PROBLEM SOLVER

מגישים: יקטרינה בטשוב ואלעד גרבר

תיאור המערכת

המערכת שבנינו נועדה לעזור בפתירת בעיות מתמטיות ואלגוריתמיות שונות.

המערכת נבנתה עבור סטודנטים, אך פתוחה לשימוש לכל אדם.

האינטרקציה עם המשתמש מתבצעת בעזרת הCLI, אך בקלות ניתן להוסיף GUI מכל סוג שהוא.

ראשית, המשתמש בוחר איזה סוג בעיה הוא רוצה לפתור.

סוג הבעיות שקיימות במערכת וניתן לפתור הן: חישוב ביטויים מתמטיים ומציאת מסלול קצר במטריצות.

```
Choose a solver:
(1) MatrixSolver
(2) MathSolver
```

לאחר שהמשתמש בוחר בסוג הבעיה שהוא מעוניין לפתור, הוא יבחר את האלגוריתם שאיתו הוא מעוניין שהמערכת תפתור את הבעיה.

```
Choose a solver:
(1) MatrixSolver
(2) MathSolver
1
Choose an algorithm to solve with:
(1) A*
```

לאחר מכן, המשתמש יבחר את הדרך שבו הוא מזין את הבעיה, למשל דרך העלאת קובץ או כתיבת הבעיה בCommand Prompt.

```
Choose a solver:
(1) MatrixSolver
(2) MathSolver
1
Choose an algorithm to solve with:
(1) A*
1
Choose a way to upload the problem:
(1) CommandPromptParser
```

לבסוף, לאחר הכנסת הבעיה על ידי המשתמש, המערכת תחשב את הפתרון ותציג אותו למשתמש:

```
Enter the problem: (write 'end' to finish)1,1,1,1
-1,2,3,2
-1,1,2,2
0,0
2,2
end
The solution is:
 Right (2.00), Down (4.00), Down (5.00), Right (7.00)
```

```
Enter the problem: (write 'end' to finish)2+3*7
end
The solution is:
 23.000000
```

תבניות העיצוב בהן השתמשנו

המערכת שלנו כוללת את תבניות העיצוב הבאות:

- Bridge (Structural)
- Strategy (Behavioral)
- Factory (Creational)
- Composite (Structural)
- Singleton (Creational)

BRIDGE

התבנית Bridge עוזרת לנו להפריד בין כל סוגי הבעיות ובין האלגוריתמים שפותרים אותם. כיוון שיש לנו הרבה מאוד בעיות והרבה מאוד אלגוריתמים שונים, במקום לממש מחלקה שתפתור בעיה ספציפית, למשל מציאת מסלול קצר ביותר בעזרת A*, מציאת מסלול קצר ביותר בעזרת BFS וכו', נפריד בין האלגוריתם שפותר את הבעיה לבעיה עצמה.

נשים לב שאף בעיה איננה תלויה באלגוריתם שפותר אותה. בכל ריצה ניתן להזריק לאותה הבעיה אלגוריתם אחר שיפתור אותה.

כמו כן, האלגוריתמים לפתירת הבעיות גם הם אינם תלויים בבעיות. במידה ובעתיד נרצה להשתמש באלגוריתמים בפרויקטים אחרים, נוכל להשתמש במימוש זה של האלגוריתמים, מבלי לשנות דבר.

במידה ונרצה להוסיף אלגוריתם חדש או בעיה חדשה – לא נצטרך להיכנס לקוד ולשנות קוד קיים.

דוגמא מהקוד – BRIDGE

```
class Solver {
public:
    virtual string solve(const string &problem) = 0;
    virtual ~Solver()= default;
};
```

ניתן לראות את שבSolver ישנה פונקציה וירטואלית Solve שמקבלת את בעיה כString ומחזירה תשובה כString.

```
class MatrixSolver : public Solver {
private:
    MatrixSearcher *_searcher;
public:
    MatrixSolver(Searcher *s): _searcher(nullptr){
        if(s != nullptr && s->getType() == "MatrixSearcher"){
            _searcher =(MatrixSearcher*) s;
        }
    }

    virtual string solve(const string &problem) {
        string solution;
        try {
            solution = _searcher->search(problem);
            Logger::getInstance()->log( message: "Found a solution using MatrixSolver...\n" + solution);
        } catch (const string &e) {
            Logger::getInstance()->log( message: "An error has occurred\n" + e);
            solution = e;
        } catch (...) {
            Logger::getInstance()->log( message: "An error has occurred...");
            solution = "An error has occurred.";
        }

        return solution + "\n";
    }
```

