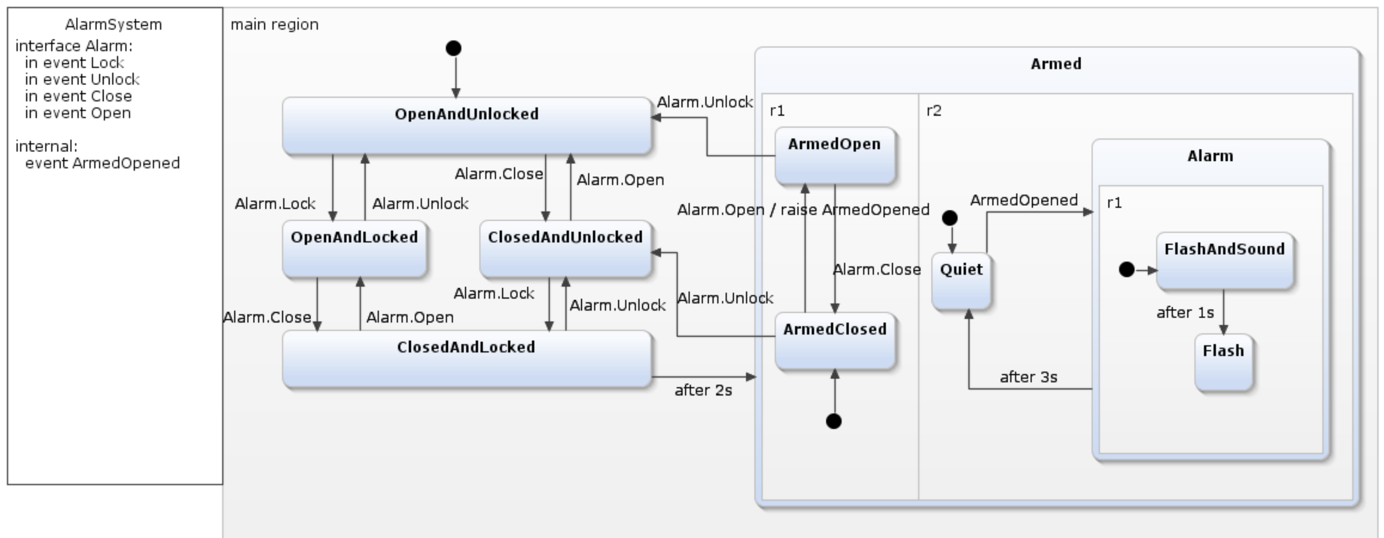
# 2. gyakorlat: Részletes tervek és forráskód ellenőrzése

A gyakorlaton a részletes tervek ellenőrzésével és a forráskód verifikációját végző statikus ellenőrző eszközökkel fogunk foglalkozni.

## Részletes tervek ellenőrzése

A feladat kidolgozása során egy előre elkészített Yakindu állapottérkép modell ellenőrzését fogjuk elvégezni. Az ellenőrzés alapja, hogy a Yakindu modellt az UPPAAL modellellenőrző bemeneti formátumára transzformáljuk (ez a modellellenőrző a *Formális módszerek* tárgyból már ismerős), majd az UPPAAL eszközben temporális logika segítségével formalizálunk és vizsgálunk egyszerű követelményeket.

- Az állapottérkép modell megtekintéséhez indítsuk el az asztalon található parancsikonnal az Eclipse eszközt. Itt megtalálható az előre beimportált *YakinduModel* projekt, amely az állapottérkép modellt tartalmazza (a *json.simple* projekt a gyakorlat második feléhez lesz majd szükséges).
- A projekt egyetlen *AlarmSystem.sct* fájlból áll. Ez egy gépkocsi riasztóberendezésének modelljét tartalmazza. A fájl megnyitása után középen látható a diagram, alul pedig a kiválasztott elem tulajdonságai. Tekintsük át a modellt[1]:




1. ábra: Állapottérkép

- A bal oldalon található deklarációs mezőben vannak felsorolva a lehetséges események. Ezek csoportosítására az interfészek adnak lehetőséget. Jelen modellben egy *Alarm* nevű interfész van, ami fogadja a külső eseményeket (*Lock*: riasztó indítása*, Unlock*: riasztó leállítása*, Close*: ajtó zárása*, Open*: ajtó nyitása). Emellett a modell rendelkezik egy belső *ArmedOpened* eseménnyel is. (Ezt az eseményt az *ArmedClosed* → *ArmedOpen* állapotátmenet válthatja ki.) Az *after …s* feliratú átmenetek nem eseményhez, hanem időzítéshez kötöttek.

