Facetas da Reusabilidade de Software

- Daremos um breve panorama da disciplina inteira: reusabilidade de software

Qual é o problema?

- Fazer software é difícil
- Fazer software é lento e caro
- Não temos tecnologia ainda para fazer software grande do zero, rapidamente e com poucos bugs

Qual é a solução?

- O reuso é "o" caminho mais freqüentemente apontado
  - As mesmas idéias básicas devem ter sido projetadas e reprojetadas pela mesma pessoa ou pessoas diferentes muitas vezes, com certeza!
  - Será que temos que começar tudo de novo, sempre???

Por que a solução não é fácil?

- Reuso não acontece automaticamente!
- Para reusar, tem que:
  - Bolar boas abstrações (i.e., abstrações úteis); e
  - Empacotá-las para facilitar o reuso
- Como reutilizar bem tem sido meio misterioso ao longo dos anos
  - Como fazer isso?
  - Que técnicas usar?
- Orientação a Objeto prometeu muito mas não cumpriu a promessa
  - Porque não é fácil
  - Pouca gente conseguiu bons resultados
  - Ficou mais no marketing (com exceção de alguns bons programadores)

- Por que é difícil?
  - Sugestões de William Opdyke em "Refactoring, Reuse, and Reality":
    - Técnicos podem não entender *o que* reusar ou *como* reusá-lo;
    - Técnicos podem não se motivar a aplicar técnicas de reuso a não ser que benefícios de curto prazo sejam alcançáveis (porém, o reuso é mais um investimento em longo prazo);
    - Overhead, curva de aprendizado e custos de descobrimento devem ser endereçados para que o caminho do reuso seja bem sucedido.

O que mudou recentemente?

- Na metade da década de 90, as lições e caminhos cristalizaram-se com:
  - Patterns (principalmente Design Patterns)
  - Frameworks
    - Frameworks são mais antigos, datando do final de 80 mas explodiram apenas recentemente devido ao casamento com design patterns
  - Componentes
    - Só recentemente (digamos desde 1999), estamos saindo do uso de componentes apenas na interface gráfica

Tipos de reuso

- Falaremos das técnicas historicamente empregadas para atingir o reuso
  - Do mais simples/antigo para mais sofisticado/novo
  - A *granularidade* de reuso mudou radicalmente ao longo dos anos

Fase infantil (500 AC até anos 1960)

- Reuso da solução do vizinho numa prova :-)

Fase pré OO: 1960-1970

- O que é reutilizado: *linhas de código roubadas* de um programa e usadas em outro
  - Nome (moderno) da técnica: Cut-and-Paste
- O que é reutilizado: *código comum* de um programa
  - Nome da técnica: Subrotinas
- O que é reutilizado: *funções inteiras genéricas*, relacionadas, para servir em várias situações em programas diferentes
  - Nome da técnica: Bibliotecas

Fase da revolução O-O: 1980

- O que é reutilizado: *conceitos inteiros* através de classes
  - Nome da técnica: herança, composição/delegação
  - Permite fazer *Programming-by-Difference*
- O que é reutilizado: *Contratos de relacionamento* entre objetos
  - Nome da técnica: Interfaces (originalmente chamado *Protocolo*)
  - Interface como conceito de "Plug Point"
  - Interface como conceito de "barreira de desacoplamento"
    - Uma lei fundamental da programação: "*Program to an Interface, not to an Implementation*"
  - Classes abstratas são usadas para criar interfaces em algumas linguagens, mas deve-se diferenciar bem dois conceitos que classes abstratas podem implementar:
    - Classes puramente abstratas definem uma interface (um tipo abstrato de dados);
      - Herança de *tipo*
    - Classes abstratas com alguma implementação são usadas para fatorar código comum

        - Herança de *implementação*
    - O primeiro conceito é o mais importante, já que o segundo pode ser feito com composição/delegação
    - Lembre: classe = implementação, interface = contrato
    - Em Java:
        - Classes abstratas são usadas apenas para herança de implementação
        - Interfaces são usadas para herança de tipo
- Late binding (polimorfismo) permite melhor reuso pois desvincula o uso da implementação
    - Qualquer implementação da interface pode pintar em tempo de execução e ser usada
    - Se fôssemos forçados a escolher (isto é, fazer o binding) em tempo de compilação, teríamos amarração a uma implementação (classe) particular;
    - Com late binding, mais objetos podem trabalhar uns com os outros, pois eles sabem menos coisas uns dos outros

Situação no final da década de 1980: OO resolveu?

