Szerverfarm Felügyeleti Rendszer
Végleges

Halász Olivér

2025.05.10

A programozás alapjai II. Nagy házi feladat

Budapesti Műszaki és Gazdaságtudományi Egyetem II. félév

Tartalomjegyzék

1. Bevezetés

Ez a dokumentáció a A programozás alapjai 2 házi feladatának, a szerverfarm felügyeleti logikai rendszernek végleges, teljes dokumentációját tartalmazza. A rendszer célja, hogy a szerverfarm kritikus állapotairól különféle szenzorok, logikai kapuk és vészcsengők (riasztók) segítségével átfogó felügyeletet biztosítson.

2. Feladat

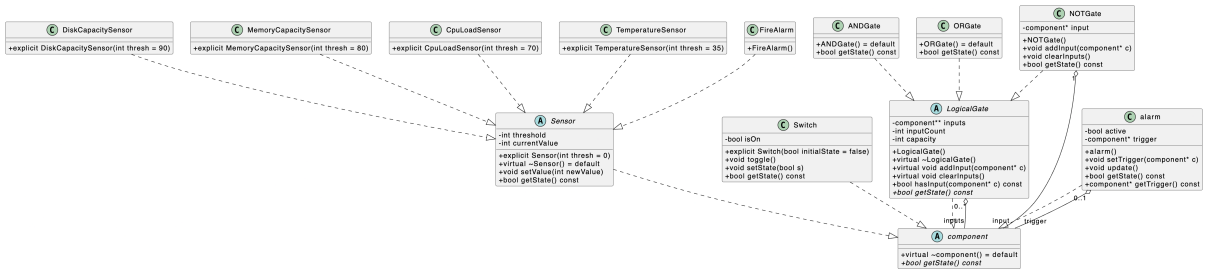
Szerverfarm

Tervezzen objektummodellt számítógépek üzemeltetését segítő felügyeleti rendszer működésének modellezésére! A modellben legyenek érzékelők (diszk kapacitás, memória kapacitás, processzor terheltség, szerverszoba hőmérséklet, tűzjelző, stb.), logikai kapuk (és, vagy, nem) kapcsolók, és vészcsengő! Tetszőlegesen bonyolult modell legyen felépíthető a komponensek és a logikai kapuk egyszerű összekapcsolásával! Demonstrálja a működést külön modulként fordított tesztprogrammal! A megoldáshoz ne használjon STL tárolót!

3. Objektumterv

A rendszer alapvetően az alábbi osztályokból és interfészekből áll:

3.1. UML osztálydiagram



3.2. Főbb osztályok rövid leírása

- **component**: Absztrakt ős, minden komponens ebből származik. Tartalmaz egy virtuális `getState()` függvényt, ami a komponens aktuális logikai állapotát adja vissza.
- **Sensor**: Absztrakt szenzorosztály, egy küszöbértékkel és aktuális értékkel. Leszármazottak konkrét szenzorok (Disk, Mem, CPU, Temp, Fire).
- **Switch**: Manuális, felhasználó által állítható logikai kapcsoló.
- **LogicalGate**: Absztrakt logikai kapu, bemenetként más komponenseket fogadhat. Leszármazottak: AND, OR, NOT.

- **alarm**: Vészcsengő, egy komponenshez (triggerhez) csatlakozva, annak állapotától függően aktív vagy inaktív.

4. Főbb algoritmusok és működés

4.1. Komponensek kezelése

A komponenseket egy maximum 100 elemű statikus tömbben tároljuk, minden új komponens az első szabad helyre kerül, egyedi ID-t kap.

4.2. Parancsértelmezés és vezérlés

A főprogram CLI-parancsokat értelmez (pl. `create_sensor`, `create_switch`, `connect`, `set`, `toggle`, `status`, `remove`, stb.). Minden parancs részletes súgóval rendelkezik.

4.2.1. Parancsok áttekintése

- **create_sensor TYPE [KÜSZÖBÉRTÉK]** – új szenzor létrehozása adott típusban és küszöbbel.
- **create_switch 0—1** – új kapcsoló kezdeti állapot szerint.
- **create_gate AND—OR—NOT** – új logikai kapu.
- **create_alarm** – új riasztó.
- **connect SRC_ID DST_ID** – komponensek összekötése (logikai kapu bemenetek, riasztó triggerének beállítása).
- **set ID VALUE** – szenzor értékének explicit beállítása.
- **set_switch ID 0—1**, **toggle ID** – kapcsoló explicit vagy átbillentett állapota.
- **clear_inputs GATE_ID** – logikai kapu bemeneteinek törlése.
- **status** – aktuális állapotok listázása.
- **remove ID** – komponens törlése, csak ha nincs rá aktív hivatkozás.

