Projeto de Software

Karin Becker
Engenharia de Software N
Instituto de Informática – UFRGS

Projeto

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."
- C.A.R. Hoare

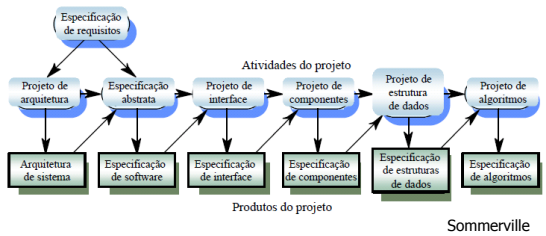
Projeto

- Especificação de requisitos dizem O QUÊ o sistema deve fazer
- Projeto trata de COMO o sistema fará isto
  - Decomposição do sistema
  - Permite dividir o trabalho entre uma equipe de desenvolvedores
  - Determina como os requisitos funcionais serão atingidos, dentro das restrições dos requisitos não funcionais
  - Define plataformas e tecnologias

Níveis de Projeto

- Arquitetural (high level design)
  - Estrutura principal
  - Sistemas – subsistemas
  - Interfaces e dependências
  - Cobre os principais casos de uso
  - Atende os requisitos não funcionais
  - Tende a ser mais difícil de mudar
- Projeto detalhado (low level design)
  - Estrutura interna em termos de módulos/componentes
  - Leva em conta características/idiossincrasias de linguagens de programação
  - Detalhado o suficente para guiar/facilitar a implementação

Projeto



Sommerville

Projeto : Por que é tão difícil?

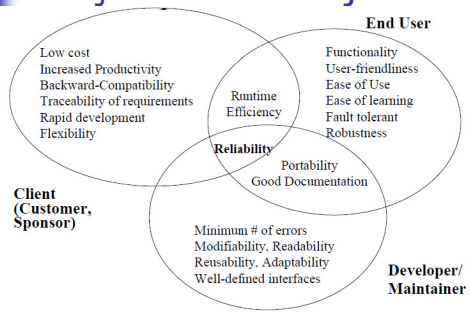
- *Análise:* foca no domínio da aplicação
- *Projeto:* foca no domínio da solução
  - Conhecimento das características do projeto e suas conseqüências evolui
    - Projetar para acomodar mudanças
    - evolução
  - Plataformas tecnológicas têm vida curta : 3 – 5 anos
    - Custo do hardware está baixando
    - Considerações de desempenho mudam com a evolução tecnológica
    - Novas linguagens, novos frameworks, novos serviços
  - Janela de tempo
    - Tempo disponível para decisões
    - Levam em conta o que se sabe (até o momento) e o que se prevê

Objetivos para o projeto

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance
- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum # of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Fonte: Kostas Kontogiannis

Relação entre os objetivos



Fonte: Kostas Kontogiannis

Há que se estabelecer compromissos ...

- Eficiência vs. Portabilidade
- Custo vs Robustês
- Funcionalidade vs. Usabilidade
- Desenvolvimento rápido vs. Funcionalidade
- Custo vs. Reusabilidade
- Compreensibilidade vs. Retrocompatibilidade/Reuso
- etc

Eficiência

- Uso dos recursos em tempo de execução, e seus impactos em termos de tempo de resposta e consumo de memória
- Mais do que algoritmos sofisticados, eficiência tem a ver com a equilibrada distribuição de responsabilidades, e o acoplamento entre componentes
- Eficiência é freqüentemente contraditória com outras propriedades

Interoperabilidade

- Um software freqüentemente deve interagir com outros sistemas que formam seu ambiente
- Arquiteturas que lidem com as dificuldades técnicas de componentes desenvolvidos para diferentes linguagens/plataformas
- Oferecer acessos bem definidos a funcionalidades que devem ser observáveis externamente
- Ex: computador / celular / embarcado

Arquitetura de Software

- Não há definição consensual:
  http://www.sei.cmu.edu/architecture/definitions.html
- Duas definições modernas:
  - [Bass et al., 2003; Wikipedia]: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them".
  - [IEEE, 2000]: "The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution"
- E ainda :
  9 definições "clássicas", 18 definições nos livros, 100+ definições adotadas pela comunidade …

