1. Использование тестовых заглушек может быть полезным при:

Изоляция зависимостей: Когда вы тестируете конкретный модуль, его часто нужно изолировать от внешних зависимостей. Заглушки позволяют заменить реальные зависимости тестируемого модуля фиктивными объектами, которые ведут себя так же, как реальные, но без фактического взаимодействия с внешними ресурсами. Таким образом, можно сконцентрироваться только на тестируемой функциональности.

Создание контролируемого окружения: Заглушки позволяют создавать контролируемое окружение для выполнения тестов. Можно настроить их поведение и возвращаемые значения в соответствии с ожидаемыми сценариями и проверить, как модуль взаимодействует с этими зависимостями. Это дает уверенность в работоспособности модуля в разных ситуациях.

Ускорение выполнения тестов: При использовании реальных зависимостей в модульных тестах, они могут замедлить выполнение тестов из-за необходимости взаимодействия с внешними сервисами или базой данных. Заглушки позволяют избежать этого, поскольку предоставляют замену этих зависимостей, которые могут возвращать минимальные данные, не требующие реального взаимодействия.

В целом, использование тестовых заглушек позволяет упростить разработку модульных тестов, сделать их более предсказуемыми и независимыми от внешних факторов, что в конечном итоге приводит к повышению качества кода.

2. Если нужно проверить, что метод был вызван с определенными аргументами, можно использовать Verify-Style тестовые заглушки.

Для создания заглушек существую фреймворки Mockito, EasyMock или PowerMock. Эти инструменты обеспечивают Verify-Style заглушки, позволяющие проверить вызовы методов с определенными аргументами.

Вот пример использования Mockito для проверки вызова метода с определенными аргументами:

```
// Создаем мок объект

List<String> mockList = Mockito.mock(List.class);

// Вызываем метод на мок объекте с определенными аргументами

mockList.add(test);

// Проверяем, что метод add был вызван с аргументом test

Mockito.verify(mockList).add(test);
```

В этом примере используем Mockito для создания заглушки списка (mockList), а затем вызываем метод add с аргументом test. Затем используем метод verify Mockito, чтобы убедиться, что метод add был вызван с аргументом test.

Этот подход позволяет контролировать, какие аргументы передаются в методы заглушек и проверять, что вызовы происходят с правильными значениями. Это полезно, чтобы обеспечить корректность взаимодействия кода с его зависимостями.

3. Если нужно вернуть определенное значение или исключение в ответ на вызов метода, можно использовать Stub-Style тестовые заглушки (заглушки-точки).

Вот пример использования Mockito для создания заглушки-точки, которая возвращает определенное значение:

```
// Создаем заглушку-точку

List<String> mockList = Mockito.mock(List.class);

// Настраиваем заглушку чтобы она вернула значение при вызове метода get()

Mockito.when(mockList.get(0)).thenReturn(test);

// Проверяем возвращаемое значение

String result = mockList.get(0); // вернет test

System.out.println(result); // выводит test
```

В этом примере мы используем Mockito для создания заглушки списка (mockList) и настраиваем ее для возврата значения test при вызове метода get(0). Затем мы вызываем метод get(0) и сохраняем результат в переменную result. В результате на консоль будет выведено test.

Можно также настроить заглушку-точку для выбрасывания исключения при вызове метода. Пример использования Mockito для этой цели:

```
// Создаем заглушку-точку

List<String> mockList = Mockito.mock(List.class);

// Настраиваем заглушку чтобы она выбросила исключение при вызове метода clear()

Mockito.doThrow(new RuntimeException()).when(mockList).clear();
```

```
// Вызываем метод clear() - будет выброшено исключение
mockList.clear();
```

В этом примере настраиваем заглушку списка так, чтобы она выбрасывала исключение RuntimeException при вызове метода clear(). При вызове mockList.clear() будет сгенерировано исключение.

Stub-Style заглушки полезны, когда нужно контролировать возвращаемые значения или генерировать исключения при вызове методов, чтобы создать предсказуемое поведение во время тестирования.

4. Для имитации взаимодействия с внешним API или базой данных следует использовать Fake-Style тестовые заглушки.

Fake-Style заглушки представляют собой имитации реальных компонентов, которые имеют схожий интерфейс, но обычно выполняют упрощенную или эмулированную логику.

Можно использовать различные подходы и инструменты для создания Fake-Style заглушек, включая собственную реализацию класса с имитационной логикой, библиотеки для внедрения зависимостей, такие как Spring или Guice, или специальные фреймворки, такие как WireMock или Mockito.

Вот использование собственной реализации класса для создания Fake-Style заглушки:

```
public interface ExternalAPI {
    void sendData(String data);
    String fetchData();
}

public class FakeExternalAPI implements ExternalAPI {
    private String storedData;
    public void sendData(String data) {
        // Логика имитации отправки данных во внешнее API
        storedData = data;
    }
```

```
    public String fetchData() {
        // Логика имитации получения данных из внешнего API
        return Fake data;
    }
}
```

В этом примере у нас есть интерфейс ExternalAPI, который определяет методы для отправки данных (sendData) и получения данных (fetchData). Затем мы создаем имитационную реализацию класса FakeExternalAPI, где мы имитируем логику взаимодействия с внешним API. В данном случае, метод sendData сохраняет переданные данные, а метод fetchData возвращает фиктивные данные Fake data.

Таким образом, мы можем использовать FakeExternalAPI вместо реального ExternalAPI, когда выполняем модульное тестирование, чтобы имитировать взаимодействие с внешним API или базой данных.