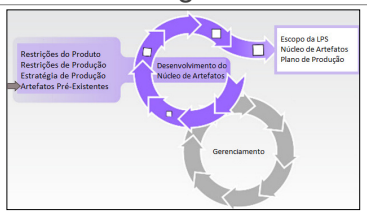


Atividades da Engenharia de LPS

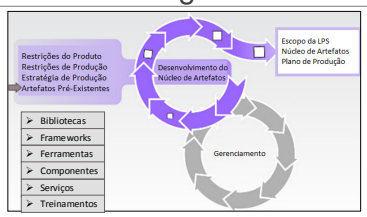


- Proativa ou Reativa
- Divisão dos Custos
- Desenvolver ou Comprar

Atividades da Engenharia de LPS

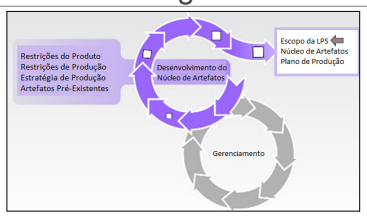


Atividades da Engenharia de LPS

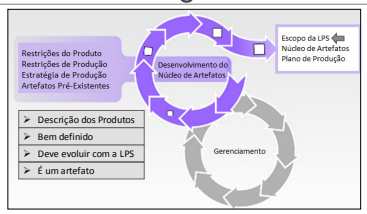


- Bibliotecas
- Frameworks
- Ferramentas
- Componentes
- Serviços
- Treinamentos

Atividades da Engenharia de LPS

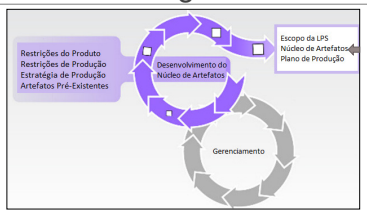


Atividades da Engenharia de LPS

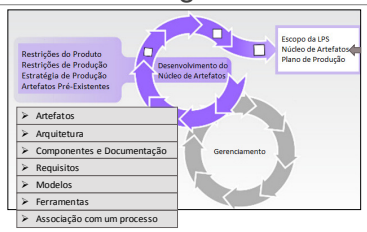


- Descrição dos Produtos
- Bem definido
- Deve evoluir com a LPS
- É um artefato

Atividades da Engenharia de LPS

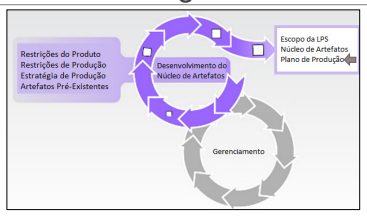


Atividades da Engenharia de LPS

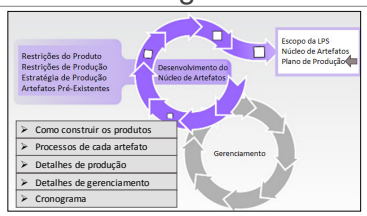


- Artefatos
- Arquitetura
- Componentes e Documentação
- Requisitos
- Modelos
- Ferramentas
- Associação com um processo

Atividades da Engenharia de LPS

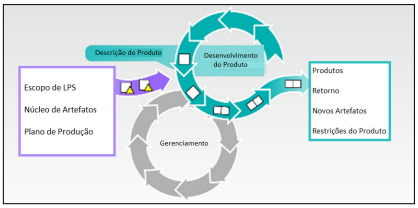


Atividades da Engenharia de LPS

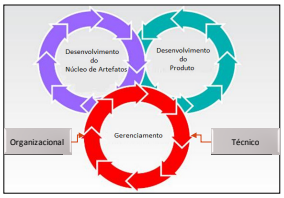


- Como construir os produtos
- Processos de cada artefato
- Detalhes de produção
- Detalhes de gerenciamento
- Cronograma

Atividades da Engenharia de LPS



Atividades da Engenharia de LPS



Referências

Roger Pressman. 2009. Software Engineering: A Practitioner's Approach (7 ed.). McGraw-Hill, Inc., New York, NY, USA.

Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. Software Reuse: Architecture, Process and Organization for Business Success. ACM Press/Addison-Wesley Publ. Co., New York, NY, USA

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus.