Matrix Solver הינה אחת מסוגי הבעיות שמימשנו שפותרת את בעיית מציאת מסלול קצר ביותר במטריצה. המחלקה מקבלת בConstructor את סוג האלגוריתם שבעזרתו תפתור את הבעיה, ובSolve משתמש בו. נשים לב שהמחלקה Matrix Solver איננה מודעת למי האלגוריתם או כיצד הוא פועל.

דוגמא מהקוד – BRIDGE

```
class Searcher {
public:
    virtual string search(const string &searchable) = 0;

    virtual ~Searcher() = default;
};
```

ניתן לראות את שבSearcher ישנה פונקציה וירטואלית Search שמקבלת את בעיה כString ומחזירה תשובה כString.

```
class MatrixSearcher : public Searcher {
protected:

    MatrixMaze createProblemFromString(const string &str);

public:

    virtual string search(const string &matrixMazeStr) = 0;

    virtual ~MatrixSearcher() = default;

};
```

Matrix Searcher הינה מחלקה אבסטרקטית שבה ישתמשו כל הבעיות אשר משתמשות במטריצה.
המחלקה אינה מממשת את Search, אך היא מממשת את הפונקציה CreateProblemFromString, בהינתן בעיה בצורת String המחלקה ממירה את הבעיה למטריצה שעליה היא תדע לעבוד.

```
class Astar : public MatrixSearcher {
public:
    virtual string search(const string &matrixMazeStr);

    virtual ~Astar()= default;

};
```

האלגוריתם A* יורש מMatrix Searcher וממש את הפונקציה Search. הוא מוצא את המסלול הקצר ביותר ומחזיר תשובה כString.

BRIDGE – דוגמא מהקוד

ניתן לראות את שבSearcher ישנה פונקציה וירטואלית Search שמקבלת את בעיה כString ומחזירה תשובה כString.

Matrix Searcher הינה מחלקה אבסטרקטית שבה ישתמשו כל הבעיות אשר משתמשות במטריצה.
המחלקה אינה מממשת את Search, אך היא מממשת את הפונקציה CreateProblemFromString, בהינתן בעיה בצורת String המחלקה ממירה את הבעיה למטריצה שעליה היא תדע לעבוד.

האלגוריתם A* יורש Matrix Searcherמ וממ מש את הפונקציה Search. הוא מוצא את המסלול הקצר ביותר ומחזיר תשובה כString.

STRATEGY

בתבנית strategy מי שמריץ את הפקודה מופרד מהאובייקט שמבצע אותה. למי שמריץ את הstrategy אין מידע על האובייקט שהוא מריץ.

כיוון שאנחנו רוצים לממש את אותה הפעולה - לפתור בעיה כלשהי, אך נרצה לממש לפתור כל בעיה בצורה שונה, התבנית strategy עוזרת לנו לממש זאת.

לדוגמא, אם הגדרנו את הsolver להיות MatrixSolver אז כשנפעיל את solver->solve() תופעל הפונקציה הווירטואלית של MatrixSolver, שתפתור את בעיית המטריצה אך אם נקרא לMathSolver נרצה שהיא תפתור בעיה שונה, ולכן המימוש שונה.

נשים לב שmenu אינו מודע איזה solver הוא מפעיל, וכל solver ספציפי מממש את solve בצורה המתאימה לו.

אובייקט הstrategy שלנו מתקבל בזמן ריצה על ידי בחירת המשתמש.

