

## 2. gyakorlat: Részletes tervek és forráskód ellenőrzése

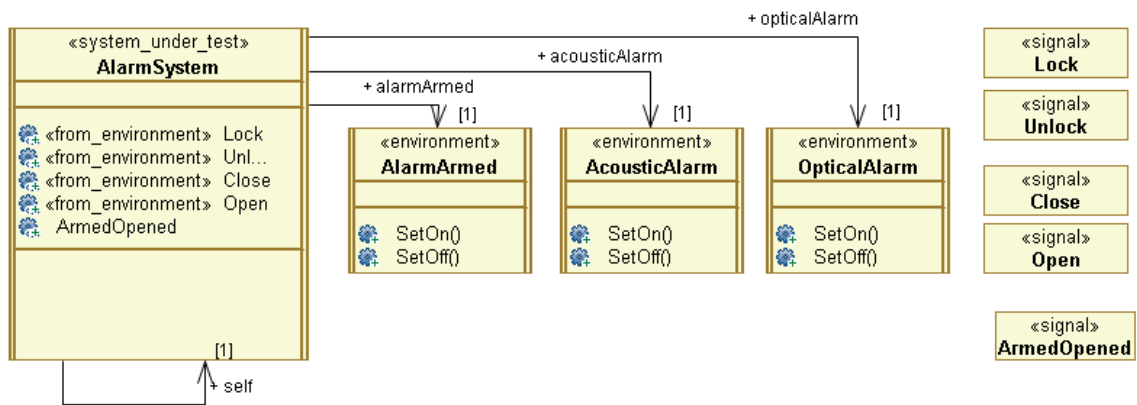
A gyakorlaton a részletes tervek ellenőrzésével és a forráskód verifikációját végző statikus ellenőrző eszközökkel fogunk foglalkozni.

### Részletes tervek ellenőrzése

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

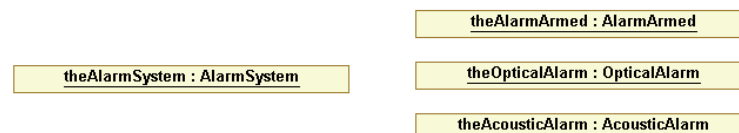
- Az UML modell megtekintéséhez indítsuk el a Papyrus eszközt. Válasszuk ki a felajánlott *workspace* könyvtárat. A megnyíló felületen a bal felső ablakban található a modell fájlok, a bal alsó ablakban a modell elemei között lehet navigálni. A jobb felső (nagy) ablakban látható a modell egy diagramja, a jobb alsó ablakban pedig egyes kiválasztott elemek tulajdonságai.
- Válasszuk ki bal oldalon az *AlarmSystem.di2* modellt (diagramot). Ez egy gépkocsi riasztóberendezésének modelljét tartalmazza. A modell kiválasztása után a fő panel alján nézzük végig a következő nézeteket:

- Az osztályok a *Context and Classes* nézeten láthatók. A vizsgálandó vezérlő az *AlarmSystem* osztály. A külső (környezetből érkező) eseményeket `<<signal>>` sztereotípiával azonosítottuk (*Open* és *Close*: ajtó nyitása és zárása, *Lock* és *Unlock*: riasztó indítása és leállítása). Ezek mint „stimulusok” a *Requirements* nézeten is láthatók.



1. ábra: A modell osztályai (*Context and Classes* nézet)

