Single responsibility principle

Primeiro, "generalizemos" o princípio para aplicá-lo a desenvolvimento de *software* em linhas gerais, não só para POO.

O Django tem uma tendência natural a esse princípio, e aproveitamos isso no projeto: O *framework* preconiza que cada parte de um site seja um *app* ao estilo *plug-and-play*, ou seja, que possa ser desenvolvido à parte e inserido em qualquer projeto em que faça sentido.

Como exemplos:

- `gmailbox` é um *backend* de emails que permite usar uma conta do gmail para mandar emails do site (em outras palavras, um administrador associa uma conta existente do gmail como sendo o "email do site"). A partir daí, emails de cadastro e similares serão enviados (por baixo dos panos) lançando mão da tal conta
- `emailsignup` é uma complementação à estratégia de cadastro de usuários do django. O *app* exige que o usuário registre seu email, manda uma ativação para ele e exige que esse link de ativação seja clicado para que a conta seja efetivamente ativada.
- `chatchannels` é um pacote para criar "canais" de chat individualizados. Cada canal tem seus "administradores", pode ser público (todos podem entrar) ou privado (somente os "participantes permitidos" entram) e é persistente, ou seja, as mensagens são permanentemente guardadas.
- `chessgames` é um pacote que define `models` (jargão do django) para armazenar jogos de xadrez na base de dados e disponibiliza `consumers` (jargão do django-channels) websocket para que dois usuários possam jogar um contra o outro.

O SRP consiste no fato de cada *app* ser mínimo, realizando uma única função bem definida

Open closed design

Não fomos nós que tivemos a ideia (já que o código usado é atribuído parcialmente a um terceiro), mas o pacote `emailsignup` ilustra o OCD.

Lá, a intenção é adicionar a exigência de email+ativação ao fluxo usual de autenticação que o django já provê. A solução, que se encaixa no princípio do "aberto-fechado", é definir uma subclasse de `UserCreationForm` (vinda do django) e, na subclasse, adicionar a lógica de emails e ativação (confira `emailsignup.forms`).

Liskov substitution principle

Este princípio é menos um "como fazer as coisas" e mais um "como não fazer as coisas". Por exemplo, considere um método que espera argumento de uma classe qualquer `Game` com um método `getPoints(): number`. Se passarmos uma instância de `SoccerGamer` para ele, ele vai chamar o método e continuar o processamento. O princípio foi obedecido, mas isso não diz muito (se fosse em Java, sugere que o código compilou, pelo menos).

Assim sendo, poderia ser dito que o princípio foi obedecido no exemplo anterior sobre o `UserCreationForm`, no qual se espera que o django também obedeça ao princípio para que o comportamento, em tempo de execução, seja como esperado pela relação estática entre as classes envolvidas.

O ponto aqui é: difícil achar uma contrução sobre a qual se diga "Ah! Aqui foi usado o princípio de Liskov", porque já se supõe que o tal princípio seja 100% obedecido em toda situação (sob pena de a estrutura de classes perder sentido se não for respeitado).

Interface segregation principle

Este princípio muito se aproxima do *Single responsibility principle*, já que infringir o segundo implica infringir o primeiro (se uma

classe faz coisa demais, ela tem métodos demais, isto é, uma interface gigante).

Para este princípio, aconteceu algo no meio do projeto que (pensamos) ilustra bem sua utilidade:

Considere que o pacote `django-channels`, usado para manipular conexões *websocket* entre servidor e cliente, disponibiliza uma classe `JsonWebsocketConsumer` com métodos genéricos como `send_json` e `receive_json` para conversar com o cliente (mandando e recebendo dicionários). Nós, ao utilizarmos o pacote, definimos uma subclasse daquela, denominada `ChessGameConsumer`, adicionando métodos específicos do contexto "jogo de xadrez".

Agora o ponto de interesse: Existe um "protocolo" de comunicação para o jogo de xadrez (inventado por nós e descrito em `chessgames/README.md`), que define a estrutura dos dicionários que o cliente pode mandar para o servidor e vice-versa. O uso do *interface segregation principle* consistiu em separar a estrutura particular dos dicionários para fora da `ChessGameConsumer`: Essas estruturas foram parar em `ServerMsgs`.

Com isso, a `ChessGameConsumer` não mais precisou definir quais chaves cada dicionário deveria ter, pois isso já era automaticamente produzido pela `ServerMsgs`. Por exemplo, se `ChessGameConsumer` quisesse mandar, para o cliente, mensagem avisando que o jogo acabou, ela poderia chamar `ServerMsgs.game_end(win='white')`, e esse método construiria o dicionário internamente.

Dependency Inversion principle

Considere que o princípio de inversão de dependência pressupõe:

- A existência de uma classe (um "componente", de modo mais geral) que requeira uma outra classe de "nível mais baixo".
- Essa outra classe de "nível mais baixo", que pode ser uma conexão com uma DB PostgreSQL (por exemplo), poderia muito bem ser trocada por uma interface menos específica (portanto mais asbtrata). No caso da conexão com a DB, em vez de usar a classe `PostgreSQLConnection`, seria usada uma abstrata `DBConnection`.
- A primeira classe (aquela de mais alto nível) funcionaria muito bem com a versão abstraída (com a `DBConnection`, no exemplo).

No projeto, o exemplo da `ServerMsgs` (seção anterior) se encaixa como inversão de dependência: Em vez de `ChessGameConsumer` depender diretamente de especificar os campos de dicionários manualmente (baixo nível), ela depende da abstração `ServerMsgs`, cujo tratamento de dicionários é ocultado de `ChessGameConsumer`. Se o funcionamento do pacote `django-channels` mudasse de forma a usar outras estruturas de dados diferentes de dicionários, a `ChessGameConsumer` não precisaria mais mudar.

Outra situação que se presta como exemplo de inversão de dependência (desde que pensemos menos em classes e mais na noção difusa de "componentes") é a maneira como o *app* `gmailbox` disponibiliza uma conta do gmail para que o restante do projeto possa enviar emails por ela: O django não obriga nenhum componente a depender diretamente de algum serviço de email (ou seja, ninguém depende diretamente de `GmailBackend`, a classe implementada por `gmailbox`). Em vez disso, django define uma abstração `EmailBackend` que é invariante a implementações concretas de envio de emails (seja SMTP, gmail, Mailgun ou outros serviços). A escolha concreta do serviço é feita "por baixo dos panos" nos arquivos de configuração do projeto, e nenhuma outra parte do projeto precisa ser modificada caso resolvamos trocar de gmail para outro fornecedor qualquer.