דוגמא מהקוד – STRATEGY

```
class Menu {
protected:
    Solver * _currentSolver= nullptr;
    Parser * _currentParser=nullptr;
```

```
class Solver {
public:
    virtual string solve(const string &problem) = 0;
    virtual ~Solver()= default;
};
```

```
bool CommandPromptMenu::solveProblem() {
    Logger::getInstance()->log( message: "Choosing problem at command prompt menu...");
    setSolver();
    setParser();
    string problem = this->_currentParser->parse();
    string solution = this->_currentSolver->solve(problem);

    cout << "The solution is:\n " + solution << endl;
    return doYouWantToSolveAgain();
}
```

```
class MatrixSolver : public Solver {
private:
    MatrixSearcher *_searcher;
public:
    MatrixSolver(Searcher *s): _searcher(nullptr){
        if(s != nullptr && s->getType() == "MatrixSearcher"){
            _searcher =(MatrixSearcher*) s;
        }
    }

    virtual string solve(const string &problem) {
        string solution;
        try {
            solution = _searcher->search(problem);
            Logger::getInstance()->log( message: "Found a solution using MatrixSolver...\n" + solution);
        } catch (const string &e) {
            Logger::getInstance()->log( message: "An error has occurred\n" + e);
            solution = e;
        } catch (...) {
            Logger::getInstance()->log( message: "An error has occurred...");
            solution = "An error has occurred.";
        }

        return solution + "\n";
    }
```

FACTORY

בתבנית factory נרצה לייצר אובייקטים שחולקים את אותו הinterface מבלי להכיר את המחלקות אשר מממשות את הinterface.

בתוכנית שלנו יש הרבה מאוד סוגי solverים. כיוון שאנחנו לא רוצים שהclient יהיה אחראי על יצירת האובייקטים, יצרנו solverFactory.
בזמן יצירת האובייקט menu ניצור את כל סוגי הsolverים ונשמור אותם במפה, כדי שלא נצטרך ליצור כל פעם (באותה הריצה) את אותו הsolver.
בעזרת קובץ הקונפיגורציה נעבור על האפשרויות השונות ונקרא לsolverFactory. בהינתן מזהה של solver – הsolverFactory ידע את מי לייצר, ויחזיר את האובייקט לmenu.

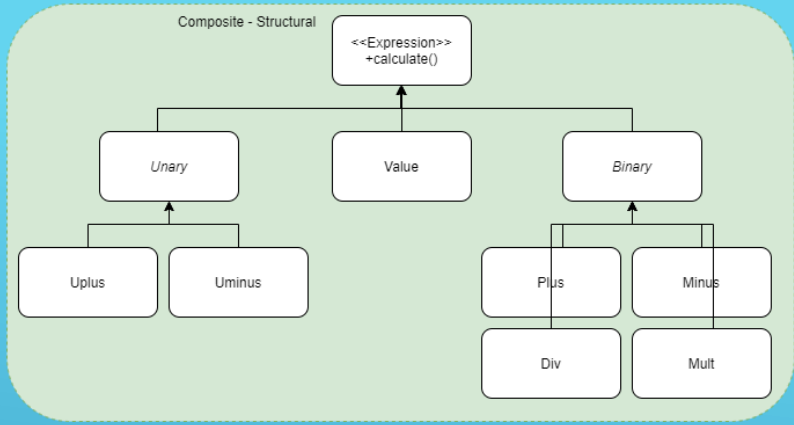
FACTORY דוגמא מהקוד

```
class Menu {
protected:
    ParserFactory parserFactory=ParserFactory();
    SolverFactory solverFactory = SolverFactory();
```

```
void Menu::initSolversMap(const vector<pair<string, vector<string>>> &s) {
    for (auto &pair : s) {
        string solver = pair.first;
        vector<string> searchers = pair.second;
        _solversMap[solver] = searchers;
        for (auto &searcher : searchers) {
            _searcherMap[solver + "_" + searcher] = _solverFactory.getSolver(solver, searcher);
        }
    }
}
```

```
class SolverFactory {
private:
    enum class SolverType {
        MATH_SOLVER, MATRIX_SOLVER
    };
    // map for searcher to int
    std::map<std::string, SolverType> searcherToInt =
            {
                    { x: "MatrixSolver",  y: SolverType::MATRIX_SOLVER},
                    { x: "MathSolver",    y: SolverType::MATH_SOLVER}

            };
public:

    Solver *getSolver(const string &solver, const string &searcher) {
        SearcherFactory searcherFactory;
        SolverType solverType = searcherToInt[solver];
        Solver *s = nullptr;
        switch (solverType) {
            case SolverType::MATH_SOLVER:
                s = new MathSolver(searcherFactory.getSearcher(searcher));
                break;
            case SolverType::MATRIX_SOLVER:
                s = new MatrixSolver(searcherFactory.getSearcher(searcher));
                break;

        }
        return s;
    }
```

