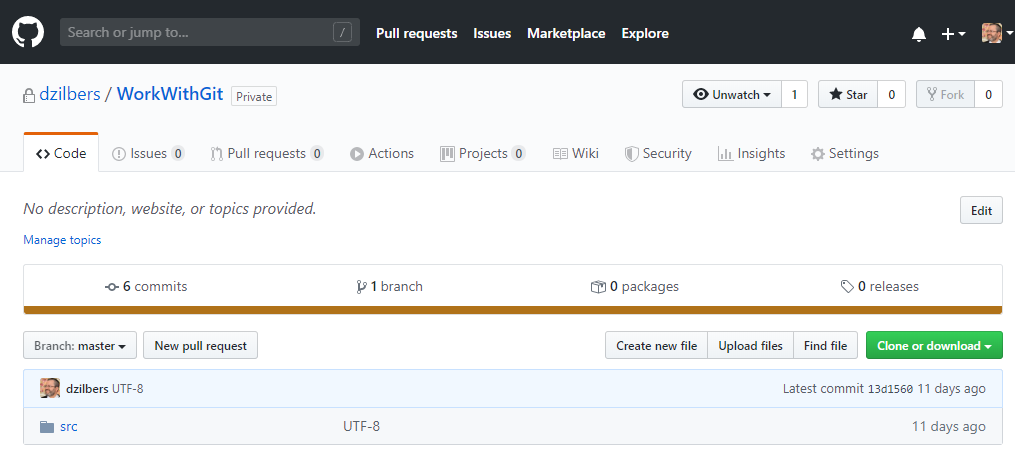
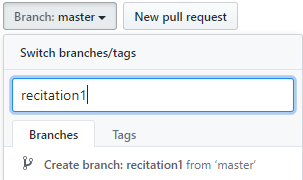
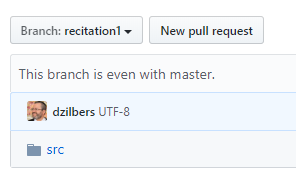
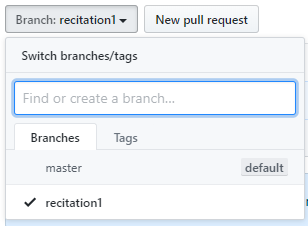
**שלב 2 – בדיקות אוטומטיות ומימוש חישובי נורמלים**

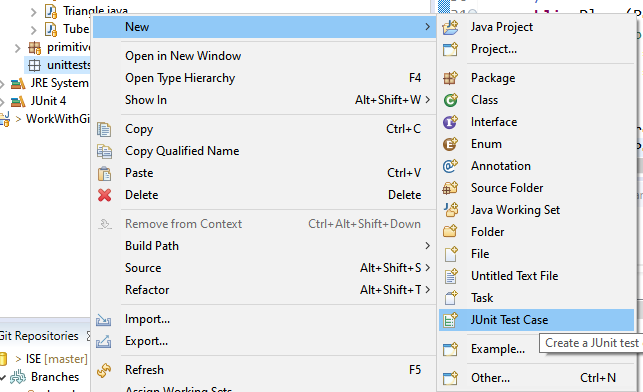
לפני הכל נקפיא את השלב האחרון באתר **github** ע"י יצירת branch חדש בשם **recitation1**. ונוודא שענף ברירת המחדל נשאר **master**. כשניכנס לאתר לפרויקט שלנו, נראה עמוד כדלקמן – נלחץ על פתיחת התפריט של **Branch: master**:

תיפתח חלונית כזו:  
  
שבה נמלא את שף הענף החדש recitation1 (באותיות קטנות) ונלחץ Enter על המקלדת. התצוגה תשתנה ככה:  
  
נלחץ עוד פעם על הכפתור של Branch, נוודא שה-master נשאר ענף בררת מחדל ונחזור אליו ע"י לחציה על עכבר:  


