Model-View-Controller
Model Widok Kontroler

https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff649643(v=pandp.10)?redirectedfrom=MSDN

MVC

Wzorzec **Model–View–Controller (MVC)** dzieli aplikację na trzy współpracujące części:

- warstwę danych (**model**),
- warstwę prezentacji (**widok**),
- mechanizm reagowania na działania użytkownika (**kontroler**).

Model

- Model w tym to część odpowiedzialna za przechowywanie danych aplikacji oraz reguły rządzące ich zmianą.
- Model wie, *co* aplikacja przechowuje i jak powinny wyglądać operacje na tych danych.
- Model udostępnia też swój aktualny stan innym częściom systemu — najczęściej widokowi — oraz pozwala kontrolerowi modyfikować ten stan w uporządkowany sposób.

Widok

- Widok odpowiada za przedstawienie danych użytkownikowi.
- Widok pobiera informacje z modelu i prezentuje je w określonej formie — jako interfejs graficzny, tekst, tabelę, wykres czy dowolną inną wizualizację.
- Widok nie zmienia danych i nie decyduje o logice działania aplikacji; jego zadaniem jest jedynie pokazać aktualny stan modelu w czytelny i spójny sposób.

Kontroler

- Kontroler jest częścią odpowiedzialną za reagowanie na działania użytkownika.
- Kontroler odbiera zdarzenia takie jak kliknięcia, wpisywanie tekstu czy wybór opcji i na ich podstawie podejmuje decyzje, co powinno się wydarzyć w aplikacji.
- Kontroler najczęściej przekazuje polecenia modelowi, aby zmienić jego stan, albo instruuje widok, by zaktualizował sposób prezentacji danych.

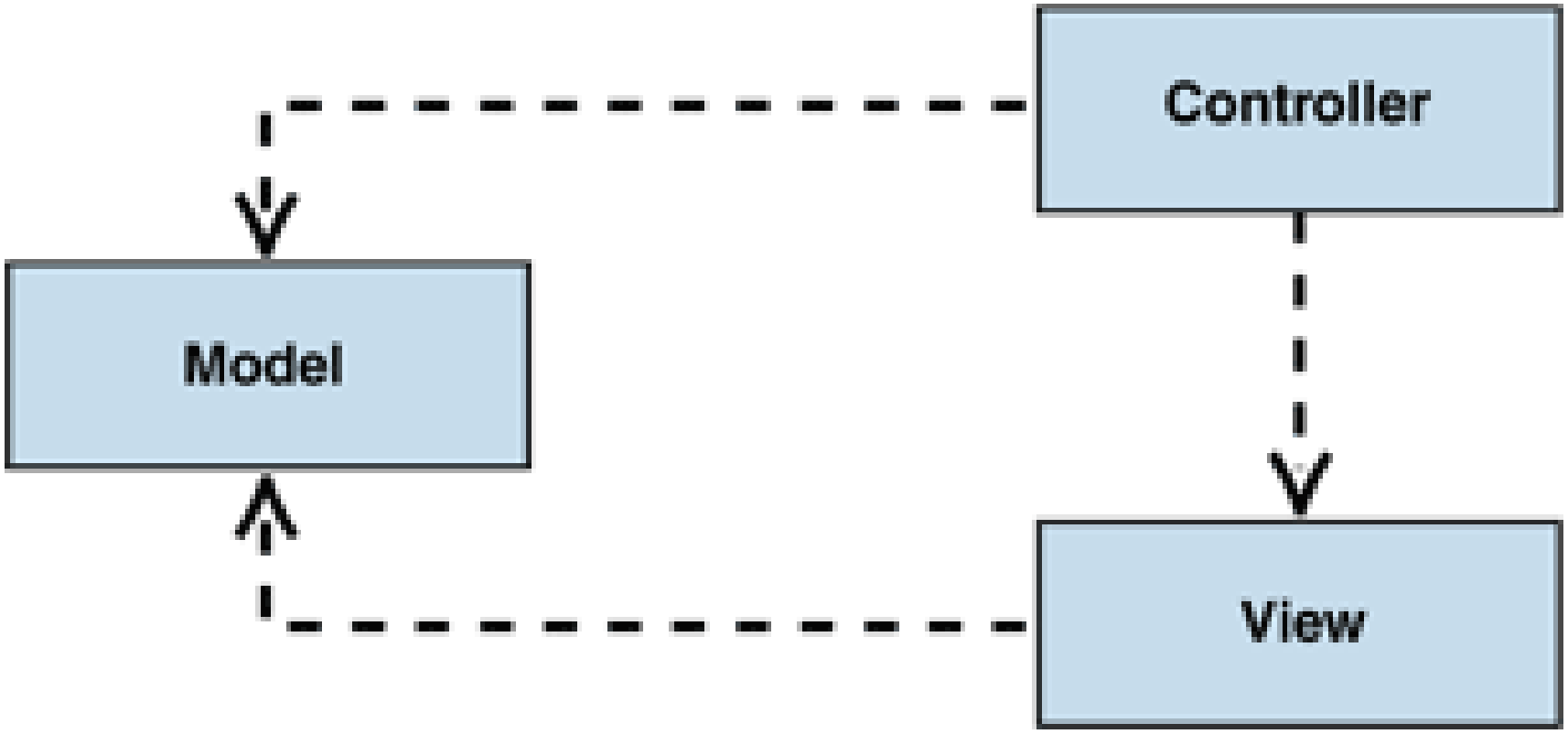
Podstawowa zasada (z małym wyjątkiem ☺)