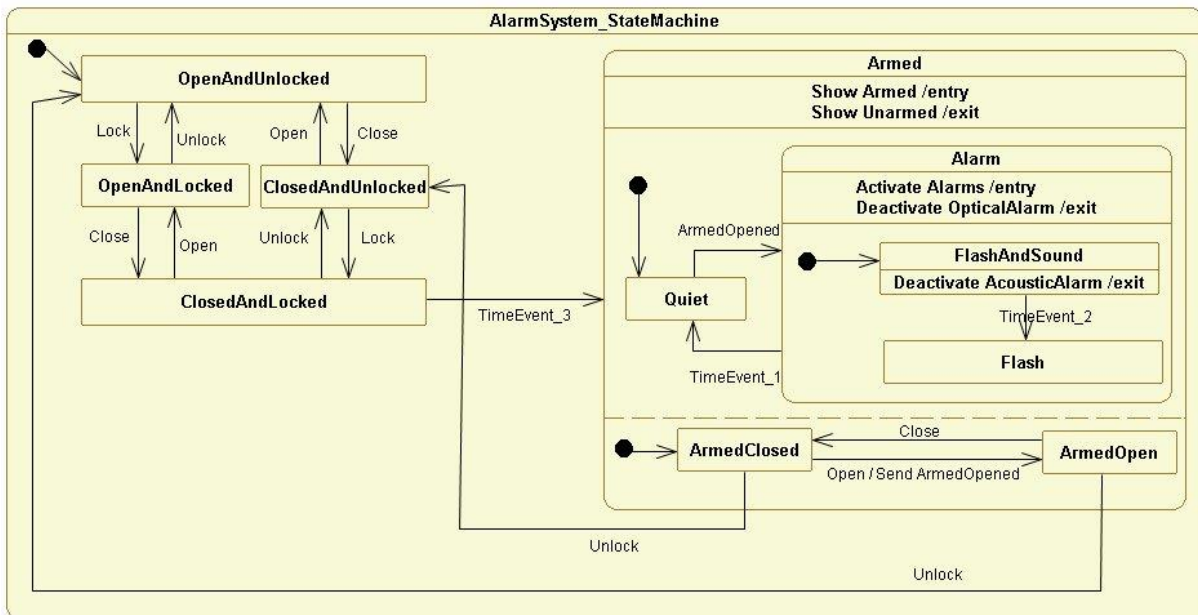
- A modell konkrét objektumai az *Initialisation* nézeten találhatóak.



2. ábra: A modell objektumai (*Initialisation* nézet)

- Az *SM\_AlarmSystem* nézeten látszik a vezérlő állapotterképe az állapotok és trigger események intuitív elnevezésével. Tanulmányozzuk az állapotterképet! A *TimeEvent\_\** elnevezések egy-egy időzítő eseményre utalnak, az akciók és őrfeltételek egy

implementáció-független AGSL (Action and Guard Specification Language) nyelven vannak megadva (ezt egy akcióval ellátott állapotátmenet, entry avagy exit esemény kiválasztásakor a lenti tulajdonság ablakban figyelhetjük meg).



3. ábra: Az AlarSystem osztály állapotterképe (SM\_AlarmSystem nézet)

A modell átnézése után transzformáljuk a modellt az UPPAAL által elfogadott formális modellé, azaz időzített automatává.

1. Ehhez az *AlarmSystem.uml* fájlban a jobb egérgombot lenyomva válasszuk ki a *State Machines / Build SMTE Model* menüpontot. Így keletkezik egy *AlarmSystem.smtf* fájl (ez egy belső modell reprezentáció).
2. Ezután ezen az *AlarmSystem.smtf* fájlban a jobb egérgombbal már a *State Machines / Transform to Uppaal* menüpontot választhatjuk ki, és a formális modell előáll (a felbukkanó ablakban nem szükséges a *coverage analysis related code* generálását kérni – ez majd teszteléshez lesz fontos)!

Az ellenőrzéshez nyissuk meg a keletkezett *AlarmSystem.xml* modellt (a Papyrus workspace *com.ford.mogentes.cas* könyvtárából) az UPPAAL eszközben, és gondoljuk végig a válaszokat a következő kérdésekre, illetve végezzük el a feladatokat:

1. Az UML állapotterkép modell nem rögzíti a környezet viselkedését (hogyan érkehetnek 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 *EnvironmentTemplate* automatájában: a környezet az *eqInsertTail* funkcióval illeszt egy-egy újabb eseményt az eseménysor végére. Mit határoz meg ez az automata, milyen sorrendben érkehetnek a külső események?
2. Nézzünk rá a vezérlőt leíró *AlarmSystemTemplate* automatára! Vajon miért ilyen bonyolult? Tipp: Gondoljunk az UML állapotterképek szemantikájára. Miért szükséges olyan sok DROP akcióval ellátott átmenet, ami egy-egy esemény eldobását modellelzi? Hogyan oldhatók fel egy olyan alacsony szintű formalizmusban, mint ez az automata, a hierarchikus állapotok és konkurens régiók?