A modell átnézése után transzformáljuk a modellt az UPPAAL által elfogadott formális modellé, azaz időzített automatává. Ehhez az *AlarmSystem.sct* fájlon a jobb egérgombot lenyomva válasszuk ki a

[1] A Yakindu eszköz használatáról és lehetőségeiről további információt a http://statecharts.org/tutorial.html és http://statecharts.org/documentation.html honlapokon lehet találni.

*YakinduToUppaal* menüpontot. Így a *C:\workspace\YakinduModel* könyvtárba egy *AlarmSystem.xml* fájl generálódik. (Egy *.uppaal* kiterjesztésű fájl is keletkezik, de ez most nem szükséges.)

Az ellenőrzéshez nyissuk meg a keletkezett *AlarmSystem.xml* modellt az UPPAAL eszközben (az asztalon megtalálható az UPPAAL parancsikonja), és gondoljuk végig a válaszokat a következő kérdésekre, illetve végezzük el a feladatokat:

1. Tekintsük át a generált időzített automatát. Látható, hogy az állapottérkép leképezése több *Template*-ből áll. Az állapottérkép modell mely elemeinek feleltethetők meg a *Template*-ek?
2. A Yakindu állapottérkép nem rögzíti a környezet viselkedését (hogyan érkezhetnek a külső események). A modellellenőrzéshez viszont ennek megadására is szükség van. A modelltranszformáció egy alapértelmezett környezeti modellt illeszt a rendszer modellje mellé; ez látható az UPPAAL *controlTemplate* automatájában. Mit határoz meg ez az automata, milyen sorrendben érkezhetnek a külső események?
3. Minden *Template* kinyitható a mellette lévő „+” ikonnal. Ekkor megtekinthetők az adott *Template*-hez tartozó deklarációk. Látható, hogy minden egyes *Template* tartalmaz egy *isActive* változót. Mit szabályoz ez a változó? Tipp: kezdetben csak a fő *Template* esetén igaz értékű, a többinél hamis.
4. Az UPPAAL Verifier ablakába írjuk be az `A[] not deadlock` követelményt, ami a modell holtpontmentességét fogalmazza meg. Végezzük el az ellenőrzést!
5. Írjuk be és ellenőrizzük az `E<> (Process_mainregionOfStatechart.OpenAndLocked && Process_mainregionOfStatechart.isActive)` követelményt! Ennek értelmezéséhez vegyük figyelembe a következőket: Az UPPAAL modellben az állapotokat precízen kell megadni, ehhez a következő elnevezési konvenció tartozik: szükséges az állapotot tartalmazó régió neve (*Process_mainregionOfStatechart*), majd magának az állapotnak a neve (*OpenAndLocked*). Ezek után fogalmazzuk meg természetes nyelven, mit ír elő ez a követelmény! A Yakindu állapottérkép modellre nézve ellenőrizzük, hogy a riasztó csak azután lesz élesítve (*Armed* állapot), ha becsukják az ajtót.
6. Mely *Template*-ek mely állapotai felelnek meg a Yakindu állapottérkép azon állapotkonfigurációjának amikor az *Armed, ArmedOpen, Quiet* állapotok aktívak?
7. Formalizáljuk és ellenőrizzük a következő lehetőséget: A vezérlő képes-e eljutni a kezdőállapotból abba az állapotba, ahol a riasztás már élesítve van, nem szól a riasztó, pedig az ajtó nyitva?
8. Nézzük meg az állapottérkép modellt, és válaszoljuk meg, hogy az előző pontban ellenőrzött állapotkonfiguráció elérhetősége miért nem jelenti azt, hogy a vezérlő terve hibás!