12

Projeto Arquitetural : Atividades Genéricas

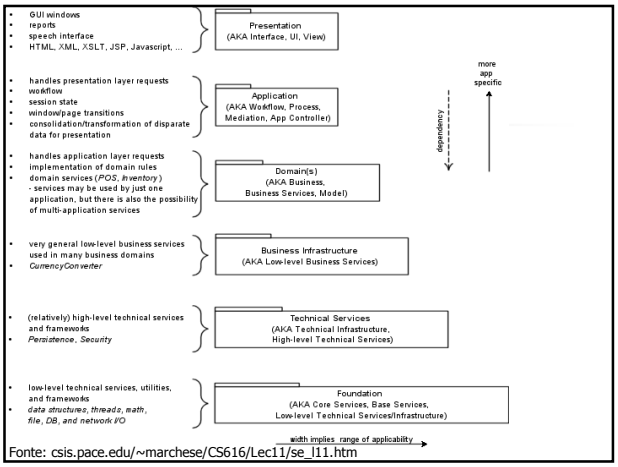
- Estruturação de sistema
  - O sistema é decomposto em vários subsistemas principais e a comunicação entre esses subsistemas são identificados.
- Modelagem de controle
  - É estabelecido um modelo geral dos relacionamentos de controle entre as partes do sistema.
- Decomposição modular
  - Os subsistemas identificados são decompostos em módulos.
  - O projetista deve decidir sobre o tipo de módulo e suas interconexões
- arquitetura de software é tudo que precisa ser mantido consistente no desenvolvimento de software
  - Capacidade de evoluir com o software e seu desenvolvimento
  - Compreensível pela equipe

Decomposição de sistemas

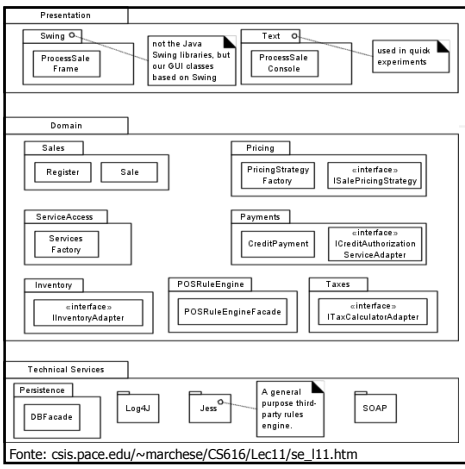
- Subsistemas (UML : Subsistema, Pacote, Componentes)
  - **Organiza componentes de software**
  - **Fonte: Coleção interrelacionadas de classes, associações, operações, eventos e restrições**
- Serviço de um subsistema
  - Responsabilidades genéricas
  - Grupos de operações fornecidas
  - Fonte: casos de uso
- Interface de Subsistema (UML : Interface)
  - Define fluxo de interação e controle entre subsistemas
  - Bem definidos e pequenos
  - Às vezes chamado de API, mas deve ser reservado para a etapa de implementação

Princípios Básicos para Arquiteturas

- Modularização
  - Decomposições de um sistema em grupos de subsistemas e componentes
  - Enpacotamento físico das entidades que constituem a lógica
- Separação de preocupações
  - Isolamento de responsabilidades diferentes ou sem relacionamento entre elas
  - Se um componente desempenha diferentes papéis em distintos contextos, então estes devem ser isolados



Fonte: csis.pace.edu/~marchese/CS616/Lec11/se_l11.htm



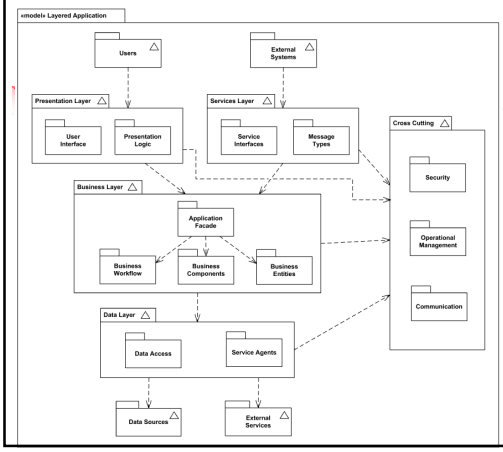
Fonte: csis.pace.edu/~marchese/CS616/Lec11/se_l11.htm

Princípios Básicos para Decomposição

- Abstração:
  - as características essenciais de um componente que permite distingui-lo dos demais, e que portanto permite definir de forma detalhada suaas fronteiras conceituais
  - Melhor tipo de abstração é a de dados
- Ocultamento de informações
  - Esconder a complexidade e detalhes internos um componente de seus clientes, minimizando o acoplamento
- Encapsulamento
  - Agrupamento de elementos em uma abstração que constitui sua estrutura e comportamento
  - Unidade fisicamente distinguível