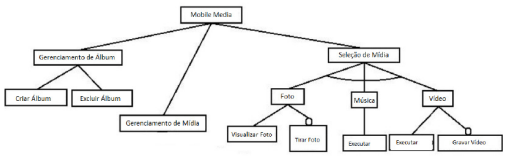
Software Engineering Institute (SEI), Software Product Lines:

http://www.sei.cmu.edu/productlines/

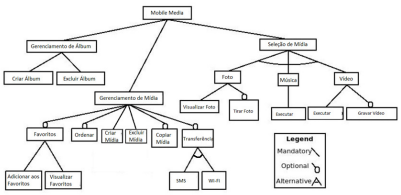
Cláudia Maria Lima Werner, Slides sobre Reutilização de Software: http://slideplayer.com.br/slide/367783/

Edson Oliveira Junior, Slides sobre Linha de Produto de Software: http://pt.slideshare.net/edson_ao_junior/mini-curso-avaliacao-lps-edson-jr-animacao

Exercício



Exercício



Padrões

Conceito de Padrão

- Um template (formulário) de solução para um problema recorrente que seja comprovadamente útil em um determinado contexto.
- Um padrão de software é instanciado através da vinculação de valores a seus parâmetros.
- Os padrões podem existir em várias escalas e níveis de abstração; por exemplo, como *padrões de arquitetura*, *padrões de análise*, *padrões de projeto*, *padrões de teste* e *idiomas* ou *padrões de implementação*.

Histórico

Arquiteto ->Christopher Alexander –

Linguagem de padrões em arquitetura.

- catálogo com 253 padrões para edificações ligadas a regiões, cidades, transportes, casas, escritórios, paredes, jardins, etc.

Definição

"Um padrão expressa uma solução reutilizável descrita através de três partes: um contexto, um problema e uma solução". (GAMMA et al., 1995).

Contexto: estende o problema a ser solucionado, apresentando situações de ocorrência desses problemas.

Problema: determinado por um sistema de forças, onde estas forças estabelecem os aspectos do problema que devem ser considerados.

Solução: mostra como resolver o problema recorrente e como balancear as forças associadas a ele.

Padrões em ES

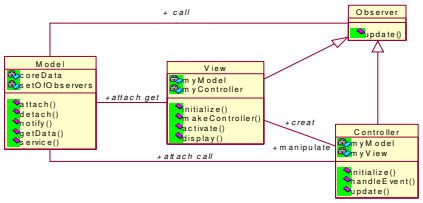
Padrões em ES permitem que desenvolvedores possam recorrer a soluções já existentes para solucionar problemas que normalmente ocorrem em desenvolvimento de software;

Surgimento: início dos anos 90;

- 1995 - livro da "Gang of Four" (GoF)
  - 23 padrões de projeto (design patterns)
- OOPSLA

Padrões capturam experiência existente e comprovada em desenvolvimento de software, ajudando a promover boa prática de projeto.

Exemplo de Padrão Arquitetural: MVC



Model-View-Controller

Motivação: Separação de interesses.

- Considere o uso deste padrão quando estiver desenvolvendo sistemas com interface de usuário.
- Interfaces de usuário são propensas a mudanças. Por exemplo, quando as funcionalidades de uma aplicação são estendidas ou adaptadas, menus devem ser modificados.
- O sistema pode ser executado em diferentes plataformas, com diferentes padrões de interface gráfica.
- A interface de usuário deve estar o mais independente possível do *kernel* funcional da aplicação.

Nome do Padrão
Camadas

Contexto Um sistema grande que requer decomposição.

Problema
Um sistema que deve resolver as questões em diferentes níveis de abstração. Por exemplo: as questões de controle de hardware, as questões de serviços comuns e as questões específicas de domínio. Seria extremamente indesejável escrever componentes verticais que lidem com essas questões em todos os níveis. Uma mesma questão deveria ser resolvida (possivelmente de maneira inconsistente) várias vezes em diferentes componentes.

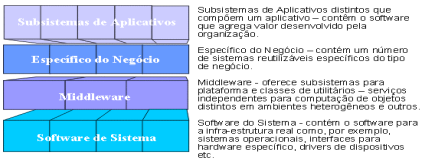
Força

- As partes do sistema devem ser substituíveis
- As alterações efetuadas nos componentes não devem ser irregulares
- Responsabilidades similares devem ser agrupadas juntas
- Tamanho dos componentes—componentes complexos talvez precisem ser decompostos

Solução

Estruture os sistemas em grupos de componentes que formem camadas umas sobre as outras. Faça com que as camadas superiores utilizem os serviços somente das camadas abaixo (nunca das camadas acima). Tente não usar serviços que não sejam os da camada diretamente abaixo (não pule camadas, a menos que as camadas intermediárias somente adicionem componentes de acesso).