- Porém, no final da década de 80, as técnicas acima ainda não cumpriram a promessa de reuso completamente
- Quais são as coisas que ainda dificultam a reutilização de software?
    - São situações novas que forçam um redesign
        - Exemplos seguem
    - Criar um objeto especificando sua classe explicitamente
        - O código se compromete com uma implementação particular
    - Dependência de operações específicas

- Especificar uma operação particular diz *como* fazer algo e não apenas qual é o efeito desejado
- É melhor "programar por intenção" dizendo "o quê" e não "como"

- Dependência de plataformas de hardware ou software específicas
  - As APIs são diferentes
- Dependência da representação ou implementação de objetos
  - Clientes que sabem como um objeto é representado, armazenado, localizado ou implementado terão que mudar se o cliente mudar
- Dependências algorítmicas
  - Objetos que dependem de um algoritmo devem mudar quando o algoritmo é estendido, otimizado ou trocado durante a manutenção
- Acoplamento forte
  - O reuso isolado de classes que têm acoplamento forte é freqüentemente impossível
  - No extremo, temos um sistema monolítico (puxa um fio e tudo vem atrás)
- Extensão de funcionalidade através da herança
  - Cria um forte acoplamento entre a superclasse e as subclasses
- Dificuldade de alterar classes convenientemente
  - Pode não ter código fonte ou não querer mudar uma classe

Fase: após o "deslumbramento OO": 1990

- Técnicas foram inventadas e catalogadas para "resolver" os problemas mencionados acima
- Catalogação das "boas idéias de projeto" que os melhores programadores do mundo tiveram ao longo dos anos

- São micro-arquiteturas que definem as *colaborações entre classes e objetos*
  - Chamados *Design Patterns (Padrões de Projeto)*
  - A granularidade de reuso está aumentando de "classes individuais" (ou hierarquias de classes) para "várias classes e suas colaborações"
- Exemplo: Padrão *Iterator* para varrer um objeto sem saber sua implementação
- Não se reutiliza código aqui, só *idéias*
- São soluções boas aos problemas de reuso apresentados acima
- Meta-padrão: Design patterns especificam formas de separar e isolar o que é igual do que é diferente entre situações
- São soluções boas aos dois problemas: "Bolar boas abstrações" e "Empacotá-las para facilitar o reuso"
  - Resolvem particularmente bem o segundo problema

- Aumentando ainda mais a granularidade: frameworks
  - Para reusar: análise, arquitetura, design inteiro, semântica de interação e até testes!
    - É *muito* reuso!
  - Imagine analisar, projetar, implementar e testar várias aplicações semelhantes
    - De um mesmo *Domínio de Problema*
  - Após implementar e testar as várias aplicações, fatora o que elas têm de comum num código único e deixe *ganchos* para enxertar as diferenças
  - Você acabou de criar um *framework*
  - Frameworks estão explodindo no final da década de 90 pois ficou mais claro como criá-los usando Design Patterns
    - Junção das duas técnicas
- Reutilização de abstrações inteiras plugáveis
  - Componentes
    - Cuidado! A palavra é usada de várias formas!

- De forma geral, significa alguma abstração plugável
- Para nós, a semântica é um pouco mais restrita: Objetos manipuláveis em *Design Time*

- Novo conceito: *Design Time*
  - Para se somar a *Compile Time* e *Execution Time*
- Queremos, em tempo de design, instanciar abstrações (objetos, talvez?), configurá-los e utilizá-los num programa
  - Uso freqüente de ferramentas visuais para criar componentes
- Apareceu para bolar interfaces gráficas mais rapidamente
  - Pioneirismo de Visual Basic
- Componentes permitiram a revolução chamada RAD (*Rapid Application Development*)
- Hoje: uso estendido para "Server Components" que nem sequer têm interface gráfica

Palavras finais: Implantação da cultura de reuso nas empresas

- Nessa nova cultura, uma empresa se foca em construir e melhorar seu *patrimônio de reuso*
- Isto deve ser encarado como *investimento*
  - Um conceito novo na área de desenvolvimento de software
  - Generalizar algo para ser reutilizado não se justifica se for considerado apenas o objetivo original proposto
  - Requer o alargamento dos requisitos originais
- Requer coisas novas:
  - Um processo de desenvolvimento adequado
    - Além de desenvolver um produto, queremos criar abstrações reutilizáveis e aumentar o patrimônio da empresa

- Novos papéis
  - Para coordenar o reuso entre times diferentes
- Treinamento
- Incentivos apropriados para o reuso
  - Por que atrapalhar meu cronograma só para que o que faço seja reutilizado por *outra equipe* depois?

programa