Alejandro Taylor Sanjuan
Carlos Rangel

Código Origianl:

```
class Answer(object):

        ...

    def positive_votes(self):
        r = []
        for vote in self.votes:
            if vote.is_like():
                r.append(vote)
        return r

    def negative_votes(self):
        r = []
        for vote in self.votes:
            if not vote.is_like():
                r.append(vote)
        return r

    def add_vote(self, a_vote):
        if any(vote.user == a_vote.user for vote in self.votes):
            raise ValueError("Este usuario ya ha votado")
        self.votes.append(a_vote)

class Question (object):

        ...

    def positive_votes(self):
        r = []
        for vote in self.votes:
            if vote.is_like():
                r.append(vote)
        return r

    def negative_votes(self):
        r = []
        for vote in self.votes:
            if not vote.is_like():
                r.append(vote)
        return r

    def add_vote(self, a_vote):
        if any(vote.user == a_vote.user for vote in self.votes):
            raise ValueError("Este usuario ya ha votado")
        self.votes.append(a_vote)
```

Código Refactorizado:

```
class Votable:
    def __init__(self):
        self.votes = []

    def positive_votes(self):
        return [vote for vote in self.votes if vote.is_like()]

    def negative_votes(self):
        return [vote for vote in self.votes if not vote.is_like()]

    def add_vote(self, a_vote):
        if any(vote.user == a_vote.user for vote in self.votes):
            raise ValueError("Este usuario ya ha votado")
        self.votes.append(a_vote)

class Answer(Votable):

class Question(Votable):
```

Alejandro Taylor Sanjuan
Carlos Rangel

Descripción de cambios:

El código original hay una **duplicación de código**, ya que las clases Answer y Question replican la misma lógica para manejar votos positivos, negativos y añadir un voto. Este tipo de duplicación hace que el mantenimiento sea más difícil y propenso a errores. El código refactorizado mejora esto al crear una clase base Votable que centraliza la lógica común. Esto elimina la duplicación, mejora la legibilidad, facilita el mantenimiento y hace el código más extensible al permitir la creación de nuevas clases que manejen votos sin repetir la misma lógica.

Alejandro Taylor Sanjuan
Carlos Rangel

Código Original:

```
def calculate_score(self):
    score = 0
    for question in self.questions:
        pv = len(question.positive_votes())
        nv = len(question.negative_votes())
        if pv > nv:
            score += 10
    for answer in self.answers:
        pv = len(answer.positive_votes())
        nv = len(answer.negative_votes())
        if pv > nv:
            score += 20
    return score
```

Código Refactorizado:

```
def calculate_score(self):
    score = 0;
    score += self._sum_votable_score(self.questions, 10)
    score += self._sum_votable_score(self.answers, 20)
    return score


def _sum_votable_score(self, votable, points):
    score = 0
    for item in votable:
        if len(item.positive_votes()) > len(item.negative_votes()):
            score += points
    return score
```

Descripción de cambios:

Una vez más hay una **duplicación de código**, ya que se repite la misma lógica tanto para las preguntas como para las respuestas con la única diferencia de la puntuación aplicada a cada uno, lo que aumenta la posibilidad de errores si se necesita modificar dicha lógica en el futuro. Además, las **variables con nombres poco descriptivos**, como pv y nv, no son claros sobre lo que representan, dificultando la comprensión del código. También se hace uso de **variables temporales innecesarias**, como pv y nv, para almacenar los contadores de votos, cuando realmente solo se necesita hacer una comparación directa entre los votos positivos y negativos. Al refactorizar el código y mover esta lógica a la función _sum_votable_score, se elimina la duplicación, se mejora la claridad y se hace más eficiente al evitar las variables temporales, lo que resulta en un código más limpio, legible y fácil de mantener.

Alejandro Taylor Sanjuan
Carlos Rangel

Código Original:

```
class CuOOra:
    def __init__(self):
        self.questions = []

    def add_question(self, a_question):
        self.questions.append(a_question)

    def get_social_questions_for_user(self, user):
        social_retriever = QuestionRetriever.create_social()
        return social_retriever.retrieve_questions(self.questions, user)

    def get_topic_questions_for_user(self, user):
        topic_retriever = QuestionRetriever.create_topics()
        return topic_retriever.retrieve_questions(self.questions, user)

    def get_news_questions_for_user(self, user):
        news_retriever = QuestionRetriever.create_news()
        return news_retriever.retrieve_questions(self.questions, user)

    def get_popular_questions_for_user(self, user):
        popular_retriever = QuestionRetriever.create_popular_today()
        return popular_retriever.retrieve_questions(self.questions, user)
```

Código Refactorizado:

```
class CuOOra:
    def __init__(self):
        self.questions = []

    def add_question(self, question):
        self.questions.append(question)

    def get_questions_for_user(self, user, retriever_factory: QuestionRetrieverFactory,
max_questions):
        return
retriever_factory.create(max_questions).retrieve_sorted_questions(self.questions, user)
```

Descripción de cambios:

- **Extracción de Clases:**
  Se dividió QuestionRetriever en clases especializadas, cada una con su propia implementación de retrieve_questions. Esto mejora la cohesión y separación de responsabilidades, evitando código condicional extenso. Además, se delegó la obtención de preguntas a otras clases cuando correspondía, reduciendo el feature envy, como en user.get_questions_from_following() y user.get_questions_from_topics_of_interest().
- **Factory Method:**
  Se eliminaron los métodos estáticos de QuestionRetriever y se introdujeron fábricas (SocialRetrieverFactory, TopicsRetrieverFactory, etc.) para gestionar la creación de instancias. Esto desacopla la lógica de negocio de la instanciación y facilita la extensión del sistema sin modificar código existente.
- **Simplificación en CuOOra:**
  En lugar de condicionales para determinar qué retriever usar, ahora CuOOra recibe una fábrica como parámetro y usa create() para obtener la instancia correspondiente. Esto hace que la clase sea más flexible y extensible, permitiendo agregar nuevos tipos de retriever sin modificar su código.
- **Template Method:**
  Como la ordenación y el límite de preguntas se repetían en todas las subclases de QuestionRetriever, se movió esta lógica a un método en la clase base: retrieve_sorted_questions(). De esta manera, las subclases solo se encargan de recuperar las preguntas sin preocuparse por la ordenación o el límite de resultados, mejorando la reutilización del código y reduciendo la duplicación.