4.3. Logikai hálózat működése

A rendszer fő logikai működése:

- A szenzor értékének beállításakor a `getState()` függvény logikai értéket ad vissza a küszöbhöz viszonyítva.
- A logikai kapuk (AND, OR) tetszőleges számú bemenettel dolgoznak, `getState()` a bemeneti komponensek állapota szerint értékel.
- A NOTGate mindig egy bemenetet vár, és logikailag invertálja azt.
- Az alarm egy komponenshez kapcsolható (`setTrigger`), állapotát (`active`) update során a trigger aktuális állapotához igazítja.

4.4. Állapotkezelés és frissítés

Az alarmok frissítését egy háttérszál végzi, amely időközönként (másodpercenként) meghívja az összes alarm `update()` metódusát. Manuális vagy automatikus (status parancs) állapotfrissítés is lehetséges.

5. Osztályok részletes dokumentációja

5.1. component (absztrakt ős)

```
class component {
public:
    virtual ~component() = default;
    virtual bool getState() const = 0;
};
```

5.2. Sensor és leszármazottak

```
class Sensor : public component {
protected:
    int threshold;
    int currentValue;
public:
    explicit Sensor(const int thresh = 0);
    void setValue(int newValue);
    bool getState() const override;
    // ...
};
```

Leszármazottak:
`DiskCapacitySensor`, `MemoryCapacitySensor`, `CpuLoadSensor`, `TemperatureSensor`, `FireAlarm`

5.3. Switch

```
class Switch final : public component {
private:
    bool isOn;
public:
    explicit Switch(bool initialState = false);
    void toggle();
    void setState(bool s);
    bool getState() const override;
};
```

5.4. LogicalGate és leszármazottai

```
class LogicalGate : public component {
```

```
protected:
    component** inputs;
    int inputCount;
    int capacity;
public:
    LogicalGate();
    virtual ~LogicalGate();
    virtual void addInput(component* c);
    virtual void clearInputs();
    bool hasInput(component* c) const;
    virtual bool getState() const = 0;
};

class ANDGate final : public LogicalGate {
public:
    bool getState() const override;
};

class ORGate final : public LogicalGate {
public:
    bool getState() const override;
};

class NOTGate final : public LogicalGate {
private:
    component* input;
public:
    NOTGate();
    void addInput(component* c) override;
    void clearInputs() override;
    bool getState() const override;
};
```

5.5. alarm

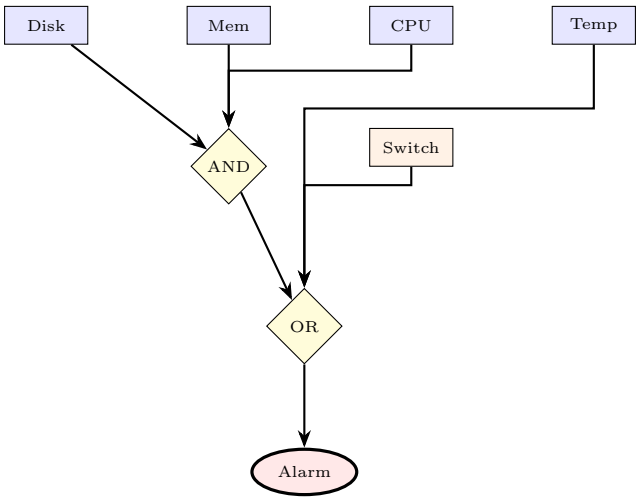
```
class alarm final : public component {
private:
    bool active;
    component* trigger;
public:
    alarm();
    void setTrigger(component* c);
    void update();
    bool getState() const override;
    component* getTrigger() const;
};
```

6. Felhasználói felület (CLI) és működési példa

6.1. Parancssori segítség (help parancs kimenete)

```
[ÁLTALÁNOS PARANCSOK]
help                    -- súgó
list                    -- jelenlegi komponensek listája
exit                    -- program befejezése

[KOMPONENS LÉTREHOZÓ PARANCSOK]
create_sensor TYPE KÜSZÖB -- új szenzor (Disk, Mem, CPU, Temp, Fire)
create_switch 0|1         -- új kapcsoló
create_gate AND|OR|NOT    -- új logikai kapu
create_alarm              -- új riasztó

[KOMPONENSEK ÖSSZEKAPCSOLÁSA]
connect FORRÁS_ID CÉL_ID  -- két komponens összekötése (logikai bemenet, riasztó trigger)

[KOMPONENSEK MÓDOSÍTÁSA]
set SZENZOR_ID ÉRTÉK      -- szenzor értékének beállítása
set_switch ID 0|1         -- kapcsoló explicit állapot
toggle ID                 -- kapcsoló átbillentése
clear_inputs GATE_ID      -- logikai kapu bemeneteinek törlése
remove ID                 -- komponens törlése
update                    -- riasztók manuális frissítése

[ÁLLAPOT LEKÉRDEZÉSE]
status                    -- komponensek aktuális állapota
```

