حوری دهش 9821413

سوال 1:

الف)

اضافه کردن تست:

۱. برای اضافه کردن تست ها به یک سیستم قدیمی، ابتدا باید سیستم را تحلیل و تجزیه کنیم.
۲. در مرحله تحلیل اولیه، بخش هایی از سیستم را که بیشتر استفاده میشوند یا بیشتر تحت تغییرات قرار می گیرند را شناسایی می کنیم.
۳. سپس سیستم را به ماژول های کوچک‌تر تقسیم می کنیم تا فرایند تست ساده تر شود.
۴. در مرحله بعد، تست های رگرسیون و واحد را اجرا می کنیم. تست های رگرسیون برای اطمینان از عدم تخریب قابلیت های موجود نوشته می شوند.
۵. همچنین، برای هر ماژول تست های واحد نوشته می شود تا صحت عملکرد هر بخش به طور مستقل بررسی گردد.
۶. سپس تست ها را به صورت تدریجی پیاده‌سازی می کنیم. این کار باعث کاهش ریسک ها و افزایش پوشش تست می شود. کم کم و با گذشت زمان، تست های جدید به بخش های مختلف سیستم اضافه می شوند.

خطرات محتمل و روش های کاهش آنها:

خطرات:

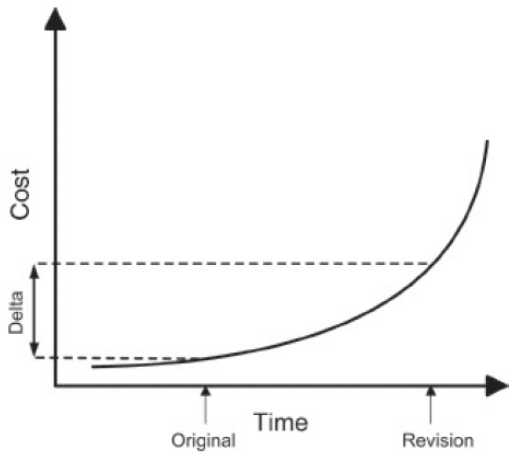
- عدم شناخت کامل از سیستم: ممکن است برای سیستم قدیمی مستندات کافی وجود نداشته باشد و شناخت کامل از آن، نداشته باشیم.
- مقاومت تیم در برابر تغییرات: تیم ممکن است در برابر تغییرات و اضافه کردن تست ها مقاومت کنند.
- افزایش زمان و هزینه: اضافه کردن تست ها به یک سیستم قدیمی می تواند زمان و هزینه بیشتری نسبت به توسعه معمولی داشته باشد.
- بروز مشکلات جدید: تغییرات در سیستم قدیمی ممکن است باعث بروز مشکلات جدیدی شود که پیش بینی نشده اند.

روش های کاهش خطرات:

- ایجاد مستندات حین تست نویسی: با نوشتن تست‌ها، مستندات لازم برای بخش های مختلف سیستم ایجاد کنیم.

- استفاده از ابزار های تست اتوماتیک: از ابزارهای تست اتوماتیک برای اجرای تست ها و اطمینان از عدم تخریب بخش های دیگر استفاده کنیم.
- استفاده از ابزارهای مدرن: از ابزارهای مدرن تست و اتوماسیون برای کاهش زمان و هزینه تست استفاده کنیم.
- آموزش و فرهنگ سازی: تیم را با مزایای اضافه کردن تست ها آشنا کنیم و آنها را به استفاده از روش های چابک تشویق کنیم.
- ایجاد محیط تست جداگانه: یک محیط تست جداگانه ایجاد کنیم تا تغییرات را در آن محیط آزمایش کرده و از ایجاد مشکل در محیط تولید جلوگیری کنیم.

ب)



توضیح نمودار:

این نمودار ریشه طرز تفکر متدلوژی های سنتی را مطرح میکند.

در متدلوژی های سنتی می گویند قبل از اینکه کدنویسی را شروع کنید، حسابی فکر کنید یعنی همه چیزها را شناسایی و تحلیل کنید و.. --> چون می خواهیم اگر یک راه حلی وجود داشت خیلی زود توسط این نمودار (زمان-هزینه) شناسایی شود.

