Padrões e Desenho de Software

Resumos
2015/2016

João Alegria | 68661

Princípios GRASP

- General Responsibility Assignment Software Patterns
  - Nome escolhido para sugerir a importância de compreender os princípios fundamentais para projetar com sucesso software orientado a objetos.
  - Descreve os princípios fundamentais do design de objetos e responsabilidades.
- Exemplo:
  - Atribuir a responsabilidade de uma classe
  - Evitar / Minimizar dependências adicionais
  - Maximizar a compreensibilidade

Criador (Creator)

Problema: Quem cria uma instância em A?

Solução: Atribuir à classe B a responsabilidade de criar uma instância da classe A, se uma desta se verificar (quanto mais melhor):

- B contém ou agrega A
- B regista A
- B usa A
- B tem os dados de inicialização para A que passarão ao construtor de A

Discussão do padrão Creator:

- Promove baixo acoplamento, fazendo instâncias de uma classe responsável para a criação de objetos que são necessários para a referência.
- Conecta um objeto para o seu criado quando:
  - Há uma relação de agregação
  - Há uma relação de conteúdo
  - Inicialização de dados passados durante a criação

Contra Indicações ou Advertências:

- A criação pode exigir uma complexidade significativa
  - Reciclar instâncias devido a razões de desempenho
  - Criando condicionalmente instâncias de uma família de classes semelhantes

Informação Especialista (Information Expert)

Problema: Como atribuir responsabilidades a objetos?

Solução: Atribuir a responsabilidade à classe que tem informações necessárias. Atribuir a responsabilidade ao especialista: a classe que tem as informações necessárias para assumir a responsabilidade.

Benefícios: O encapsulamento da informação é mantida uma vez que os objetos usam os seus próprios dados para realizar as tarefas. Isto leva a uma baixo acoplamento entre classes.

- Facilita a informação do encapsulamento
    - Classes usam a sua própria informação para cumprir tarefas
        - Classes altamente coesas
    - Código de fácil percepção
- Promove o baixo acoplamento

MAS

- Pode tornar uma classe excessivamente complexa
- Exemplo: Quem é o responsável por guardar *Vendas* na base de dados? *Vendas* é a 'Information Expert', mas com esta decisão cada classe tem os seus próprios serviços a guardar na base de dados. -> É preciso outro tipo de separação (domínio e persistência).

Baixo Acoplamento (Low Coupling)

Problema: Como reduzir o impacto da mudança e incentivar a reutilização?

Solução: Atribuir a responsabilidade de modo que o acoplamento (entre classes) permaneça baixo. Tentar evitar que uma classe tenha de saber muitos sobre os outros.

- Mudanças são localizadas
- Mais fácil de entender
- Mais fácil de reutilizar

Classes com forte acoplamento:

- Sofrem com as mudanças de classes relacionadas
- São mais difíceis de perceber e sustentar
- São mais difíceis de reutilizar

-> Mas o acoplamento é necessário se quisermos trocar mensagens entre classes

Ponto Chave: "Information Expert" favore o "Low Coupling". O Information Expert pede-nos que entremos o objeto que tem maior parte da informação necessária para assumira a responsabilidade.

Nas programações orientadas a objetos, as formas mais comuns de acoplamento do TipoX para o TipoY incluem:

- O TipoX tem um atributo (membro da memória ou uma variável instanciada) que se refere a uma instancia do TipoY ou mesmo a TipoY.
- O TipoX tem um método que se refere a uma instancia do TipoY, ou até mesmo a TipoY por quaisquer meios. Estes incluem tipicamente um parâmetro ou variável local do TipoY, ou um objeto retornado de uma mensagem a ser instância do TipoY.
- TipoX é uma superclasse direta ou indireta de TipoY.
- TipoY é uma interface e TipoX implementa essa interface.

Benefícios e Contra Indicações:

- Compreensibilidade: As classes são mais fáceis de estudar de forma isolada
- Manutenção: As classes não são afetadas por mudanças em outros componentes
- Reutilização: Mais fácil de agarrar as classes

MAS

- Um grande acoplamento às classes estáveis não é um grande problema
- Exemplo: Bibliotecas e classes bem testadas

- Classes que, por natureza, são genéricas e têm alta probabilidade de reutilização deveriam ter acoplamento baixo
- O caso extremo do baixo acoplamento é o não acoplamento: contraria o princípio da orientação a objetos: objetos conectados, trocando mensagens entre si.
- O acoplamento alto não é problema em si. O problema é o acoplamento a classes que, de alguma forma são instáveis: sua interface, sua implementação ou a sua mera presença.