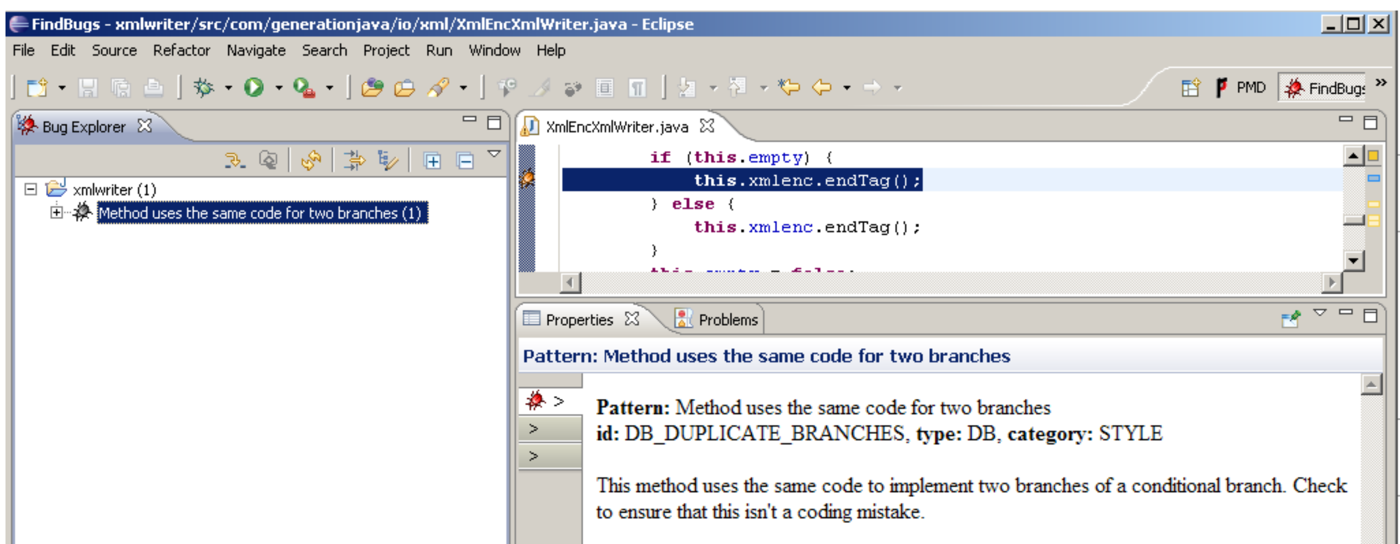
## Forráskód ellenőrzése

Forráskód ellenőrzéséhez két, Java forrásokhoz való statikus ellenőrző eszközt, a FindBugs-ot[2] és a PMD-t[3] fogjuk használni. A gyakorlat első felében használt Eclipse eszközben már fel lett telepítve a FindBugs és PMD Eclipse plugin verziója.

A gyakorlat második felében a *json.simple* nevű nyílt forrású projektet fogjuk megvizsgálni. A forráskódban már a fordító is talál problémákat, amiket warningok segítségével megjelöl, pl. nem elérhető kód, soha nem olvasott változó.

### *FindBugs*

1. Futtassuk le a FindBugs ellenőrzését: *jobb gomb a projekt nevén > FindBugs > Find Bugs*.
    a. A FindBugs esetén cél volt, hogy kevés téves hibát (false positve) jelezzen, így általában kevés dolgot jelöl, de azokkal érdemes is mindenképp foglalkozni.
    b. Váltsunk át a FindBugs nézetre (jobb felső sarokban található gombbal), és nézzük meg a hibák leírását, majd keressük ki a hozzájuk tartozó kódot. Valóban hibák ezek?
    c. Nézzük meg a projekt tulajdonságainál a FindBugs beállításait. Itt kapunk egy részletes listát arról, hogy milyen ellenőrzéseket hajt végre. Engedélyezzük, hogy ezt projekt szinten tudjuk szabályozni, állítsuk magasabbra a jelentés szintjét, hogy a kevésbé súlyosabb hibákat is jelezze. (Ezek a beállítások ilyenkor bekerülnek a *.settings* könyvtárban lévő FindBugs prefs fájlba, amit akár berakhatunk a verziókezelő rendszerbe, így minden fejlesztő ugyanazokat a szabályokat fogja használni.) Talált-e a FindBugs újabb hibát?
    d. Ha végeztünk, akkor *jobb gomb a projekt nevén > FindBugs > Clear Bug Markers* menüponttal rejtsük el a FindBugs hibajelzéseit.