Princípios Básicos para Decomposição

- Alta coesão
  - Interface dirigida a um único objetivo global
  - Pequenas interfaces
  - A melhor forma de coesão é a funcional
- Baixo acoplamento
  - Poucas interdependências (dependência de poucas interfaces)
  - Acoplamento alto prejudica a compreensão, e manutenção de um sistema
- Todas as dependências são explícitas
- Suficiência, completude
  - em relação à abstração que captura



Princípios Básicos para Decomposição

- Separação de interface e "implementação"
  - Interface
    - define a funcionalidade oferecida e especifica como usá-la.
    - Acessível pelos clientes do componente
  - "Implementação": detalhes internos
    - Decomposição em unidades menores
    - Código + estruturas de dados
  - Proteger o cliente de detalhes desnecessários
  - Dar apoio à capacidade de mudanças

Propriedade Não Funcional

- Quanto maior e mais complexo um sistema, maior seu tempo de vida
- Propriedades não funcionais têm alto impacto na qualidade do software, e na sua capacidade de sobreviver a mudanças

Testabilidade

- Arquitetura deve facilitar a avaliação do bom funcionamento
  - Melhor detecção, isolamento e correção de problemas
  - Planejar e organizar testes ao longo do tempo
  - Integração contínua
  - Facilitar integração de componentes

Capacidade de mudança

- "Manutenção": erros
  - capacidade de localizar as mudanças, minimizar o impacto da mudança nos demais componentes
- Extensibilidade: novas propriedades, versões melhoradas
  - componentes com fraco acoplamento, troca de componentes sem afetar seus cliente
- Reestruturação: reorganização dos componentes e de seus relacionamentos
  - Flexibilidade na configuração
- Portabilidade
  - Adaptação do sistema a novas interfaces, plataformas,
  - Poucas dependências de software/hardware

Capacidade de mudança

- Dá melhor apoio à construção de variantes de configuração de software conforme usuário
- Cuidado com excessos
  - Aumenta a complexidade (desenvolver, manter)
  - Deteriora o desempenho
  - Consome mais recursos
  - Decidir as partes prioritárias a serem projetadas para acomodar mudanças, e quais partes permanecem relativamente mais estáticas
  - Se estas decisões não se revelarem acertadas com o tempo, reestruturar o software para flexibilizar os componentes/relacionamentos críticos
  - É mais barato que projetar total flexibilidade desde o início
- Métodos ágeis costumam não planejar extensivamente para acomodação de mudanças

Escolhendo Subsistemas

- Subsistema
  - Possui bastante autonomia e pouca dependência de outros componentes
  - Pouca interação para cumprir seu objetivo
  - Alta coesão, baixo acoplamento
  - Análise de Dependências
    - Subsistema depende de outros?
    - Depende de quais?
    - Quais dependem dele?
- Pergunta principal
  - Qual é o serviço (papel) do subsistema?
  - O que a sua interface deve oferecer a outros subsistemas?
- Questão complementar
  - Existe alguma ordem/hierarquia entre os subsistemas?
  - Qual o melhor modelo para representar esta ordem/hierarquia?

Estilos/padrões arquiteturais

- Introdução
  - Formas recorrentes observadas entre sistemas diferentes.
  - Ocorrem com freqüência e merecem ser estudadas.
  - Devem satisfazer certas restrições.
  - Um sistema tipicamente não se baseia em um único estilo MAS SIM adota um conjunto de estilos.
- Aspectos que determinam um estil:
  - Componentes que realizam tarefas em tempo de execução
  - Topologia dos relacionamentos entre componentes
  - Conjunto de restrições (no nível semântico)
  - Conectores que mediam a cooperação entre os componentes

27

Padrões

- descrição de módulos/componentes/objetos/classes cooperantes que podem ser adaptados para resolver um problema genérico de projeto em um contexto particular
- nomeia, abstrai e identifica os aspectos chave de uma estrutura comum de projeto com ampla aplicabilidade
  - nome : resume ...
    - problema de projeto
    - solução
    - conseqüências
- independente de domínio
- é muito abstrato para ser codificado
- registro estruturado de EXPERIÊNCIA!!!