Exemplo



Subsistemas de Aplicativos distintos que compõem um aplicativo – contêm o software que agrega valor desenvolvido pela organização.

Específico do Negócio – contêm um número de sistemas reutilizáveis específicos do tipo de negócio.

Middleware - oferece subsistemas para plataforma e classes de utilitários – serviços independentes para computação de objetos distintos em ambientes heterogêneos e outros.

Software do Sistema - contêm o software para a infra-estrutura real como, por exemplo, sistemas operacionais, interfaces para hardware específico, drivers de dispositivos etc.

Model-View-Controller

O padrão propõe a divisão de uma aplicação em 3 tipos de componentes: modelo, controle e apresentação

- O ***modelo*** representa as classes do domínio do problema, sendo independente de uma forma específica de apresentação.
- Encapsula os dados e funcionalidade do negócio. O modelo deve ser independente de representações de saída específicas e do tratamento das interações dos usuários com a aplicação.
- A ***visão*** apresenta as informações do modelo ao usuário. Podem existir múltiplas visões de um mesmo modelo. Cada Visão tem um controlador.

Model-View-Controller

- Os ***controladores*** recebem a entrada, geralmente um evento (i.e. movimentos do mouse, ativação de botões, teclas etc.), que é traduzida em requisições de serviços ao modelo ou visão.

Se o usuário altera o modelo através do controlador de uma visão, todas as outras visões dependentes destes dados devem refletir a mudança.

As Visões devem refletir o estado atual do modelo.

Model-View-Controller

Vantagens: flexibilidade e reutilização.

Aplica o padrão de projeto *Observer*.

Pode aplicar o padrão de projeto *Composite* quando trabalha com objetos Visão complexos. Por exemplo, um Frame pode ser composto por painéis etc. O agrupamento de objetos é tratado como um objeto individual.

Outros padrões de projeto podem ser aplicados.

Referências

Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. SIGSOFT Software Engineering Notes, 17(4):408211;52, October 1992.

David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

David Garlan and Mary Shaw. An introduction to software architecture. Technical report CMU-CS-94-166, Carnegie Mellon University, Pittsburgh, PA 15213-3890, January 1994.

Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice. Addison-Wesley Professional, 2 edition, April 2003.

Frank Buschmann, Régine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stahl 1996. *Pattern-Oriented Software Architecture - A System of Patterns*, Nova York, NY: John Wiley and Sons, Inc.

Antônio Mendes. *Arquitetura de Software*. Campus-Elsevier, 2002, (ISBN 853521013X)

http://www.sei.cmu.edu/architecture/

Frameworks

Definição

"Um conjunto de classes cooperantes que constrõem um projeto reutilizável para uma categoria de software específica" (JOHNSON, 1997)

"Um framework é uma coleção de classes abstratas e concretas, e uma interface entre elas, e um projeto para um subsistema" (WIRFS-BROCK E JOHNSON, 1990)

"Um framework é um software parcialmente completo (subsistema) que pretende ser instanciado" (BUSCHMANN ET AL. 1996)

Definição

"Aplicação semi-completa reutilizável que, quando especializada, produz aplicações personalizadas"(Johnson & Foote, 1988)

"Coleção de classes abstratas e concretas e a interface entre elas, representando o projeto de um sub-sistema" (Pree, 1995)

Frameworks: Características

Aspectos variáveis - ***hot-spots:*** representam as partes do framework de aplicação que são específicas de sistemas individuais. São projetados para serem genéricos - podem ser adaptados às necessidades da aplicação

Uma parte do *framework* onde uma adaptação pode ser feita

Exemplos de Hot-spots: Classes Abstratas, métodos abstratos, métodos hook, etc.

Frameworks: Características

Aspectos invariáveis - ***frozen-spots*:** Definem a arquitetura geral de um sistema de software - seus componentes básicos e os relacionamentos entre eles. Permanecem fixos em todas as instanciações do framework de aplicação.

Uma parte do *framework* que não foi projetada para adaptação

Exemplos de Frozen-spots: Classes Concretas, métodos template, etc.

Frameworks: Características

Template Method (padrão de projeto do Gamma):

Assim como as Classes e métodos abstratos, este padrão está no cerne do projeto de um Framework.