3. Az UPPAAL Verifier ablakába írjuk be az  $A[] \text{ not deadlock}$  követelményt, ami a modell holtponmentességét fogalmazza meg. Végezzük el az ellenőrzést!
4. Írjuk be és ellenőrizzük az  $E\langle \rangle \text{ theAlarmSystemProcess.STABLE\_OpenAndLocked}$  követelményt! Ennek értelmezéséhez vegyük figyelembe a következőket: Az UPPAAL modellben az állapotokat precízen kell megadni, ehhez a következő elnevezési konvenció tartozik: szükséges az objektumhoz tartozó automata példány neve (*theAlarmSystemProcess*), majd annak megadása, hogy stabil állapotkonfigurációról van szó (*STABLE*), majd maga az állapotkonfiguráció az állapottérkép modell alapján (itt egyszerűen az *OpenAndLocked* állapot). Ezek után fogalmazzuk meg természetes nyelven, mit ír elő ez a követelmény! Az UML állapottérkép modellre nézve ellenőrizzük, hogy a riasztó csak azután lesz élesítve (*Armed* állapot), ha becsukják az ajtót.
5. Egy összetett állapotkonfiguráció esetén ennek nevét az egyes alállapotok \_ jellel elválasztott egymás után írásával állítja elő a transzformáció. Így például egy összetett állapotkonfiguráció a *theAlarmSystemProcess.STABLE\\_Armed\\_Quiet\\_ArmedClosed* - keressük meg ezt az állapotkonfigurációt az állapottérkép modellen!
6. 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?
7. Nézzük meg az állapottérkép modellt, és válaszoljuk meg, hogy az előző pontban ellenőrzött állapotkonfiguráció elérhetősége miért nem jelenti azt, hogy a vezérlő terve hibás!

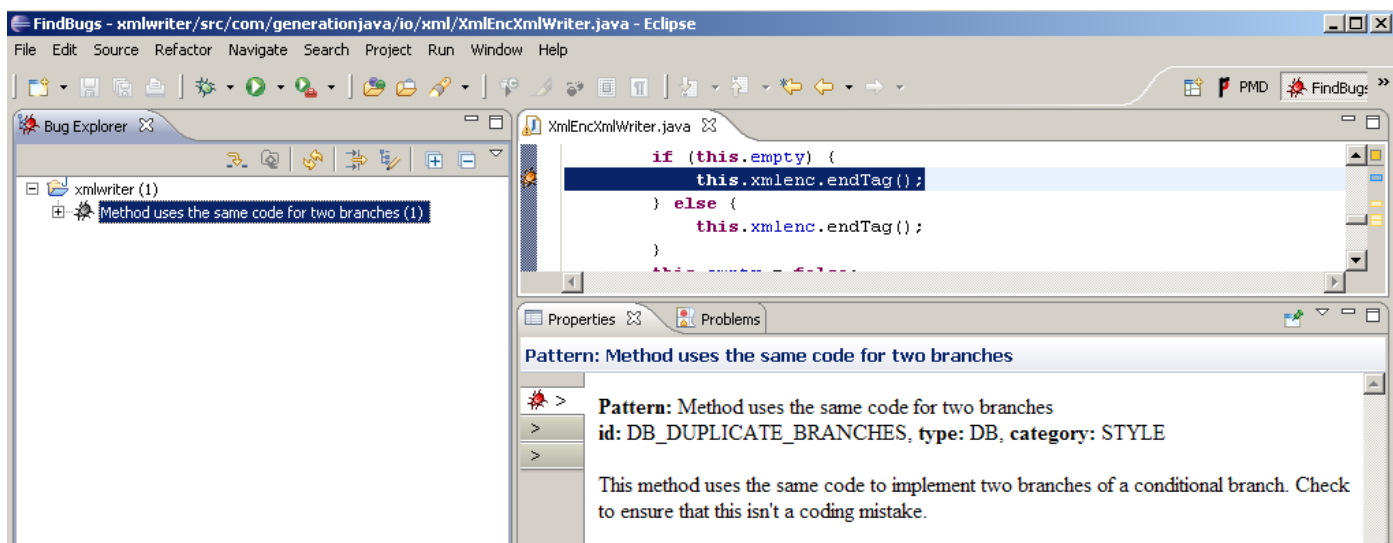
## Forráskód ellenőrzése

Forráskód ellenőrzéséhez két, Java forrásokhoz való statikus ellenőrző eszközt, a FindBugs-ot és a PMD-t fogjuk használni.

1. Indítsuk el az Eclipse-et, és nyissuk meg a C:\code\GYAK2 workspace-t. Ebben az Eclipse példányban már fel lett telepítve a FindBugs és PMD Eclipse plugin verziója. Ezek elérhetősége:
  - a. FindBugs - <http://findbugs.cs.umd.edu/eclipse>
  - b. PMD - <http://sourceforge.net/projects/pmd/files/pmd-eclipse/update-site/>
2. A gyakorlat során a `json.simple` nevű nyílt forrású projektet fogjuk megvizsgálni.
  - a. A forráskódban már a fordító is talál problémákat, amiket warningok segítségével megjelöl, pl. nem elérhető kód, soha nem olvasott változó.