COMPOSITE



בתבנית Composite אנו שומרים את הנתונים בצורה גנרית. יש היררכיית מחלקות מסוימת, כאשר השורש מכיל רשימה של אובייקטים מאותו הסוג.

בפרויקט השתמשנו בComposite כדי לממש את היררכית המחלקות של ביטויים מתמטיים.

כדי לפרש ביטוי מתמטי בצורת string העברנו את הביטוי לצורה של expressions, ויכלנו לחשב אותו באמצעות המתודה calculate().

למשל:

Expression * e = new Plus(new Value(3), new Value(4))

כאשר נבצע e.calculate() המתודות calculate יתבצעו באופן רקורסיבי ויחזירו את הפתרון.

דוגמא מהקוד COMPOSITE

```
class Expression {
public:
    virtual double calculate() = 0;

    virtual ~Expression() {}
};
```

```
class Value : public Expression {
    double num;
public:
    Value(const double number);

    double calculate() override;
};


class BinaryOperator : public Expression {
protected:
    Expression *right;
    Expression *left;
public:
    virtual Expression *getLeft();

    virtual Expression *getRight();

    virtual ~BinaryOperator();
};
```

```
class Plus : public BinaryOperator {
public:
    Plus(Expression *leftEx, Expression *rightEx);

    double calculate() override;

    ~Plus() override = default;

};
```

SINGLETON

תבנית Singleton מאפשרת לנו לשמור על אובייקט ייחודי. Singleton פותר לנו את בעיית האתחול הגלובלי.

בפרויקט השתמשנו בתבנית זו כדי לייצר Logger ולעקוב אחרי היסטוריית פעולות התוכנית בקובץ נפרד, כך שגם אם יש שגיאות נוכל לתעד אותן. בזכות זה שעשינו את הlogger שלנו Singleton מחלקות שונות יכולות לפנות אליו ולהכניס הודעות לקובץ הlog. נשים לב שהן תמיד פונות לאותו האובייקט, ולכן כותבות לאותו הקובץ.

בctor של הlogger אנו פותחים את קובץ הlog, כיוון שאנחנו משתמשים בתבנית Singleton, הקובץ ייפתח רק פעם אחת. נסגור את הקובץ ונמחק את האובייקט באמצעות פונ' סטטית resetInstance.