- A idéia é definir um método gabarito (o Template Method) em uma superclasse, definindo o esqueleto de um algoritmo com suas partes variantes e invariantes.
- O Template Method invoca outros métodos, alguns dos quais são operações que podem ser redefinidas em subclasses (implementadas por métodos hooks).
- Assim, as subclasses podem redefinir os métodos que variam, de forma a acrescentar o seu próprio comportamento específico nos pontos de variação.

Frozen-Spots e *Hot-Spots*

Identificar e projetar os *hot-spots* em um *framework* é uma das principais dificuldades para desenvolver projetos reutilizáveis.

Um *framework* para ter um alto grau de qualidade deve ter bons *hot-spots* para permitir futuras adaptações.

Frozen-spots definem a arquitetura geral de um sistema, ou seja, seus componentes básicos e os relacionamentos entre eles.

Classificação de *Frameworks* quanto ao escopo

***Frameworks* de Infra-estrutura**

***Frameworks* de Integração**

***Frameworks* de Aplicação**

Classificação de *Frameworks* quanto ao escopo

***Frameworks* de Aplicação**: são dirigidos a um domínio específico de aplicações, ou seja, a uma família de aplicações em uma determinada área.

- voltados a domínios de aplicação mais amplos e são a pedra fundamental para atividades de negócios das empresas.
- exemplos: telecomunicações, aviação, manufatura e engenharia financeira.
- são mais caros para desenvolver ou comprar, mas podem dar um retorno substancial do investimento, já que permitem o desenvolvimento de aplicações e produtos diretamente

Classificação de *Frameworks* quanto ao escopo

***Frameworks* de Integração**: estes *frameworks* são projetados para dar supote à modularização, ao reúso e à integração de aplicações que apresentam componentes distribuídos. (ex: *middleware* para sistemas distribuídos)

- usados em geral para integrar aplicações e componentes distribuídos
- projetados para melhorar a habilidade de desenvolvedoresem modularizar, reutilizar e estender sua infra-estrutura de software para funcionar "sem costuras" em um ambiente distribuído
- exemplos: Object Request Broker(ORB), middleware orientado a mensagens e bases de dados transacionais

Classificação de *Frameworks* quanto ao escopo

***Frameworks* de Infra-estrutura**: *frameworks* que apóiam a infra-estrutura de qualquer tipo de sistema (ex: SISOP, interfaces de usuário, persistência de objetos, de comunicação e de processamento de linguagens)

- simplificam o desenvolvimento da infra-estrutura de sistemas portáveis e eficientes,
- exemplos: sistemas operacionais, comunicação, interfaces com o usuário e ferramentas de processamento de linguagem
- em geral são usados internamente em uma organização de software e não são vendidos a clientes diretamente

Classificação de *Frameworks* quanto à Adaptação

caixa branca (*white box*): baseiam-se no conceito de herança e ligação dinâmica que permite uma sub-classe reutilizar a interface e a implementação de sua super-classe.

caixa preta (*black box*): estão baseados no conceito de composição de objetos onde estes não revelam detalhes internos de sua implementação, tendo-se somente acesso à interface do mesmo.

caixa cinza (*gray box*): permite adaptação tanto por herança e ligação dinâmica, quanto por composição de componentes.

Comparação

- Framework caixa branca é mais fácil de projetar
- Framework caixa preta é mais fácil de usar
- Frameworks caixa-branca evoluem para se tornar mais caixa preta
  - Aumenta o numero de objetos, mas eles ficam menores
  - Complexidade está na interconexão
  - Objetos compostos de objetos menores

Dificuldades em *Frameworks*

- **Desenvolvimento de *frameworks***:
  - *Análise de domínio, projeto arquitetural, projeto do framework, implementação, teste e documentação.*
  - determinar partes variáveis e semelhantes numa família de aplicações;
  - limitar a porção de código necessária para completar a aplicação, a qual deve ser pequena;
  - testes e liberação para uso de um framework;
  - evolução do framework;
  - custo e esforço de desenvolvimento.

Dificuldades em *Frameworks*

- **Utilização de *frameworks***:
  - verificação da aplicabilidade de um framework como solução ao problema em questão;
  - estimativa de esforço na compreensão e uso do framework;
  - confiabilidade.

