תכנות מתקדם 2: 89-211 – מועד ב' תשע"ו

זמן המבחן: **שעתיים**, יש לענות על 6 מתוך 6 שאלות, **בגוף השאלון בלבד**. חומר **סגור**.

בקיאות

שאלה1: (20 נק') לאפליקציית ה web שלנו יש בעיית סקלביליות.

- אהרון טוען שיש לבצע scale out, ואז load balancing באמצעות אלג' round robin
  - כלומר הטלת המשימות (טיפול בבקשות הלקוחות) על השרתים תתבצע ע"פ סבב קבוע.
- ברכה טוענת שיש לבצע scale up, ומכיוון שמדובר ב sticky session עם הלקוח אז ממילא חלוקת המשימות צריכה להתבצע באמצעות session affinity.
  - כלומר אותו השרת שטיפל בלקוח מסוים ימשיך לטפל גם בבקשות הבאות שלו, ולכן אין צורך ב scale out.
- גדעון לא מסכים עם ברכה. הוא לא מאמין ב session affinity והוא טוען שיש צורך ב scale out.
  - כל שרת יוכל לטפל בכל משימה שתהיה, כי את ה data של ה session נשמור בשרת אחר המהווה central session store, וכך נוכל לבצע load balancing טוב יותר!
- דקלה מציעה גם scale out וגם session affinity באופן הבא:
  - כבר בבקשה הראשונה נשמור ב cookie אצל הלקוח את ה ID של ה session. בכל בקשה נוספת מהלקוח הוא יעביר את המידע שב cookie וכך נוכל להקצות את המשימה אל השרת שטיפל בו בעבר.
- הרשלה טוען שיש לבצע scale out, ו IP Address Affinity.
  - בהינתן בקשת לקוח, נריץ hash על ה IP שלו, והתוצאה מודולו N תקבע מי מתוך N השרתים צריך לטפל בבקשה שלו.

עבור על אחד תארו בקצרה מהם היתרונות והחסרונות בשיטה שהציע (4 נק' לכל שיטה, בטופס המבחן היה כתוב בטעות 5 נק' לכל שיטה). מספיק יתרון אחד וחסרון אחד מהותי לשיטה שהוצעה על מנת לקבל ניקוד מלא.

אהרון יתרונות:

- Scale out עוזר למקבל משימות ולכן עשוי לעזור לבעיות שלנו
- חלוקת המשימות נעשית בצורה מאד מאוזנת
- אין באמת בעיה אם אחד השרתים נופל

חסרונות:

- השרת שמטפל בלקוח כלשהו כעת לא בהכרח טיפל בו בעבר, ולכן תהיה בעיה היכן לשמור את האינפורמציה של ה session איתו

ברכה יתרונות:

- אין בעיה לזכור את האינפורמציה של ה session עם לקוח כלשהו

חסרונות:

- אין load balancing כי יתכן ששרת אחד יטפל בהמון לקוחות בעוד אחר מטפל במעט בלבד

- Scale up משפר את הביצועים של כל שרת בודד, אך עד גבול מסוים. בסוף בעיית הסקלביליות תגבור על הפתרון הזה.

גדעון יתרונות:

- Scale out עוזר למקבל משימות ולכן עשוי לעזור לבעיות שלנו
- קיים load balancing בשיטה זו עבור ה app servers

חסרונות:

- ה central session store הופך לצוואר בקבוק שיכול לעכב את התגובתיות של כל האפליקציה שלנו

דקלה יתרונות:

- Scale out עוזר למקבל משימות ולכן עשוי לעזור לבעיות שלנו
- שמירת האינפורמציה אצל הלקוח של מי טיפל בו פותרת את בעיית ה central session store
  - כל לקוח מטופל היכן שטיפלו בו בעבר, ניתן לזכור את אינפורמציית ה session בקלות

חסרונות:

- הלקוח לא מחויב לשמור cookies אצלו... קיים חוסר אמון ב cookies
- תקורה של עוד אינפורמציה שהלקוח צריך להעביר על ערוצי התקשורת

הרשלה יתרונות:

- קיים load balancing טוב בתלות כמובן בפונקציית ה hash
- אין צורך ב cookies
- השרת שטיפל בלקוח בעבר יטפל בו גם כעת
  - אמנם ה IP של הלקוח היא דינמית אך לרוב, sessions לא נמשכים כל כך הרבה זמן.

חסרונות:

- זה לא consistent hashing, אם אחד השרתים נופל הלקוחות יפנו לשרתים שלא טיפלו בהם בעבר...

יתכנו כמובן עוד ניתוחים מהותיים שכתבתם וקבלנו.