Contribuições de Padrões

- padrões auxiliam na construção de aplicações baseado na experiência coletiva de engenheiros de software experientes
  - capturam experiência de desenvolvimento de comprovado sucesso e qualidade
  - auxiliam na disseminação de boas práticas de projeto
- padrões fornecem um vocabulário comum para princípios conhecidos de projeto, facilitando sua compreensão
  - "o editor edix segue uma arquitetura MVC"
  - "use uma arquitetura broker para seu sistema distribuído"

Tipos de Padrões

- Padrões arquiteturais
  - Problemas arquiteturais de maior escala
- Design Patterns
  - Problemas arquiteturais de menor escala
  - Complementam os padrões arquiteturais resolvendo questões pontuais
- Idioms
  - Problemas específicos a linguagens
- Anti-Padrões
  - O que não fazer

Estilos/padrões arquiteturais

- Exemplos de estilos:
  - Cliente-Servidor
  - Camadas
  - SOA – Service-oriented Arquitecture
  - MVC – Model View Controller
  - P2P – peer to peer
  - Dados compartilhados (Blackboard)
  - Filtragem (Pipe)
  - etc.

31

Padrões Arquiteturais

- POSA (Pattern-oriented software Architecture)
  - "da lama à estrutura"
    - camadas
    - pipes
    - quadro-negro (blackboard)
  - Sistemas Distribuídos
    - broker
  - Sistemas Interativos
    - MVC
    - PAC (Presentation-Abstraction-Control)
  - Sistemas Adaptáveis
    - Microkernel
    - Reflexão

[POSA] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, UK, 1996

Camadas

- "A layered system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below." (Garlan and Shaw)
  - cada camada oferece serviços em um determinado nível de abstração
  - em sistemas puros, somente as camadas adjacentes se conhecem

Padrão Camadas: Contexto

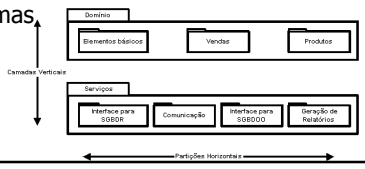
- Você está projetando um sistema cuja característica principal é uma mistura de assuntos de alto nível com assuntos de baixo nível, em que os assuntos de alto nível usam os assuntos de baixo nível.
  - A parte de baixo nível está frequentemente perto do hardware
  - A parte de mais alto nível está frequentemente perto do usuário
  - O fluxo de comunicação tipicamente consiste de pedidos fluindo do alto para o baixo níveis
  - As respostas andam na direção contrária

Padrão Camadas: Solução

- Organizar a estrutura global do sistema em camadas distintas, com uma separação coesa de responsabilidades, tal que os níveis mais baixos sejam de serviços gerais, e os níveis mais altos, mais relacionados a uma aplicação específica
- Colaboração e acoplamento é dos níveis mais altos para os níveis inferiores
  - O contrário deve ser evitado

Padrão Camadas: Solução

- Decomposição em partições e camadas
  - Uma camada é um subsistema que adiciona valor a subsistemas de menor nível de abstração
  - Uma partição é um subsistema "paralelo" a outros subsistemas



Camadas: Implementação

- 1. Defina o critério de abstração para agrupar tarefas em camadas;
- 2. Determine o número de níveis de abstração (baseado no critério acima)
  - Cada nível de abstração corresponde a uma camada
  - Decisão não é fácil
    - Camadas demais podem afetar o overhead
    - Camadas de menos comprometem a estrutura
- 3. Dê um nome e atribua tarefas a cada camada
  - Para a camada de cima, a tarefa é a tarefa global do sistema, do ponto de vista do usuário

Camadas: Implementação

- 4. Especifique os serviços
  - O princípio básico é de separar as camadas uma das outras
  - Nenhum módulo abrange duas camadas
  - Tente manter mais riqueza acima e menos abaixo
- 5. Refine as Camadas
- 6. Especifique uma interface para cada camada
  - A camada N nada sabe sobre a camada N-1 e usa uma interface para acessá-la;
  - Pode usar o padrão Façade para organizar a interface;
- 7. Estruture as camadas individuais
  - Quebre a camada em subsistemas (partições) menores se ela for complexa;

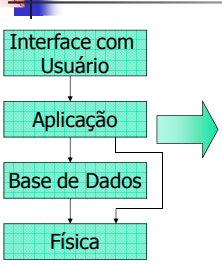
Camadas: Implementação

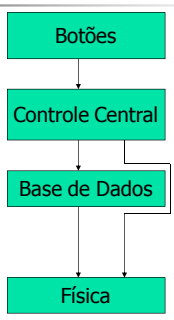
- 8. Especifique a comunicação entre camadas
  - A camada N passa a informação necessária para a camada N-1 ao chamá-la
- 9. Desacople camadas adjacentes
  - Evite situações em que a camada de baixo sabe algo sobre seus clientes (camada acima)
    - Acoplamento unidirecional é preferível