این نمودار نشان میدهد: اگر در مبدا یک مشکلی بروز پیدا کند، فاصله زمانی بین ان لحظه ای که مشکل به وجود امده تا ان لحظه ای که احساس نیاز برای رفع این مشکل می کنیم، اگر این فاصله زمانی زیاد باشد، هزینه زیادی هم باید بدهیم --> پس یک حالت تصاعدی اینجا داریم بین زمان و ان هزینه به عبارت دیگر این نمودار نشان می دهد که چگونه هزینه ها با گذشت زمان افزایش می یابند، به ویژه زمانی که تغییرات و بازبینی ها در یک پروژه نرم افزاری انجام میشود.

قسمت Original: در ابتدای نمودار، هزینه‌ها نسبتا پایین هستند. این بخش نمایانگر زمانی است که سیستم اصلی در حال توسعه است.

قسمت Revision: با گذشت زمان و با نیاز به تغییرات و بازبینی ها، هزینه ها به طور مداوم افزایش می یابند پس این قسمت نمایانگر زمانی است که سیستم باید به‌روزرسانی و اصلاح شود.

هدف نمودار:

هدف این نمودار مدیریت تغییرات و اصلاح در پروژه های نرم افزاری است که با برنامه ریزی دقیق و استفاده از روش های موثر و کارآمد می توان از افزایش هزینه های بی رویه جلوگیری کرد. همچنین این نمودار به تیم توسعه یادآوری می کند که تغییرات و اصلاح به مرور زمان خیلی هزینه بر خواهد بود پس مدیریت صحیح و بهینه سازی فرآیندها از اهمیت بالایی برخوردار است.

دو دلیل افزایش مداوم هزینه ها:

۱. افزایش پیچیدگی در سیستم ها: با گذشت زمان، سیستم ها پیچیده‌تر می شوند و هر چه سیستم پیچیده تر شود، تغییرات و به روزرسانی آن به تلاش و منابع بیشتری نیاز خواهد داشت. این افزایش پیچیدگی ممکن است شامل اضافه شدن کدهای بیشتر، ماژول های متعددتر و وابستگی های پیچیده تر باشد که همگی نیازمند هماهنگی دقیق و تست های جامعی هستند.
۲. مشکلات ناشی از تصمیمات ناکارآمد در مراحل اولیه توسعه: در مراحل اولیه توسعه، ممکن است تصمیماتی گرفته شود یا به روش هایی پرداخته شود که به نظر می آید در کوتاه مدت کارایی داشته باشند اما به مرور زمان، این تصمیمات می توانند به عنوان بدهی فنی شناخته شوند که باعث افزایش هزینه و کاهش کارایی در آینده می شوند.

ج)

چالش ها و نقاط ضعف:

۱. تغییرات مکرر در نیازمندی ها: در روش چابک، تغییرات مکرر در نیازمندی ها می‌تواند برنامه ریزی و اجرای تست ها را به شدت تحت تاثیر قرار دهد. این تغییرات ممکن است تست ها را به طور مداوم تغییر دهد، که این امر می تواند منجر به تاخیر و کاهش کیفیت تست ها شود.
۲. پوشش کافی تست ها: در روش چابک، به دلیل فشار زمانی و تغییرات مکرر، احتمال نادیده گرفتن برخی بخش های سیستم در فرآیند تست بیشتر است. این وضعیت ممکن است به وجود باگ ها و نقص هایی منجر شود که در آینده مشکلات جدی تری ایجاد می کنند.
۳. فشار زمانی و اولویت بندی نیازمندی ها: در روش های چابک از آنجایی که می خواهند نرم افزار را سریع به مشتری تحویل دهند پس برایشان اولویت بندی نیازمندی ها اهمیت بیشتری پیدا می کند و زمان کمتری را به تست ها اختصاص می دهند که این وضعیت ممکن است به کاهش کیفیت و پوشش تست‌ها منجر شود.

۴. محدودیت منابع: ممکن است منابع محدودی برای توسعه و اجرای تست ها در دسترس باشد که امر باعث میشود که پوشش تست ها و کیفیت آنها کاهش پیدا کند.