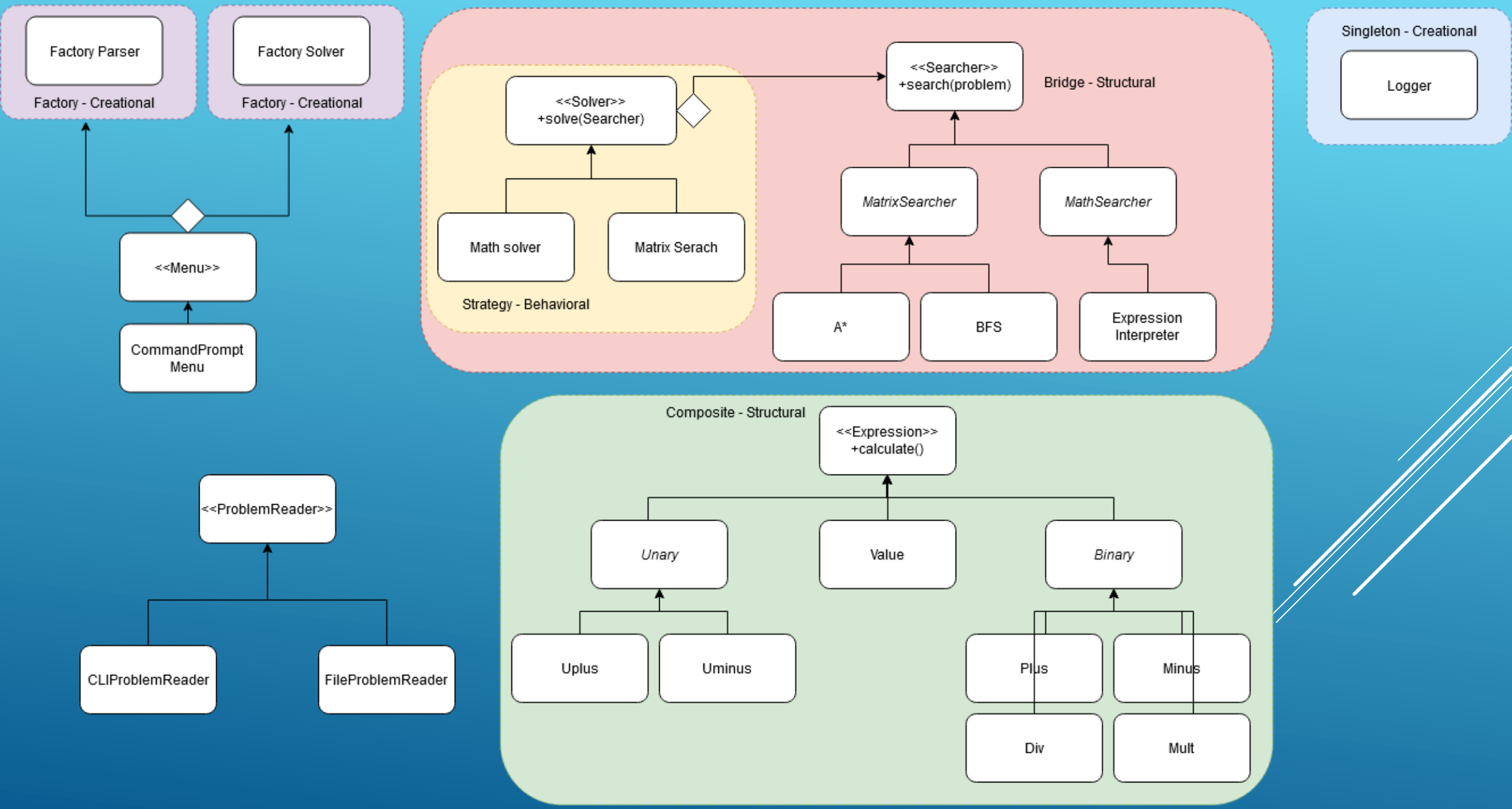
דוגמא מהקוד SINGLETON

```
class Logger {
private:
    static Logger *logger;
    ofstream file;

    Logger();

    ~Logger();
public:
    static Logger *getInstance();

    void log(const string &message);

    static void resetInstance();
};
```

```
Logger *Logger::logger = nullptr;

Logger::Logger() {
    file = ofstream( s: "logger.txt");
}

Logger::~Logger() {
    if (file.is_open()) {
        file.close();
    }
}

Logger *Logger::getInstance() {
    if (!logger) {
        logger = new Logger();
    }

    return logger;
}

void Logger::log(const string &message) {
    file << message + '\n';
}


void Logger::resetInstance() {
    delete logger;
    logger = nullptr;
}
```

```
bool CommandPromptMenu::solveProblem() {
    Logger::getInstance()->log( message: "Choosing problem at command prompt menu...");
```

```
string Astar::search(const string &matrixMazeStr) {
    Logger::getInstance()->log( message: "Starting to solve the problem using A*...\n" + matrixMazeStr);
```


Factory Parser
Factory - Creational
Factory Solver
Factory - Creational
<<Menu>>
CommandPrompt Menu
<<Solver>>
+solve(Searcher)
Math solver
Matrix Serach
Strategy - Behavioral
<<Searcher>>
+search(problem)
Bridge - Structural
MatrixSearcher
MathSearcher
A*
BFS
Expression Interpreter
Singleton - Creational
Logger
<<ProblemReader>>
CLIProblemReader
FileProblemReader
Composite - Structural
<<Expression>>
+calculate()
Unary
Value
Binary
Uplus
Uminus
Plus
Minus
Div
Mult