Referências Bibliográficas:

1. SHAW, M., GARLAN, D., 1996, "Software Architecture: perspectives on an emerging discipline", 1 ed, Nova Jersey, Prentice-Hall: 1996.
2. SEI, 1994, http://www.sei.cmu.edu/ata/ata_init.html.
   PENEDO, M. H., RIDDLE, W., 1993, "Process-sensitive SEE Architecture (PSEEA) – Workshop Summary", Software Engineering Notes, ACM SIGSOFT, vol. 18, n^{o} 3, July, pp.A78 – A94.
3. APPLETON, Brad. Patterns and Software: Essential Concepts and Terminology. Disponível na internet em: http://www.enteract.com/~bradapp/docs/patterns-intro.html.
4. BUSCHMANN, F. et al. Pattern-Oriented Software Architecture: a system of patterns. John Wiley & Sons, England, 1996. 467p.
5. FAYAD, Mohamed et al. Building application frameworks: object-oriented foundations of framework design. John Wiley & Sons, 1999. 664p.

Componentes

Definição

- Objetivo: quebra de blocos monolíticos em componentes interoperáveis
- Componentes são construídos/empacotados com o objetivo de serem reutilizados em diferentes aplicações
- Um componente provê um conjunto de serviços acessíveis por meio de uma interface bem definida
- Motivações: desenvolvimento da Internet/WWW, arquitetura cliente/servidor, computação distribuída, Orientação a Objetos, Componentware, dentre outros

Definição

1. (D'Souza & Wills, 1998): Um pacote coerente de software que
   - (a) pode ser desenvolvido e instalado independentemente como uma unidade,
   - (b) **tem interfaces explícitas e bem definidas para os serviços que provê,**
   - **(c) tem interfaces explícitas e bem definidas para os serviços que espera de outros**,
   - e (d) pode ser utilizado para composição com outros componentes, sem alterações em sua implementação, podendo eventualmente ser customizado em algumas de suas propriedades.
2. (Councill & Heineman, 2001): Um *componente de software* é um elemento que está em **conformidade com um modelo de componentes** e pode ser instalado independentemente e composto sem modificações.

Definição

3. Johannes Sametinger [1997]: Pedaço de software auto-contido, claramente identificável, que descreve ou **executa funções** específicas, tem interfaces claras, documentação apropriada e um status de reúso.

Exemplo de componentes são os plugins – estender funcionalidades, Componentes visuais de interface com o usuário (widgets)

Definição: Componentes e Classes

Diferentes níveis de abstração.
Classe pode ser uma abstração lógica do domínio (classe conceitual) ou uma estrutura de uma linguagem de programação utilizada para instanciar objetos (classe de software).

Um componente é uma unidade executável, que pode ser a implementação "física" de uma ou mais classes. Uma classe só pode pertencer a um único componente

Pode-se ter somente o executável do componente e estes são raramente alterados.

Definição: Componentes x Bibliotecas de funções.

Uma biblioteca provê serviços coesos para um sistema e é auto-contida, reutilizável e substituível.

Uma biblioteca eventualmente também é disponibilizada na forma binária e pode ser orientada a objetos.

Uma biblioteca normalmente não pressupõe uma padronização, instalação e configuração (*deploy*), e nem a existência de uma plataforma específica de execução.

Definição: Componentes

É um elemento arquitetural mapeado em um **arquivo executável**, que segue uma especificação, e foi concebido para ser auto-contido, reutilizável, substituível, além de prover serviços específicos de uma maneira coesa e bem definida.

Portanto, classificar algo como componente depende também de como o código foi concebido e construído e como ele se relaciona com o restante do sistema, e não somente da aderência a uma padronização.

Sinônimos para componentes, utilizados em contextos específicos: plugin, add-on, módulo, serviço e widget.

Benefícios

- Manutenabilidade – fácil substituição, evolução
- Reúso – redução de custos
- Composição e Extensibilidade – facilidade para adição de novos serviços.
- Integração – oferece diferentes visões
- Escalabilidade
- Redução do tempo de desenvolvimento
- Facilidade para prototipação
- Diferentes equipes alocadas melhora a abstração
- Desenvolvedor do componente oferece suporte.

Dificuldades

Dificuldades do desenvolvimento para componentização (construção dos componentes e da infra-estrutura)

- Projetar e preparar um pedaço de software para futura reutilização aumenta a necessidade de flexibilidade, documentação, estabilidade e abrangência do software
- O software deve ser bem documentado, testado e validado, e deve ter um esquema robusto de validação
- Os componentes não podem ser nem muito genéricos e nem muito específicos.