Alta Coesão (High Cohesion)

Problema: Como manter as classes focadas e geridas? (e consequentemente com baixo acoplamento)

Solução: Atribuir responsabilidades para que a coesão continue elevada.

Benefícios e Contra Indicações:

- Compreensibilidade, manutenção
- Complementa baixo acoplamento

MAS

- As vezes é desejável criar objetos menos coesos
    - Que forneça uma interface para muitas operações, devido à necessidade de desempenho associados aos objetos remotos e comunicação remota.

Controlador (Controller)

Problema: Quem deve ser responsável pelos UI?

Solução: Se um programa receber eventos de fontes externas que não do seu GUI, adicionar uma classe para dissociar a fonte do evento dos objetos que fazem mesmo parte. Atribuir a responsabilidade de manipulação de uma mensagem do sistema para uma classe que represente uma destas opções:

- Os negócios ou o “sistema” global (controlo de Erro - façade controller)
- Uma classe artificial, ‘Pure Fabrication’ representa o caso de uso (um controlador de use cases).

Benefícios e Contra Indicações:

- Aumento do potencial de reutilização
  - Usando um objeto controller, mantém as fontes dos eventos externos e internos do tipo e do comportamento de cada um.
  - Tanto as classes UI como as de domínio do problema / software podem ser alterados sem afetar o outro lado.
- Controller apenas encaminha
  - Pedido de manipulação de evento
  - Pedidos de Output
- Motivo sobre o estado dos Use Cases
  - Garantir que as operações do sistema ocorrem em sequência lógica, ou para ser capaz de raciocinar sobre o estado atual da atividade e operações dentro do use case.

Polimorfismo (Polymorphism)

Problema: Como lidar com o comportamento no tipo (ou seja, classe), mas não com as instruções "if-then-else" ou "switch" que envolva o nome da classe ou um atributo?

Solução: - Quando comportamentos alternativos são selecionados com base no tipo de um objeto, usar o método polimórfico para selecionar o comportamento, ao invés de usar a instrução "if" para testar o tipo.

- Métodos polimórficos: Dar o mesmo nome a serviços em classes diferentes. Serviços são implementados por métodos.

Benefícios e Contra Indicações:

- Mais fácil e mais confiável do que usar a lógica de solução explícita
- Mais fácil de adicionar, posteriormente, comportamentos adicionais

MAS

- Aumenta o número de classes do design
- Pode tornar o código mais complicado de entender

Pura Fabricação (Pure Fabrication)

Problema: - Que objeto deve ter a responsabilidade quando nenhuma classe do domínio do problema pode levá-lo sem violar a alta coesão e baixo acoplamento?

- Nem todos as responsabilidades ajustam-se em classes de domínio, como persistência, comunicação de rede, interação do utilizador, etc…

Solução: Atribuir um conjunto altamente coeso de responsabilidades para uma classe artificial que não representa nada no domínio do problema. (Uma classe fictícia que possibilite alta coesão, baixo acoplamento e reutilização)

Benefícios e Contra Indicações:

- Alta coesão é suportada porque a responsabilidade são fatoradas numa classe que só incide sobre um conjunto muito específico de tarefas relacionadas.
- A reutilização pode ser aumentada devido à presença de classes "Pure Fabrication".

Indireção (Indirection)

Problema: - Como evitar o acoplamento direto?

- Como desacoplar objetos para que o baixo acoplamento é suportado e o potencial alto?

Solução: Atribuir a responsabilidade a um objeto intermédio para mediar entre outros componentes ou serviços, para que eles não são diretamente acoplados.

Benefícios e Contra Indicações:

- Baixo acoplamento
- Promove a reutilização

Variações Protegidas (Protected Variations)

Problema: Como projetar objetos, subsistemas e sistemas, para que variações ou instabilidades nos elementos não tenham um impacto indesejável sobre os outros elementos?

Solução: Identificar pontos de variação previstos ou imprevistos, atribuir responsabilidades para criar uma interface estável em torno deles.

Protected Variations é um princípio fundamental de design que é a base para muitos padrões de design.

Mecanismos motivados por Protected Variations:

- Mecanismos PV: Encapsulamento de dados, interfaces, polimorfismo, indicação, padrões
- Data-Driven designs: “style sheets”, ficheiros pessoais, outros mecanismos para leitura de dados de configuração em tempo real
- Acesso uniforme