مدیریت نقاط ضعف:

۱. استفاده از ابزارهای خودکار: استفاده از ابزارهای خودکار برای اجرای و مدیریت تست ها می‌تواند به بهبود کارایی و کاهش فشار زمانی کمک کند.
۲. پوشش تست ها: تلاش برای افزایش پوشش تست ها و اطمینان از آنکه تمامی نقاط کلیدی سیستم پوشش داده شده اند، می تواند باعث جلوگیری از ظهور باگ های ناخواسته شود.
۳. توسعه استراتژی تست: تیم ها باید استراتژی تست مناسبی را توسعه داده و آن را به صورت مداوم اصلاح کنند تا مطمئن شوند که تست ها بر اساس نیازمندی های فعلی و موارد استفاده واقعی سیستم انجام می شوند.
۴. انطباق با فرآیند Agile: تیم های توسعه و تست باید با روحیه Agile آشنا شوند و توانایی تطبیق با تغییرات مکرر و اولویت بندی نیازمندی ها را داشته باشند.
۵. هماهنگی بین توسعه و تست: تیم‌های توسعه و تست باید به طور پیوسته با یکدیگر همکاری و هماهنگی کنند تا اطمینان یابند که تست‌ها با نیازمندی‌ها سازگار هستند و هر گونه تغییر در نیازمندی‌ها مورد بررسی قرار گیرد.

د)

۱. تعریف user stories:

user stories به شکل کوتاه و ساده ای نیازمندی های کاربران را توصیف می کنند. هر user stories شامل 3 بخش اصلی است:

- Card: توضیح مختصری از نیازمندی یا ویژگی
- Conversation: بحث های بین تیم توسعه و سهامداران برای روشن کردن جزئیات
- Confirmation: معیارهای پذیرش که برای تایید، کامل شدن داستان استفاده می شوند.

۲. تبدیل user stories به Acceptance Tests:

هر user stories باید دارای معیار های پذیرش باشد که به صورت Acceptance Tests نوشته میشوند. این تست ها به عنوان معیارهای دقیقی عمل می کنند که نشان می دهند user stories به درستی پیاده سازی شده است.

۳. نوشتن Unit Tests:

بر اساس Acceptance Tests، توسعه دهندگان شروع به نوشتن unit test می کنند. در روش TDD، ابتدا unit test نوشته می شوند که برای ویژگی های جدید مورد نیاز هستند. این تست ها شکست می خورند چون هنوز کد مربوط به آنها نوشته نشده است.

۴. نوشتن کد برای پاس کردن تست ها:

بعد از نوشتن unit test و اطمینان از شکست آنها، توسعه دهندگان شروع به نوشتن کد می کنند تا این تست ها پاس شوند. این فرآیند به طور مکرر تکرار می شود تا همه unit testها پاس شوند و معیارهای پذیرش برآورده شوند.

۵. Refactoring:

بعد از اینکه تست ها پاس شدند، کد مورد بازبینی قرار می گیرد و بهبود داده می شود تا ساده تر و کارآمدتر شود. این کار بدون شکستن تست ها انجام می شود تا کیفیت کد به مرور زمان حفظ شود.

۶. پیگیری پیشرفت:

تیم چابک از ابزارهایی مانند تابلوهای Kanban یا Scrum برای پیگیری پیشرفت user stories و تست ها استفاده می کند. این ابزارها به تیم کمک می کنند تا وضعیت هر user stories را در هر لحظه ببینند و مدیریت بهتری بر فرآیند توسعه داشته باشند.

و)

A. Prefix values:

در حالت کلی آن مقدارهایی میشود که مورد استفاده قرار می گیرند نه بخاطر تست بلکه بخاطر این که برنامه را برسانند به آن استیتی که می خواهیم آن را تست بکنیم.

B. Controllability:

کنترل کردن یک نرم افزار یعنی این که چجوری بهش input بدهیم. هر چقدر ساده تر بتوانیم ورودی ها را در اختیار یک نرم افزار قرار بدهیم ینی کنترل کردنش راحت تر است و در نتیجه تست کردنش هم راحت تر می باشد پس این مربوط به ورودی ها و تاثیرات محیط روی نرم افزار است که تا چقدر خوب می توانیم، مشاهده بکنیم و کنترل بکنیم.

C. Observability:

یعنی این که چقدر خوب می توانیم خروجی های نرم افزار و تاثیراتی که روی محیط و یا سخت افزارها دارد را مشاهده کنیم --> یعنی رفتارها و output های نرم افزار و تاثیرات نرم افزار روی محیط و سخت افزارها و نرم افزارهای دیگر را تا چقدر خوب می توانیم بررسی کنیم.