Dificuldades

Dificuldades do desenvolvimento para componentização (construção dos componentes e da infra-estrutura)

- dificuldades do desenvolvimento com componentização:
- esforço despendido no entendimento dos componentes e das ferramentas envolvidas,
- perda de flexibilidade para incluir novas funcionalidades,
- esforço continuado de atualização de suas versões e de reconfiguração do sistema.
- dependência de terceiros, fora do controle dos desenvolvedores
- adaptação do processo de desenvolvimento: incluir etapas como análise de domínio, busca de componentes, testes específicos, etc.

Conceitos Básicos

Porta é um meio identificável de conexão, por onde um componente oferece seus serviços ou acessa os serviços dos outros

Operação, propriedade e evento são exemplos de tipos de portas de um componente. Cada porta tem uma identificação (nome ou número pelo qual a porta é acessada) e pode ser de entrada ou de saída de dados. Cada porta define os tipos de valores que são transmitidos ou recebidos e é normalmente implementada por uma operação

Conceitos Básicos

Conector: meio por onde é feita a conexão entre duas ou mais portas

Implementado por invocação explícita de função, envio de mensagens síncronas ou assíncronas, propagação de eventos, stream de dados, workflow, código móvel, diálogo através de uma API, transferência de arquivo, pipe, propagação de eventos, buffer, sinalização, compartilhamento de arquivo via ftp, etc.

Variam para cada tecnologia e ferramenta adotada: JavaBeans, COM+

Componentes interagem entre si por chamadas de métodos, em um estilo cliente/servidor ou publicador/ouvinte, em tempo de codificação, compilação, inicialização ou execução.

Conceitos Básicos

Interface: um conjunto de portas relacionadas

Define seus pontos de acesso, por meio dos quais outros componentes podem utilizar os serviços oferecidos .

Representa o contrato de utilização do componente. Respeitando-se este contrato, pode-se alterar a implementação interna do componente ou substituí-lo4 por outro, sem modificar quem o utiliza.

O contrato pode cobrir aspectos funcionais (sintaxe e semântica da interface) e não funcionais (referentes à qualidade de serviço).

Permitem separar especificação e a implementação do componente, e desenvolver e substituir componentes de forma transparente para seus clientes

Conceitos Básicos

Interface: várias de acordo com o cliente para diminuir acoplamento.

Pelo menos duas:
- para funcionalidade oferecida a outros componentes
- conexão com a infra-estrutura de execução: serviços técnicos do ciclo de vida, instalação, persistência, etc.

- requerida - se usa uma operação definida naquela interface.
- fornecida - dá suporte a uma interface fornecida se contém uma implementação de todas as operações definidas por aquela interface.

Podem utilizar a implementação de interface da orientação a objetos para implementar o conceito de interface de componente. Propriedade e eventos são mapeados na forma de métodos.

Conceitos Básicos

Instância de componente: conjunto de objetos pelos quais se manipula o componente que são a sua manifestação em tempo de execução.

Deployment: instalação em uma infra-estrutura de execução.

Customização: é a habilidade de adaptar um componente antes de sua instalação ou uso, normalmente com o objetivo de especializar seu comportamento.
- modificação de propriedades (whitebox – detalhes código são liberados)
-composição com outros componentes que especializam determinados comportamentos

Implementação

- definir a estrutura de dados e serviços necessários para descrever o formato de empacotamento usando padrões
- apoiar o empacotamento (desenvolvimento para reutilização
  - Empacotamento permite que ele seja instalado como uma unidade.
  - Mecanismos para o empacotamento diferem de tecnologia para tecnologia. arquivos JAR para e componentes Java, que incluem as classes que implementam os serviços dos componentes, as classes adicionais, etc.
- apoiar a avaliação (desenvolvimento com reutilização) e permitir anotações

Implementação – Modelo de Componentes.

Define vários aspectos da construção e da interação dos componentes:

-como especificar o componente,

-como instanciar o componente,

-quais os tipos de conectores e portas disponíveis,

-qual o modelo de dados que é entendido por todos os componentes,

- como as ligações entre os objetos pertencentes a componentes diferentes são realizadas,

-- como são feitas transações distribuídas,

-- quais são os tipos de interface, as interfaces obrigatórias,

Implementação – Modelo de Componentes.

-como são tratados o catálogo e a localização dos componentes, o despacho das requisições e respostas,

-questões a segurança, interoperabilidade, documentação

-o repositório, o formato, a nomeação, os meta dados, a

-mecanismos de customização, a composição, a evolução,

-o controle de versões,

-a instalação e a desinstalação,

- os serviços de execução, o empacotamento, etc.

Implementação – Modelo de Componentes.