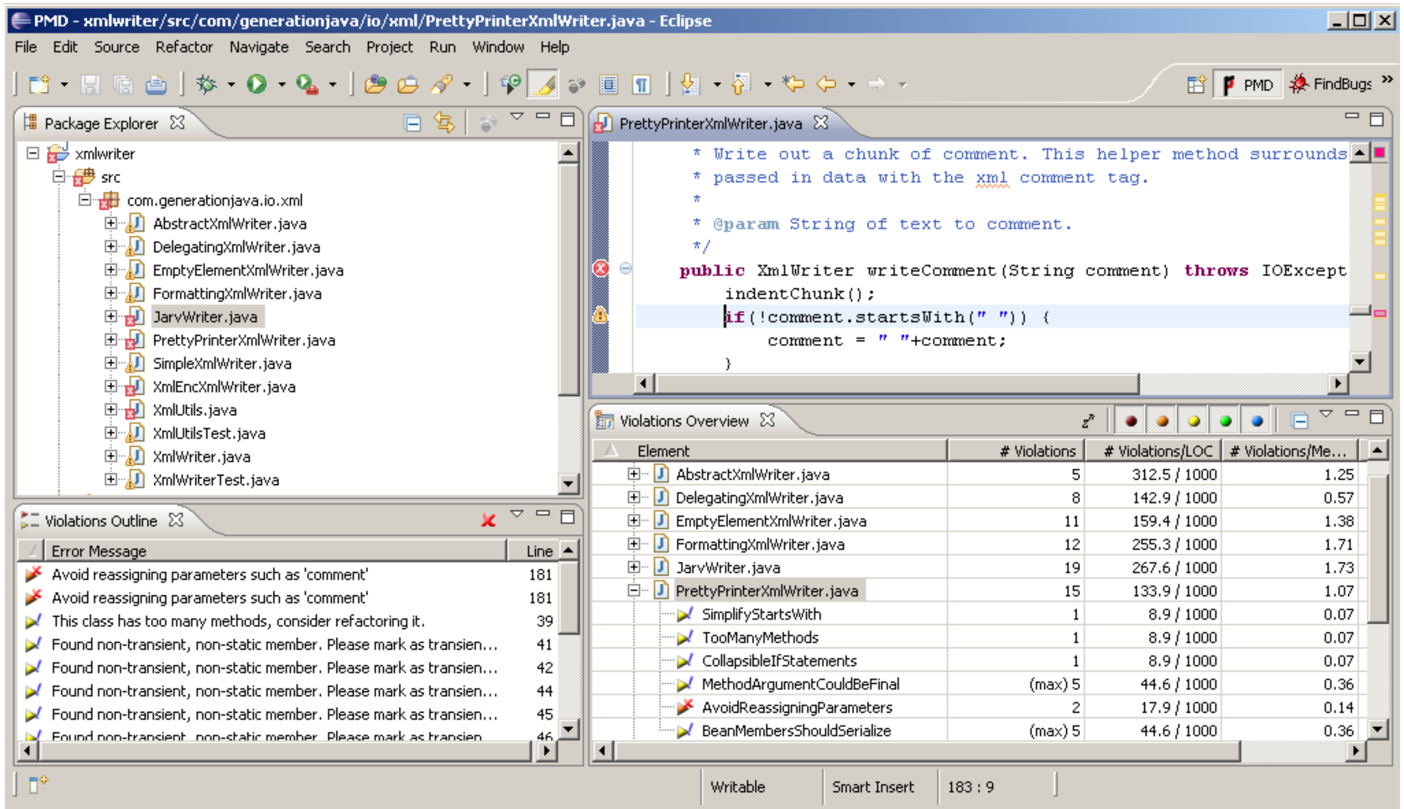

**1. ábra: A FindBugs Eclipse nézete**

### *PMD*

2. Futtassuk le a PMD ellenőrzését is: *jobb gomb a projekt nevén > PMD > Check code*

[2] http://findbugs.cs.umd.edu/eclipse
[3] http://sourceforge.net/projects/pmd/files/pmd-eclipse/update-site/