D. Software testability:

یعنی این که تا چه حد می توانیم با تست، fault ها رو پیدا کنیم. به میزانی اشاره دارد که یک سیستم یا مولفه از آن، قابلیت ایجاد معیارهای تست و انجام تست ها را برای تعیین اینکه آیا این معیارها برآورده شده اند یا خیر، فراهم می کند.

E. Postfix values:

در حالت کلی آن مقدارهایی میشود که می خواهد برنامه را بعد از تست برساند به آن نقطه پایانی یا به جایی که کار می خواهد خاتمه پیدا کند به عبارت دیگر بعد از test case values این را داریم.

۵)

در روش data-driven، داده های ورودی مختلف را به برنامه می دهیم و خروجی مورد انتظار برای هر ورودی مشخص میکنیم و در انتها خروجی که از برنامه تولید میشود را با خروجی مورد انتظار مقایسه می کنیم.

مثالی که نوشته ام برای سه کاربر است.

یک تابع test_login نوشته ام برای تست کردن برنامه

نکته: برای فهم بهتر کامنت هایی درون این کد قرار گرفته است

```
def login():
    # اطلاعات 3 کاربر را در یک دیکشنری ذخیره می کنیم
    users = {
        'user1': 'pass1',
        'user2': 'pass2',
        'user3': 'pass3'
    }

    # ورود اطلاعات کاربر
    username = input("نام کاربری: ")
    password = input("رمز عبور: ")

    # بررسی صحت اطلاعات ورودی
    if username in users and users[username] == password:
        print("موفقیت آمیز است!")
    else:
        print("نام کاربری یا رمز عبور اشتباه است.")
```

```python
# تست صفحه ورود با داده‌های مختلف
def test_login():
    # تست صحت ورود با داده‌های مختلف
    test_cases = [
        ('user1', 'pass1', True),  # اطلاعات صحیح
        ('user2', 'pass123', False),  # رمز اشتباه
        ('user123', 'pass3', False),  # نام کاربری اشتباه
        ('user5', 'pass5', False)  # هیچ یک از اطلاعات صحیح نیست
    ]

    for username, password, expected_result in test_cases:
        print(f"ورود با نام کاربری '{username}' و رمز عبور '{password}':")
        if (username in users and users[username] == password) == expected_result:
            print("تست با موفقیت انجام شد.")
        else:
            print("خطا در تست.")

# اجرای صفحه ورود
login()

# اجرای تست‌ها
test_login()
```

ی)

قابلیت کنترل:

۱. یک سیستم که ورودی را از صفحه کلید خوانده و راه آهن ها را کنترل می کند. (زیاد)
۲. یک وب سایت با رابط کاربری تمیز و کاربر پسند. (متوسط)
۳. نرم افزاری برای یک میکروکنترلر که ورودی ها را از یک سنسور دما گرفته و یک کولر را کنترل می کند. (کم)

قابلیت مشاهده:

۱. یک وب سایت با رابط کاربری تمیز و کاربر پسند. (زیاد)
۲. یک سیستم که ورودی را از صفحه کلید خوانده و راه آهن ها را کنترل می کند. (متوسط)
۳. نرم افزاری برای یک میکروکنترلر که ورودی ها را از یک سنسور دما گرفته و یک کولر را کنترل می کند. (کم)

سوال 2:

در این سوال برای هر کد برای کاری که انجام داده ام، کامنت مشخصی برایش نوشته ام.

- Test adding items to the shopping cart with different quantities:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ShoppingCartTest {

    @Test
    public void testAddItemsToShoppingCart() {

        ShoppingCart shoppingCart = new ShoppingCart();

        // Adding Cake with quantity 2
        shoppingCart.addItem("Cake", 2);
        assertEquals(2, shoppingCart.getItemQuantity("Cake"));

        // Adding Lion with quantity 3
        shoppingCart.addItem("Lion", 3);
        assertEquals(3, shoppingCart.getItemQuantity("Lion"));

        // Adding Chocolate with quantity 5
        shoppingCart.addItem("Chocolate", 5);
        assertEquals(5, shoppingCart.getItemQuantity("Chocolate"));

        // Adding Bread with quantity 1
        shoppingCart.addItem("Bread", 1);
        assertEquals(1, shoppingCart.getItemQuantity("Bread"));