Padrões de composição: o cliente explicitamente chama operações do servidor:

- cliente/servidor
- publicador/ouvinte: o ouvinte se registra como tratador de eventos e informações disponibilizadas pelo publicador.

Exemplos:

- IDL – Interface Description Languages
- CORBA(*Common Object Request Broker Architecture*) – OMG/OMA, base para a construção de modelos especializados.
- CORBAservices – padronização para serviços gerais de SD.
- CORBAfacilities – padronização para serviços horizontais.

Implementação – Modelo de Componentes.

Exemplos de Modelos de Componentes

- Corba Component Model (CCM),
- Microsoft OLE,(D)COM/COM+, DCOM e ActiveX lg Visual Basic. Atualmente .NET
- Sun EJB, JavaBeans : Java Server Faces (JSF) e o modelo Portlets (JSR)
- Webservice: SOAP e XML

Implementação – Modelo de Componentes.

- ***Component kit****:* coleção de componentes que foram projetados para trabalhar em conjunto),

- ***Toolkit***: tipo de SDK (*Software Development Kit*), que apresenta, além dos componentes, um conjunto de ferramentas para criar aplicações para uma determinada plataforma, sistema ou framework.

Exemplos: toolkits de widgets para a linguagem Java:
AWT (*Abstract Windowing Toolkit*), Swing e SWT (*Standard Widget Toolkit*).

Desenvolvimento Baseado em Componentes (DBC)

Metodologia para DBC

1. UML Components -
   J. J.Cheesman and J.Daniels
2. Catalysis (http://www.iconcomp.com/catalysis)
   D. D'Souza and A.A. Wills
3. KobrA
   C.Atkinson et al.

Representação de Componentes

UML: conceito mais amplo

- Um componente é a parte física (feita de bits e bytes) e substituível de um sistema ao qual se adapta e fornece a realização de um conjunto de interfaces. Os componentes são empregados para a modelagem de coisas físicas que podem residir em um nó, como executáveis, bibliotecas, tabelas, arquivos e documentos. Um componente tipicamente representa o pacote físico de elementos lógicos, como classes, interfaces e colaborações.
- Um componente representa um empacotamento físico de elementos relacionados logicamente (normalmente classes)

Componentes – UML x DBC

No DBC o termo componente representa um elemento arquitetural mapeado a um executável, que foi concebido visando a substitutibilidade, a reusabilidade e a interoperabilidade, entre outros fatores.

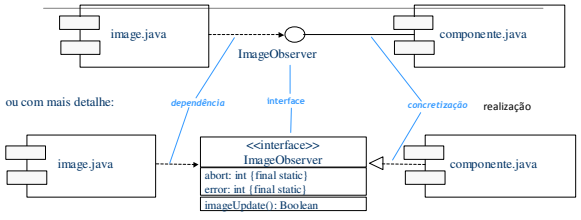
- Mas pode-se utilizar a notação UML e o diagrama de componentes.

Interfaces - UML

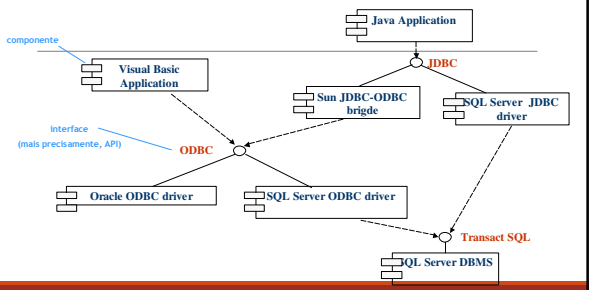
Relação de dependência: um componente pode usar uma ou mais interfaces

- Diz-se que essas interfaces são importadas
- Um componente que usa outro componente através de uma interface bem definida, não deve depender da implementação (do componente em si), mas apenas da interface

Interfaces – Exemplo 1



Interfaces – Exemplo 2



Tipos de dependências entre componentes

Dependências simples:

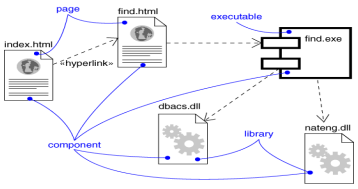
- entre arquivo com código fonte, para controle de alterações
- entre executáveis e/ou bibliotecas, para gestão de configurações e dependências
- entre executáveis ou bibliotecas e tabelas ou documentos de ajuda por eles usados

Dependências estereotipadas:

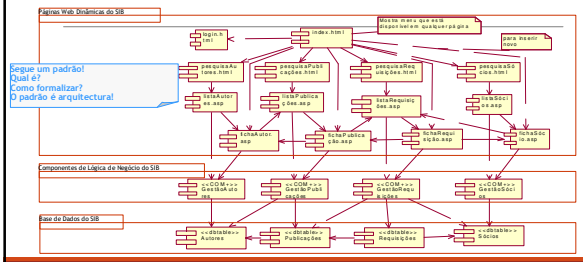
- estereótipo «hyperlink» - entre páginas html ou páginas html e executáveis
- estereótipo «trace» - entre versões consecutivas do mesmo tipo de componente

Não esquecer que é melhor depender das interfaces do que das implementações

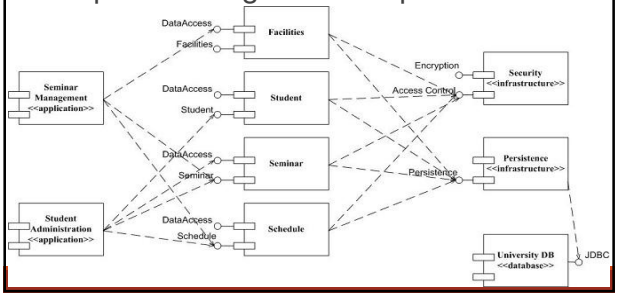
Dependências - Exemplo



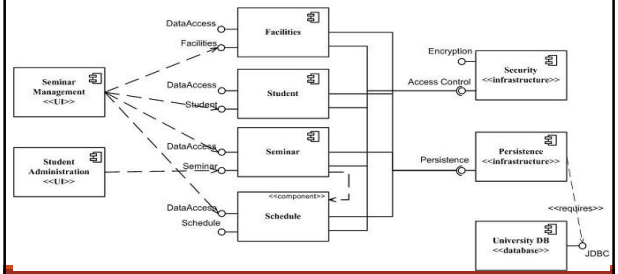
Caso de estudo (biblioteca)



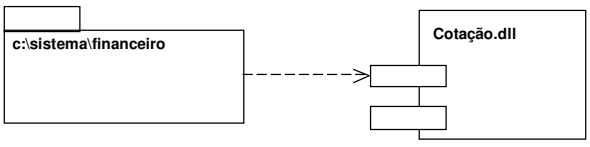
Component Diagram Example in UML 1.4



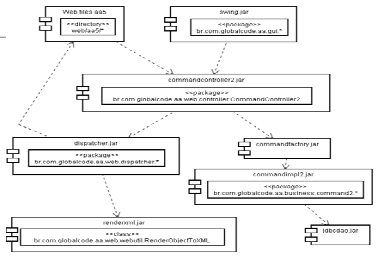
Component Diagram Example in UML 2



Dependência entre pacotes e componentes



Visão Lógica: Visão de Componentes: Exemplo



Componentes e Frameworks

- Um framework provê um conjunto genérico de classes que deve ser completado para instanciar uma aplicação específica, enquanto o uso de componentes possibilita construir um sistema compondo-o a partir de unidades de execução.
- Um framework pode ser instanciado a partir de componentes e um componente pode ser feito a partir de um framework

Exercício

- Dado o modelo desenvolvido para a loja de informática
- Crie a visão de componentes (diagrama de componentes)
- Crie visão de camadas (Não existe diagrama de camadas, usar um diagrama de classes com pacotes para camadas)
- Crie um diagrama de classes com os principais pacotes da aplicação

Bibliografia

Barroca, L., Gimenes, I.M.S. & Huzita, E.H.M. (2005) "Conceitos Básicos", in: Desenvolvimento Baseado em Componentes, Gimenes, I.M.S. & Huzita, E.H.M. (eds), Editora Ciência Moderna, Rio de Janeiro, 2005. ISBN 85-7393-406-9, pg.

D'Souza, D.F. & Wills, A.C. (1998) Objects, Components and Frameworks with UML: The Catalysis Approach. Addison Wesley, ISBN 0-201-31012-0, 1998.

Gerosa,M.A. Desenvolvimento Orientado a Componentes, USP, SP

Gimenes, I.M.S. & Huzita, E.H.M. (2005) Desenvolvimento Baseado em Componentes, Editora Ciência Moderna, Rio de Janeiro, 2005. ISBN 85-7393-406-9.

Szyperski, C. (1997), Component Software: Beyond Object-Oriented Programming, Addison-Wesley, ISBN 0-201-17888-5