שאלה 2: (12 נק')

הקיפו בעיגול את התשובות הנכונות:

א. ניתן באמצעות double check locking להתגבר על בעיית ה singleton בסביבה שהיא multithreaded. - נכון
ב. MVVM ניתן ממש רק ב NET. בטכנולוגיית WPF. – לא נכון
ג. Service Locator מהווה Runtime linker - נכון
ד. ב MVP שכבות המודל וה View צריכות להכיר רק את עצמן. – נכון

מיומנות עיצוב קוד (Design) וכתיבת קוד

שאלה 3: (20 נק')

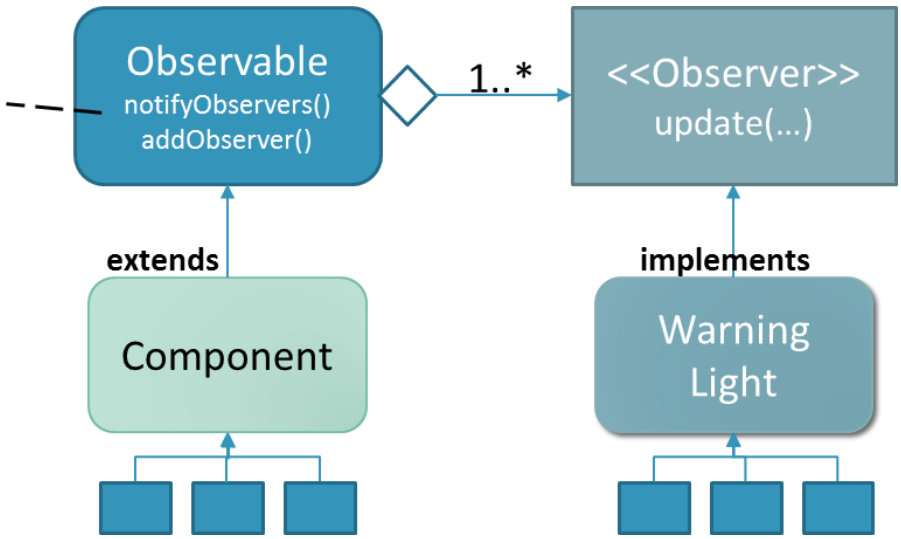
לצורך איתור ואבחון תקלות ברכבים אנו ממדלים את ההתנהגות הצפויה של כל רכיב ורכיב – כל רכיב (Component) זוכה לאובייקט משלו. המחלקה WarningLight תומכת בפעולות ()on, ()off ו ()blink. יש לה מימושים שונים כמו LowFuel, CheckEngine וכו'. כאשר רכיב כלשהו הופך לתקול אז נורית האזהרה צריכה לפעול בהתאם. לדוגמא, כאשר הרכיב TimingChain הופך לתקול אז ה CheckEngine צריכה להיות דולקת, וה OverdriveOff צריכה להבהב. באופן דומה גם אם המנוע תקול אז CheckEngine צריכה לדלוק. לכל נורת אזהרה קיימים תנאים שונים שתחתם עליה לפעול בצורה הרצויה.

המתכנתים של המחלקות מסוג WarningLight יודעים להגדיר את התנאים לפעולה הרצויה (להידלק, להבהב, או להיכבות) אך הם אינם יודעים *מתי* ליזום את הפעולה הרצויה (כלומר מתי הרכיב הופך לתקול)

עליכם **לשרטט** תרשים מחלקות (class diagram) **ב UML**, המציג עיצוב שמאפשר ליזום את הפעולה הרצויה של כל נורית אזהרה כאשר הרכיבים הרלוונטיים הופכים לתקולים (ללא busy waiting).

תשובה:

```
for(Observer o : observersList){
        o.update(...);
}
```



שימוש בתבנית עיצוב מסוג Observer, Publisher/subscriber, event & delegate – 10 נק'
הרכיב ניתן לצפייה, נורית האזהרה צופה – 8 נק'
שרטוט נכון – 2 נק'

שאלה 4: ממשו בקוד (באיזו שפה מונחית עצמים שתרצו) את הפתרון העיצובי של שאלה 3. (16 נק')

```
// in Java:
public class Component extends Observable { // 4 points
        // when at fault just call notifyObservers()
}
public abstract class WarningLight implements Observer{ // 4 points
        // the update() method should be overridden in the subclasses
}
public class ExampleLight extends WarningLight{ // 2 points
        @Override
        Public void update(Observable o, Objects arg){ // 4 points
                // store each observable that called update
                // apply the policy to call on() blink() or off()
        }
        public ExampleLight(Component[]  components){ // 2 points
                for(Component c : components) c.addObserver(this);
        }
}
```

מימוש אפשרי ב #C :