6.2. Mintahálózat



A mintahálózat értelmezése

A fenti ábra egy egyszerűsített szerverfarm felügyeleti logikát mutat be. Célja, hogy szemléltesse a rendszer főbb komponenseinek (szenzorok, kapcsoló, logikai kapuk, riasztó) összekapcsolhatóságát és alapvető működését.

Komponensek és logika:

- **Szenzorok** (kék: `Disk`, `Mem`, `CPU`, `Temp`): Adott küszöbérték felett 'IGAZ'állapotot vesznek fel.
- **Switch** (narancs): Manuálisan állítható, 'BE'vagy 'KI'állapotú.
- **Logikai kapuk** (sárga):
  - `AND`: Akkor 'IGAZ', ha minden bemenete 'IGAZ'. Itt a `Disk`, `Mem` és `CPU` szenzorok állapotát összesíti.
  - `OR`: Akkor 'IGAZ', ha legalább egy bemenete 'IGAZ'. Ez fogadja az `AND` kapu kimenetét, a `TempSensor` és a `Switch` állapotát.
- **Alarm** (piros): Akkor aktiválódik ('RIASZT'), ha a hozzá kapcsolt `OR` kapu kimenete 'IGAZ'.

Működés röviden: A riasztó akkor szólal meg, ha: (a `Disk`, `Mem` ÉS `CPU` egyszerre kritikus) VAGY (a `TempSensor` jelez) VAGY (a `Switch` be van kapcsolva). Ez a láncolat bemutatja, hogyan lehet összetett feltételeket modellezni.

Fontos tudnivalók:

- Az ábrán az egyszerűség kedvéért nincsenek feltüntetve a komponensek egyedi ID-jai.
- A `NOT` kapu – bár fontos része a rendszernek – ebből a példából kimaradt a kompaktság érdekében. w
- A hálózat tetszőlegesen bővíthető további elemekkel a valós igényeknek megfelelően.

A minta célja a rendszer alapvető működési elvének és a komponensek közötti interakcióknak a demonstrálása.

7. Tesztelés

7.1. Tesztelési stratégia

A teljes rendszert részletesen leteszteltem, unit-tesztekkel és funkcionális tesztekkel ellenőriztem:

- Szenzor küszöbérték-logika helyes működése (`setValue`, `getState`)
- Kapcsolók állapotváltása (`toggle`, `setState`)
- Logikai kapuk működése (AND, OR, NOT különböző bemeneti állapotokkal)
- Riasztók triggerhez kapcsolása, és állapotfrissítés (update)
- Összetett logikai hálózatban több komponens együttműködésének tesztje
- Hibakezelés (érvénytelen bemenetek, nem létező komponensek, duplikált kapcsolatok, stb.)

7.2. Automatizált unit-teszt példa (GoogleTestLite):

```
TEST(Sensor, ValueAndThresholdLogic) {
    DiskCapacitySensor disk(80);
    disk.setValue(50);
    EXPECT_EQ(false, disk.getState());
    disk.setValue(85);
    EXPECT_EQ(true, disk.getState());
    // stb.
}

TEST(Switch, ToggleFunctionality) {
    Switch sw(false);
    EXPECT_EQ(false, sw.getState());
    sw.toggle();
    EXPECT_EQ(true, sw.getState());
}

TEST(ANDGate, LogicalOperation) { ... }
TEST(ORGate, LogicalOperation) { ... }
TEST(NOTGate, SingleInputAndOverwrite) { ... }

TEST(Alarm, RespondsToTriggerChange) { ... }

// ...
```

7.3. Memóriakezelés ellenőrzése

A rendszer minden fordítási egységben tartalmazza a `memtrace` modult, amely futás közben ellenőrzi, hogy nincs-e memóriaszivárgás. Futtatáskor minden dinamikusan lefoglalt memória felszabadul, a memtrace nem jelez hibát.

8. Összegzés

Az elkészült szerverfarm-felügyeleti logikai rendszer rugalmasan bővíthető, moduláris felépítésű és könnyen tesztelhető. Az összes követelménynek megfelel, a komponensek önállóan, vagy egymással kombinálva is kipróbálhatók. A rendszer stabil, nem tartalmaz memóriaszivárgást, minden funkció részletesen dokumentált és unit-tesztekkel igazolt.