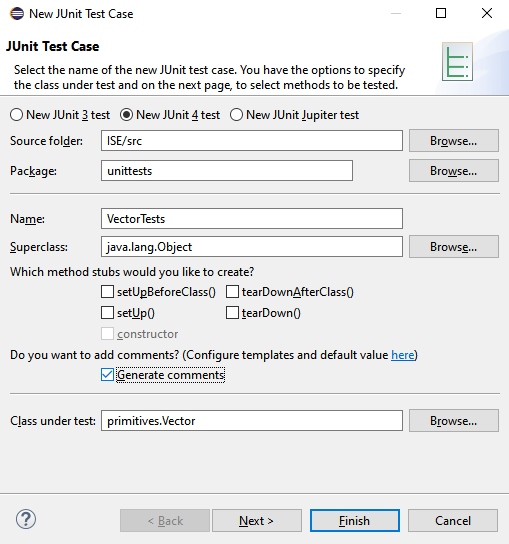
על בסיס חבילות הפרימיטיבים והגופים הגאומטריים שהגדרנו בשלב 1 נוכל עכשיו להרחיב את אוסף המחלקות ולהוסיף כמה פעולות. והעיקר – נתחיל לבנות מערכת בדיקות אוטומטיים – **Unit Tests** על מנת לבנות בסיס ל-**TDD** – פיתוח מבוסס בדיקות.

1. נוסיף לפרויקט את ספריית הבדיקות האוטומטיות **JUnit** ע"פ ההנחיות שקיבלנו בכיתה
2. נוסיף חבילת בדיקות **unittests**, ובה מחלקת **VectorTests** ומחלקות בדיקה עבור כל גוף גאומטרי בהתאם (**PlaneTests**, וכו'), הערות כלליות לכתיבת הבדיקות:

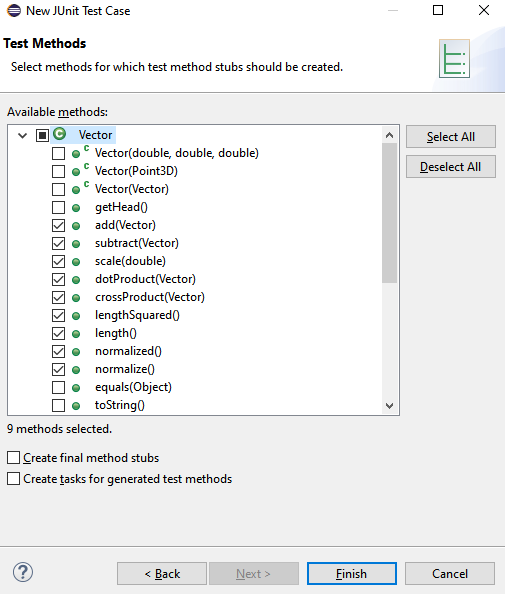
* על מנת ליצור מודול (מחלקה) לבדיקת מחלקה מסוימת מהפרויקט נעמוד עם העכבר מעל שם החבילה **unittests**, נפתח תפריט הקשר (לחיצה על כפתור ימין של העכבר) ונוסיף משם בדיקה חדשה:



* בחלון שנפתח נוודא שהבדיקה מסוג **Junit 4**, נרשום את שם המחלקה של בדיקות, נסמן יצירת הערות ונרשום את השם המלא או נבחר בעזרת הכפתור **Browse…** את המחלקה הנבדקת ונלחץ על הכפתור **Next>**:



* בחלון הבא נסמן את הפעולות שאנחנו רוצים לבדוק ונלחץ על הכפתור **Finish**:



* נעשה את הפעולות כנ"ל עבור כל מחלקת בדיקות
* בשלב הבא קודם כל נשלים את תעוד ה-**Javadoc** של המחלקה והפונקציות כנדרש:

**/\*\***

**\* Unit tests for primitives.Vector class**

**\* @author Dan**

**\*/**

**public class VectorTests {**

**/\*\***

**\* Test method for {@link primitives.Vector#crossProduct(primitives.Vector)}.**

**\*/**

**@Test**

**public void testCrossProduct() {**

***fail*("Not yet implemented");**

**}**

* כשאנחנו כותבים בדיקות בפונקציות, נקפיד להוסיף הערות כדלקמן:
  + בתחילת כל הבדיקות של מחלקות שקילות ולפני כל הבדיקות של מקרי גבול נוסיף הערות כותרת:

**// ============ Equivalence Partitions Tests ==============**

**...**

**// =============== Boundary Values Tests ==================**

**...**

* + לפני כל **test case** נוסיף הערה המתארת את המקרה המסוים הנבדק במשפט אחד
* נמחק את השורה הבאה: ***fail*("Not yet implemented");**
* ונוסיף את כל הבדיקות שתכננו, למשל דוגמא איך יכולה להיראות פעולת בדיקה:

**/\*\***

**\* Test method for {@link primitives.Vector#crossProduct(primitives.Vector)}.**

**\*/**

**@Test**

**public void testCrossProduct() {**

**Vector v1 = new Vector(1, 2, 3);**

**Vector v2 = new Vector(-2, -4, -6);**

**// ============ Equivalence Partitions Tests ==============**

**Vector v3 = new Vector(0, 3, -2);**

**Vector vr = v1.crossProduct(v3);**

**// Test that length of cross-product is proper (orthogonal vectors taken for simplicity)**

***assertEquals*("crossProduct() wrong result length", v1.length() \* v3.length(), vr.length(), 0.00001);**

**// Test cross-product result orthogonality to its operands**

***assertTrue*("crossProduct() result is not orthogonal to 1st operand", *isZero*(vr.dotProduct(v1)));**

***assertTrue*("crossProduct() result is not orthogonal to 2nd operand", *isZero*(vr.dotProduct(v3)));**

**// =============== Boundary Values Tests ==================**

**// test zero vector from cross-productof co-lined vectors**

**try {**

**v1.crossProduct(v2);**

***fail*("crossProduct() for parallel vectors does not throw an exception");**

**} catch (Exception e) {}**

**}**

1. נעביר את הבדיקות של פעולות ווקטוריות מהתוכנית הראשית של שלב 1 לפונקציות בדיקה המתאימות (פונקציית בדיקה לכל פעולה וקטורית שנבדקת – בתוך כל פונקציית בדיקה יהיו כל הבדיקות של אותה הפעולה הוקטורית) של המחלקה **VectorTests** תוך התאמות לסביבת בדיקות אוטומטיות של **Junit**
2. נוסיף למחלקות בדיקות הגופים את בדיקות פעולת **getNormal(Point3D point)** במודולים נפרדים עבור כל גוף, בכתיבת הבדיקות האלה יש לדאוג מראש שתמיד הנקודה בארגומנט הפעולה תהיה על פני שטח הגוף הנבדק, **חובה לבנות את הבדיקות קודם כתיבת המימושים של הפונקציה**! כמו כן:

* בתיבת התרגיל ניתן לסטודנטים מודול **PolygoneTests.java** מוכן שכולל את פוקנציית בדיקות הבנאי ואת פונקציית בדיקות הנורמל של מצולעים, יש להעתיק את המודול לחבילה\תיקיה **unittests** של הפרויקט שלכם ולעיין בו על מנת ללמוד איך לבנות מודול ופונקציות של בדיקות
* בבדיקות נורמל של מישור, מצולע ומשולש [חובה לבדוק בכולם – בלי קשר לדרך המימוש כפי שתפורט בהמשך – אנחנו עובדים ע"פ **TDD**!) יש רק מחלקה שקילה (**EP**) אחת ואין מקרי גבול (**BVA**)
* בבדיקות נורמל של ספירה וגליל אין סופי גם יש רק מחלקה שקילה אחת ואין מקרי גבול
* עבור מי שעושה חישוב נורמל לגליל סופי – בבדיקות נורמל של גליל סופי יש שלוש מחלקות שקילות (בצד ובכל אחד משני הבסיסים) ושני מקרי גבול – בגבול בין כל אחד מהבסיסים לצד הגליל (עליכם להכריע לאן תצביע הנורמל במקרה הזה)

1. נממש את הפעולה **getNormal(Point3D point)** עבור כל הגופים – ע"פ הנלמד בקורס התאורטי חזרנו בשקפים של המעבדה (עבור ספירה, מישור וגליל אין סופי (צינור))

* יש להניח שהנקודה שמתקבלת בפרמטר נמצאת על פני שטח הגוף ואין צורך לבדוק את תקינות הנקודה
* יש לממש את הפעולה רק במחלקות המתאימות (לפי הארכיטקטורה בסרטוט של שלב 1)
* ב**מישור** הפעולה תחזיר את ערך השדה של נורמל, **אך בשלב הזה נשלים את הבנאי עם 3 נקודות** כך שיחשב את הנורמל לפי מה שלמדנו – במודל המתמטי של נורמל למשולש
* ב**מצולע** הפעולה כבר ממומשת מהשלב הקודם ע"י האצלה לפעולה מאחזרת של נורמל המישור המוכל
* ב**משולש** מימוש הפעולה לא נדרשת – המימוש מתקבל בירושה מהמצולע
* בספרה ובגליל הפעולה תמומש לפי המודל המתמטי שלמדתם בקורס התאורטי (ראו גם בשקפים של המעבדה)
* אין חובה לממש את הפעולה עבור גליל סופי – מי שיממש בצורה נכונה – יקבל בונוס של 1 נק' לציון הכללי