Camadas: Implementação

- **10. Projete uma estratégia de tratamento de erros**
  - O esforço de programação e overhead podem ser grandes para tratar erros numa arquitetura em camadas
    - Um erro pode ser tratado na camada onde foi descoberto ou numa camada superior
    - No segundo caso, deve haver mapeamento para tornar o erro semanticamente reconhecível pela camada de cima;
    - Tente tratar erros na camada mais baixa possível
    - Isso simplifica todas as camadas superiores que não sabem, nem devem saber, do erro;

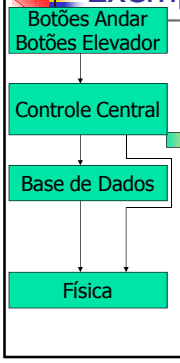
Exemplo: Controle de Elevador

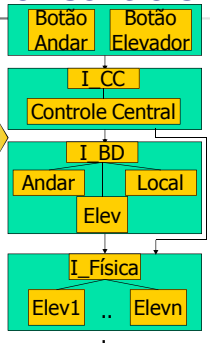




- Chamada em algum andar (sobe/desce)
- Solicita movimentação para algum andar
- Processamento
- Tomada de decisão:
  - subir, descer, abrir a porta, ...
- Armazena dados:
  - onde está cada elevador, andares com chamadas, chamadas por elevador
- Sistema Operacional
- Movimento elevadores

Exemplo: Controle de Elevador





- Chamada em algum andar (sobe/desce)
- Solicita movimentação para algum andar
- Processamento
- Decisão: subir, descer, abrir a porta, ...
- Andar(andar, sobe, desce)
- Local(elev, andar)
- Elev(elev, chama_andar)
- Interface única (fachada)
- Sistema Operacional
- Movimento elevadores

Camadas

- Vantagens
  - Separação de preocupações – níveis diferenciados de abstração ao longo das camadas
  - Partição de um problema mais complexo em problemas mais simples
  - Facilidade de manutenção devido ao isolamento em camadas
  - Padronização devido à definição das camadas (e.g. OSI)
- Desvantagens
  - Estruturação em camadas nem sempre é aplicável
  - Desempenho – sobrecarga devido à comunicação limitada a níveis contíguos

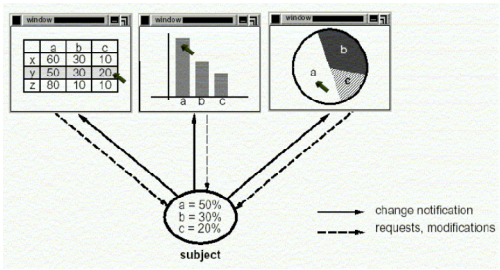
Sugestão de leitura:
www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/camadas.html

Model-View-Controller (MVC)

- Facilita a obtenção de múltiplas visões dos mesmos dados.
- Desacopla a interface da lógica da aplicação.
  - Modelo mantêm os dados e contém lógica do negócio.
  - Visão é responsável pela apresentação dos dados.
  - Controlador trata os eventos que afetam dados e visão
- A visão trabalha em par com o controlador
- Mecanismo de controle de propagação de mudanças no modelo às respectivas visão e controlador

44

MVC



Model-View-Controller

- Forças
  - a mesma informação pode ser apresentada simultaneamente de forma diferente
  - interface e semântica devem estar consistentes
    - a exibição e o comportamento da aplicação deve refletir imediatamente mudanças ocorridas
  - mudanças na interface devem fáceis de executar, se possível em tempo de execução
  - deve ser possível portar a aplicação para diferentes plataformas/novas versões sem afetar o núcleo da aplicação

Em resumo ...

- Assim como o bom projeto, uma boa arquitetura
  - Resulta da aplicação de um conjunto de técnicas e princípios aplicados consistentemente em diferentes fases do projeto
  - Resiliente face às inevitáveis mudanças
  - Norteia o desenvolvimento ao longo da vida do produtos
  - Reusa conhecimento consolidado de engenharia
- Padrões, estilos arquiteturais são soluções cujo valor já foi comprovado
  - Reuso de experiência
  - Padrões para problemas de grande escala (arquiteturais) e pequena/média escala (projeto)

Para saber mais ...

- Recomendada
  - Sommervile, Ian. Engenharia de software. 8ª edição. Pearson Education. São Paulo:, 2007.
    - Capítulo 10 – Projeto de Arquitetura
    - Capítulo 11 – Projeto Detalhado
- Literatura Opcional
  - F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, UK, 1996
  - M. Shaw and D. Garlan. *Software Architecture: Perspectives on a Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996