        // Adding another Cake
        // Adding more of an existing item
        shoppingCart.addItem("Cake", 1);
        // Now Cake should have a quantity of 3
        assertEquals(3, shoppingCart.getItemQuantity("Cake"));
    }
}
```

- Test removing items from the cart with different quantities:

نکته: از این مرحله به بعد، از کد قبلی استفاده می کنم یعنی فرض میکنم در قسمت قبلی که سبد خرید پر شده است الان در این قسمت و قسمت های بعدی از سبد خرید پر شده که در قسمت قبل داشتم، دارم استفاده میکنم:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ShoppingCartTest {

    //********There is testAddItemsToShoppingCart...********
    //******

    @Test
    public void testRemoveItemsFromShoppingCart() {

        ShoppingCart shoppingCart = new ShoppingCart();

        // Adding items to the shopping cart
        addItemsToShoppingCart(shoppingCart);

        // Removing 1 Cake from the cart
        shoppingCart.removeItem("Cake", 1);
        assertEquals(2, shoppingCart.getItemQuantity("Cake"));

        // Removing 2 Lions from the cart
        shoppingCart.removeItem("Lion", 2);
        assertEquals(1, shoppingCart.getItemQuantity("Lion"));

        // Removing 3 Chocolates from the cart
        shoppingCart.removeItem("Chocolate", 3);
        assertEquals(2, shoppingCart.getItemQuantity("Chocolate"));

        // Removing 1 Bread from the cart
        shoppingCart.removeItem("Bread", 1);
        assertEquals(0, shoppingCart.getItemQuantity("Bread"));

        // Removing another Cake from the shopping cart
        shoppingCart.removeItem("Cake", 1);
        assertEquals(1, shoppingCart.getItemQuantity("Cake"));
    }
}
```

- Test removing items that don't exist in the cart:

نکته: اسم Exception یه چیزی از خودم است:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ShoppingCartTest {

    //********There is testAddItemsToShoppingCart...********
    //******
    //********There is testRemoveItemsFromShoppingCart...********
    //******

    @Test(expected = TheItemDoesNotExistException.class)
    public void testRemoveItemsThatDoNotExistInShoppingCart() {
        ShoppingCart shoppingCart = new ShoppingCart();

        // Adding items to the shopping cart
        addItemsToShoppingCart(shoppingCart);

        // Removing an item (Cheese) that doesn't exist in the cart
        shoppingCart.removeItem("Cheese", 1);

        // Removing another item (Chips) that doesn't exist in the cart
        shoppingCart.removeItem("Chips", 3);

    }
}
```

- Test adding negative quantities:

نکته: اسم Exception یه چیزی از خودم است:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ShoppingCartTest {

    //********There is testAddItemsToShoppingCart...********
    //******
    //********There is testRemoveItemsFromShoppingCart...********
    //******
    //********There is testRemoveItemsThatDoNotExistInShoppingCart...********
    //******

    @Test(expected = TheNumberIsNegativeException.class)
    public void testAddNegativeQuantitiesToShoppingCart() {

        ShoppingCart shoppingCart = new ShoppingCart();
        addItemsToShoppingCart(shoppingCart);

        shoppingCart.addItem("Cake", -2);

        shoppingCart.addItem("Coffee", -3);
    }
}
```

- Test removing more items than is available:

نکته: اسم Exception یه چیزی از خودم است:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ShoppingCartTest {

    //********There is testAddItemsToShoppingCart...********
    //******
    //********There is testRemoveItemsFromShoppingCart...********
    //******
    //********There is testRemoveItemsThatDoNotExistInShoppingCart...********
    //******
    //********There is testAddNegativeQuantitiesToShoppingCart...********
    //******

    @Test(expected = ThisNumberIsNotAvailableException.class)
    public void testRemoveMoreItemsThanAvailableFromShoppingCart() {

        ShoppingCart shoppingCart = new ShoppingCart();
        addItemsToShoppingCart(shoppingCart);

        shoppingCart.removeItem("Chocolate", 8);

        shoppingCart.removeItem("Lion", 12);

        shoppingCart.removeItem("Bread", 6);

    }
}
```

سوال 3:

```python
import unittest
from library_system import LibrarySystem

class TestLibrarySystem(unittest.TestCase):

    def setUp(self):
        self.library = LibrarySystem()

    def test_add_book(self):
        self.library.add_book(1, "The Midnight Library", "Matt Haig", 6)
        self.assertEqual(self.library.books[1]['title'], "The Midnight Library")
        self.assertEqual(self.library.books[1]['author'], "Matt Haig")
        self.assertEqual(self.library.books[1]['quantity'], 6)

    def test_add_book_already_exists(self):
        self.library.add_book(1, "The Midnight Library", "Matt Haig", 6)
        with self.assertRaises(ValueError):
            self.library.add_book(1, "Harry Potter", "J. K. Rowling", 2)

    def test_borrow_book_success(self):
        self.library.add_book(1, "The Midnight Library", "Matt Haig", 6)
        result = self.library.borrow_book(1)
        self.assertTrue(result)
        self.assertEqual(self.library.books[1]['quantity'], 5)
```

```
    def test_borrow_book_not_available(self):
        self.library.add_book(1, "The Midnight Library", "Matt Haig", 0)
        result = self.library.borrow_book(1)
        self.assertFalse(result)
        self.assertEqual(self.library.books[1]['quantity'], 0)

    def test_borrow_book_not_exist(self):
        result = self.library.borrow_book(1)
        self.assertFalse(result)

    def test_return_book_success(self):
        self.library.add_book(1, "The Midnight Library", "Matt Haig", 6)
        self.library.borrow_book(1)
        result = self.library.return_book(1)
        self.assertTrue(result)
        self.assertEqual(self.library.books[1]['quantity'], 6)

    def test_return_book_not_exist(self):
        result = self.library.return_book(1)
        self.assertFalse(result)
```

```
    def test_get_book_quantity(self):
        self.library.add_book(1, "The Midnight Library", "Matt Haig", 6)
        quantity = self.library.get_book_quantity(1)
        self.assertEqual(quantity, 6)

    def test_get_book_quantity_not_exist(self):
        quantity = self.library.get_book_quantity(1)
        self.assertIsNone(quantity)

if __name__ == '__main__':
    unittest.main()
```

سوال 4:

تست کیس :

prefix values : آن مقدارهایی است که استفاده میشود برای اینکه به نقطه میانگین گیری برسیم که می خواهیم آن را تست بکنیم. (در عکس مشخص شده است --> رنگ های قرمز) --> و مقداری که اینجا داریم command =1 میشود برای اینکه مقادیر را برای میانگین گیری بگیرد تا بتواند به آن نقطه ای برسد که میانگین مورد نظر را تست کند پس ابتدا باید مقادیر را گرفت که میشود prefix.

test values : 64 , 17 , 46 , 3 , 6 , 2 , 11 , 9 , 5 , 1

expected values : 16.4

postfix values : آن مقدارهایی است که بعد از تست میانگین گیری داریم تا برنامه روال عادی خودش را طی کند تا به مرحله خاتمه برسد. (در عکس مشخص شده است --> رنگ های زرد) به عبارت دیگر مقادیر بعد از test case values میشود که در اینجا میشود:

- مقدار 1- که در تابع addScores است برای اینکه وقتی که مقادیر را گرفت این مقادیر میشود test case values ما و بعد از آن اگر مقدار 1- را بزند دیگر ورودی نمی گیرد و از برنامه خارج میشود پس این مقدار میشود postfix ما
- command =2 چون این مقدار بعد از test case values است و باعث میشود خروجی برنامه را ببینیم
- چون صورت سوال این فرض را کرده است که خروجی برنامه قابل اجرا است پس مقداری که می توانیم برای این داشته باشیم command =0 است


prefix value
void addScores(std::vector<float>& scores) {
std::cout << "enter scores (enter -1 to finish):" << std::endl;
// code for inputting scores until -1 is entered
postfix
}
void findPrintAVG(const std::vector<float>& scores) {
// code for finding and printing the average of scores
postfix value
}
int main() {
int command = -1;
std::vector<float> scores;
while (command) {
std::cin>> command;
switch (command) {
case 1:
addScores (scores);
break;
prefix value
case 2:
findPrintAVG(scores);
break;
case 0:
std::cout << "goodbye!" << std::endl; break;
break;
postfix value
default:
std::cout << "wrong command!" <<< std::endl;
}
}
}