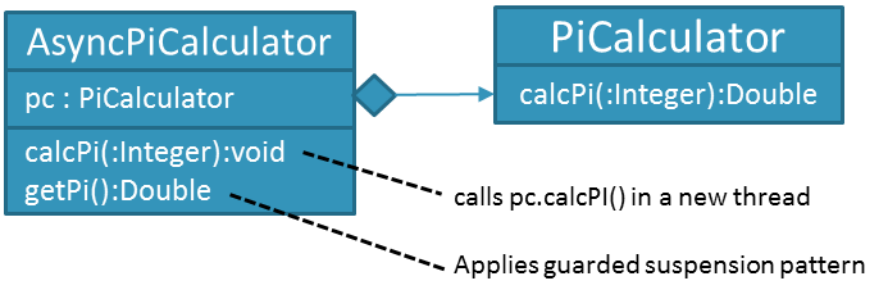
```
public class Component{ // 8 points
        public void delegate update(object sender, EventArgs args);
        public event update notify;
        // when at fault apply notify() if it is not Null
}
public class ExampleLight : WarningLight{ // 2 points
        public updateMe(object sender, EventArgs args) { // 4 points
                // store senders & apply your policy here
        }
        public ExampleLight(Component[] components){ // 2 points
                foreach(Component c in components)    c.notify+=updateMe;
        }
}
```

שאלה 5: (16 נקודות)

נותנה המחלקה PiCalculator שממשה את המתודה double calcPi(int digits) המחשבת את פאי עד ל digits ספרות לאחר הנקודה. לצערנו זה עלול לקחת לא מעט זמן וזה קוד סגור שלא ניתן לשינוים. עליכם **לשרטט** תרשים מחלקות (class diagram) **ב UML**, המציג עיצוב שמאפשר

- להריץ את calcPi בצורה אסינכרונית
- ולטפל בערך המוחזר כך ש
  - הלקוח יוכל לשלוף אותו בעת הצורך
  - ורק אם הלקוח מנסה לשלוף את הערך לפני הזמן הוא יאלץ להמתין עד לסוף החישוב.

תשובה:



נשתמש בסוג של Object Adapter שעושה שימוש ב Guarded Suspension

ההכלה של PiCalculator עדיפה על הירושה שלה משתי סיבות עיקריות:

1. אולי בעתיד יהיו לנו עוד כמה סוגים של PiCalculator ואז מספיק לנו object adapter אחד (במקום class adapter לכל סוג של PiCalculator)
2. לו הינו משתמשים ביורשה אז המתודה הסינכרונית calcPi() שמחזירה double עדיין היתה אפשרית לשימוש ב AsyncPiCalculator

ניתן לטעון "אז למה לא? בוא נתמוך גם באפשרות הזו" – לא רצוי אם מהות המחלקה היא להוות גרסה אסינכרונית של PiCalculator

ב C++ למשל, ניתן לבצע ירושת private של PiCalculator ואז להסתיר את המתודה הנ"ל, אך כאמור עדיין יהיה החסרון של class adapter.

נשים לב שבמחלקה AsyncPiCalculator הפרדנו בין פקודת החישוב לבין מתודת ה get עבור לקיחת התוצאה. זו תבצע guarded suspension במידה ויקראו לה לפני הזמן. המתודה calcPi תקרא ל someObject.notify() לאחר סיום החישוב, ואילו getPi() תבצע someObject.wait() לפני ההחזרה של פאי, במידה ופאי עדיין לא חושב (נניח שווה ל 1-).

אפשרות נוספת היא כמובן להשתמש בכלים שנמצאים בשפה בה אתם כותבים. למשל ב Java נוכל לכתוב ל AsyncPiCalculator את המתודה Future<Double> calcPi(int digits); כך שזו תחזיר מיד אובייקט מסוג Future ולכשיסתיים הת'רד היא תזין לו את הערך המוחזר של פאי. Future כבר מיישמת את ה Guarded suspension כך שאם מבקשים ממנה get לפני הזנת הערך היא תמתין.

8 נקודות על ההכלה

8 נקודות הפרדת הקריאה לחישוב מקבלת ערך ההחזרה (ע"י 2 מתודות או שימוש future)

שאלה 6: ממשו בקוד (באיזו שפה מונחית עצמים שתרצו) את העיצוב של שאלה 5 (16 נק')

בהצלחה!

תשובה 6:

```
// in Java using Future
public class AsyncPiCalculator{
	PiCalculator pc;
	public  AsyncPiCalculator(PiCalculator pc) { this.pc=pc;}
	Future<Double> calcPi(int digits){
		// ver 7 requires ret to be final…
		Future<Double> ret=new Future<Double>();
		new Thread(new Runnable(){
			public void run(){
				ret.set(pc.calcPi(digits));
			}
		).start();
		return ret;
	}
}
```

התאמה לתרשים 8 נק'

נכונות הקוד 8 נק'