a. A PMD általában nagyszámú problémát jelez. Ezek egy része nem biztos, hogy gondot jelent az adott projektben, így érdemes testreszabni a szabálykészletét mindig az adott projekthez. Ezért nagyon fontos, hogy már a fejlesztés legelején használjuk a statikus ellenőrző eszközt. Ha 1000 sor forrás megírása után indítjuk el először, akkor már sokkal nehezebb az 50-100 hiba kijavításának nekiállni.




**2. ábra: A PMD Eclipse nézete**

b. Váltsunk át a PMD nézetbe, és nyissuk meg az `org.json.simple.parser.JSONParser` fájlt.

   i. Nézzük meg, hogy milyen típusú hibákat talált a fájlban!

   ii. Nézzük meg az „Avoid instantiating Integer objects…" hiba részletes leírását (*jobb gomb > Show Details*). A hibák leírásánál mindig találunk egy rövid indoklást és példát, valamint egy URL-t a hibatípus hivatalos leírására. Miért javasolja a forrás módosítást ebben az esetben?

   iii. Ha megnéztünk egy adott hibát, és úgy döntünk, hogy az adott esetben nem gond, akkor lehet a „Mark as reviewed" opcióval lehet ezt külön jelölni (ilyenkor bekerül egy speciális `//NOPMD` komment az adott sorhoz). Jelöljük meg az egyik hibát így, azonban ne felejtsünk el indoklást is írni hozzá!

   iv. Ha úgy gondoljuk, hogy egy szabályt egyáltalán nem akarunk használni, akkor azt a projekt tulajdonságainál ki lehet kapcsolni. Példaként kapcsoljuk ki az egyik szabály!

   v. A PMD képes az egy az egyben átmásolt kódrészletek azonosítására. Keressünk ilyen kódrészleteket a projektben (*PMD > Find Suspect Cut and Paste*)!

   vi. Nézzük át a többi fájlban szereplő hibatípusokat, hogy pontosabb képet kapjunk arról, hogy milyen hibák megtalálásában segíthet minket egy ilyen eszköz!

vii. Nézzük meg a projekt beállításainál a PMD-re vonatkozó részt. Itt lehetne egyesével ki- és bekapcsolni az egyes ellenőrzési szabályokat. Fussuk át, hogy miket tud vizsgálni a PMD!

viii. Az eddigiek a PMD-nek még csak egy kis szeletét mutatták. Kapcsoljuk be az összes szabályt (ez jelenleg több mint 300 szabály!), és így is futtassunk egy ellenőrzést.

## *További információ*

Az alábbi cikkben John Carmack, a Doom és a Quake játékok vezető fejlesztője érvel amellett a saját tapasztalatai alapján, hogy felelőtlenség nem használni statikus analízis eszközöket:

> John Carmack. „In-Depth: Static Code Analysis", Gamasutra, December 27, 2011. URL: http://www.gamasutra.com/view/news/128836/InDepth_Static_Code_Analysis.php

A következő cikk egy érdekes összefoglaló, hogy a Google belül hogyan használja a FindBugs eszközt:

> N. Ayewah *et al.* „Experiences Using Static Analysis to Find Bugs", IEEE Software, vol. 25 (2008), pp. 22-29, URL: http://research.google.com/pubs/pub34339.html

Ebben a cikkben pedig a Coverity statikus analízis eszköz fejlesztői osztják meg a tapasztalataikat:

> Bessey et al. „A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World", Comm. of the ACM, Vol. 53 No. 2, pp. 66-75. DOI: 10.1145/1646353.1646374