### FindBugs

3. Futtassuk le a FindBugs ellenőrzését: *jobb gomb a projekt nevéen > FindBugs > Find Bugs*.
  - a. A FindBugs esetén cél volt, hogy kevés téves hibát (false positive) jelezen, így általában kevés dolgot jelöl, de azokkal érdemes is mindenképp foglalkozni.
  - b. Váltunk át a FindBugs nézetre, 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 nevéen > FindBugs > Clear Bug Markers* menüponttal rejtjük el a FindBugs hibajelzéseit.

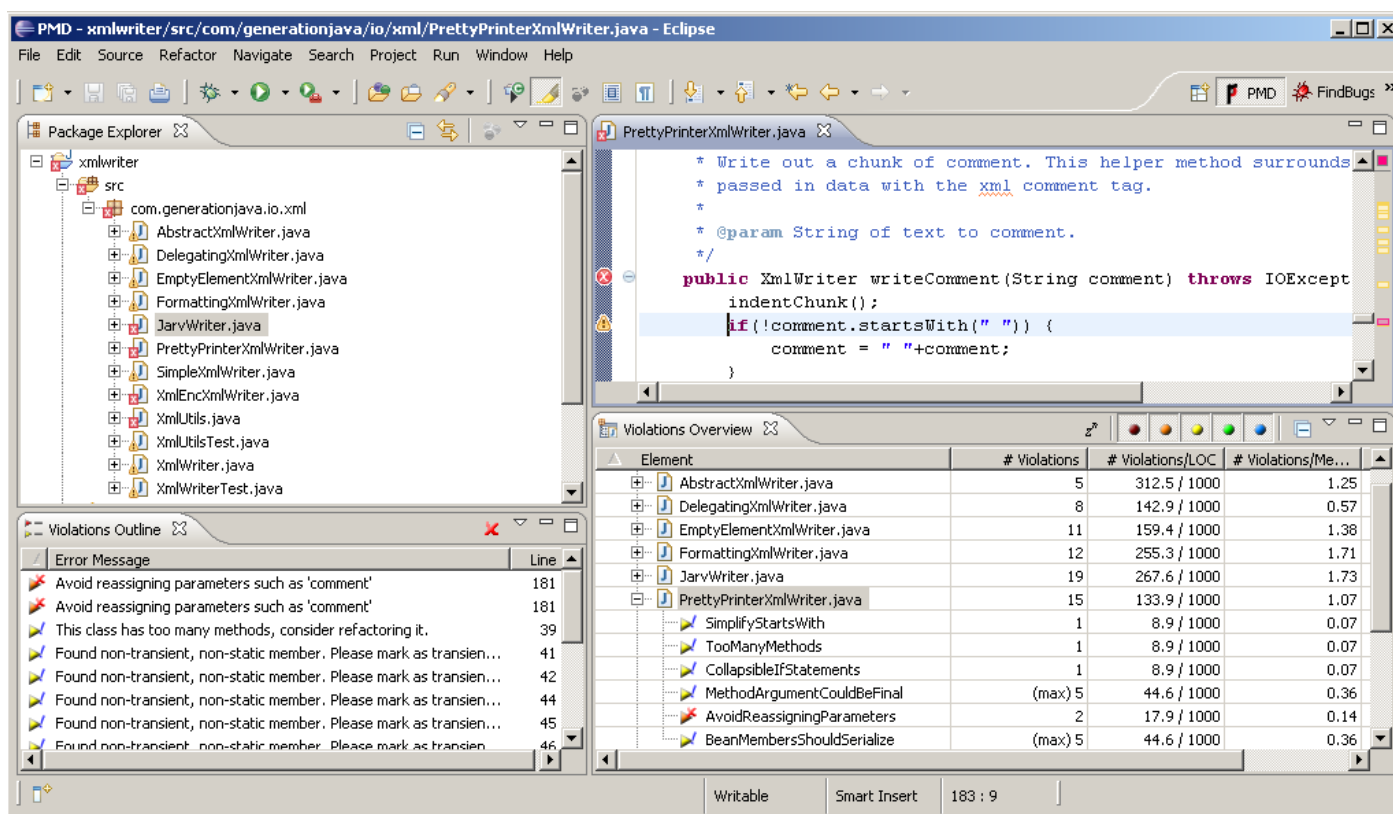


1. ábra: A FindBugs Eclipse nézete

## PMD

4. Futtassuk le a PMD ellenőrzését is: *jobb gomb a projekt nevéen > PMD > Check code*

- 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. Váltunk 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 felejtünk 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 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