Model **nie zna** ani widoków, ani kontrolerów.
Nie powinien ich importować, odpytywać ani wywoływać. Dzięki temu:

- pozostaje niezależny od sposobu prezentacji,
- można go testować i rozwijać bez zmian w UI,
- wiele widoków może korzystać z tego samego modelu,
- logika danych nie zależy od interakcji użytkownika.

Widoki i kontrolery znają model, ale zależność działa wyłącznie **w tę stronę**.

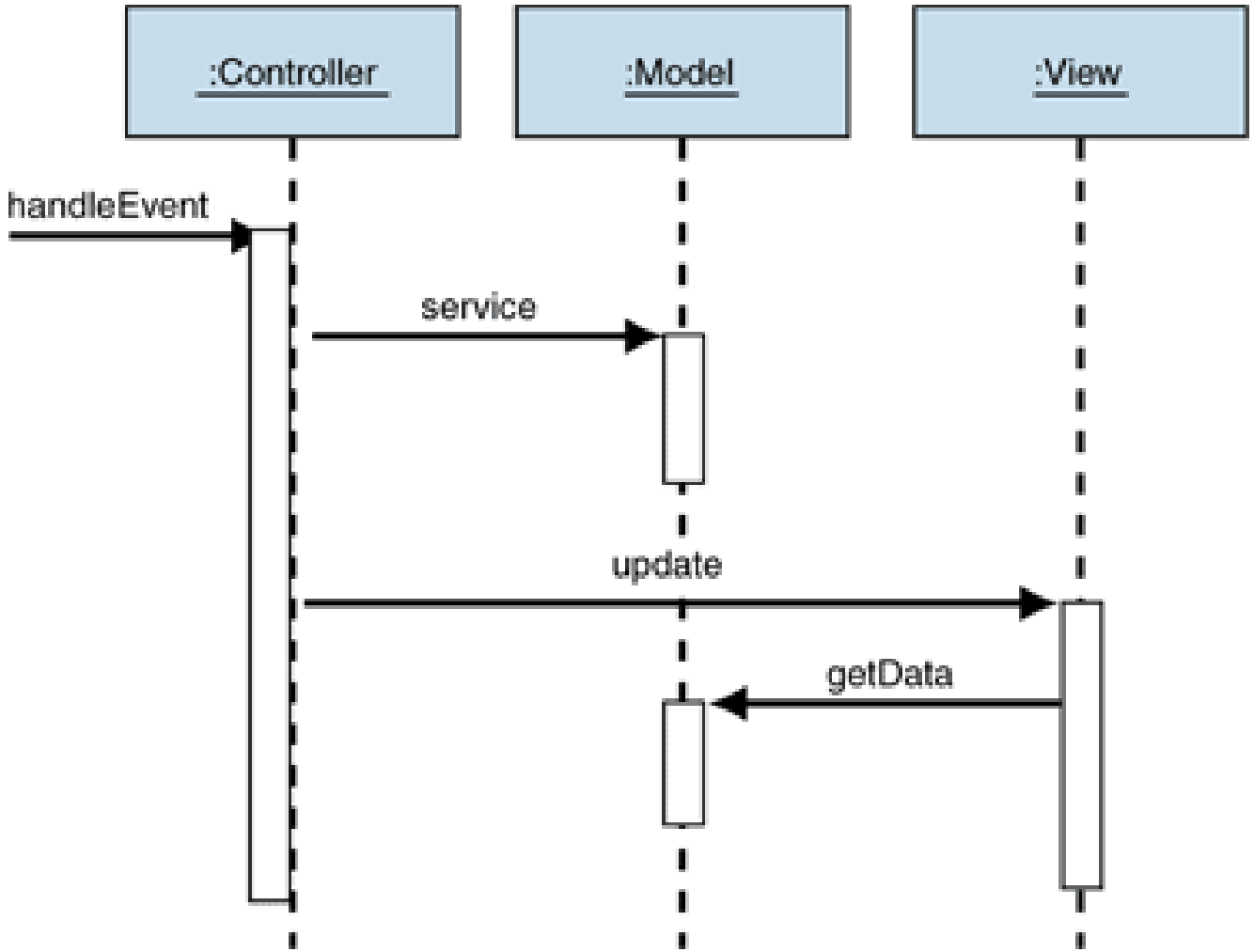
MVC – struktura dla modelu pasywnego



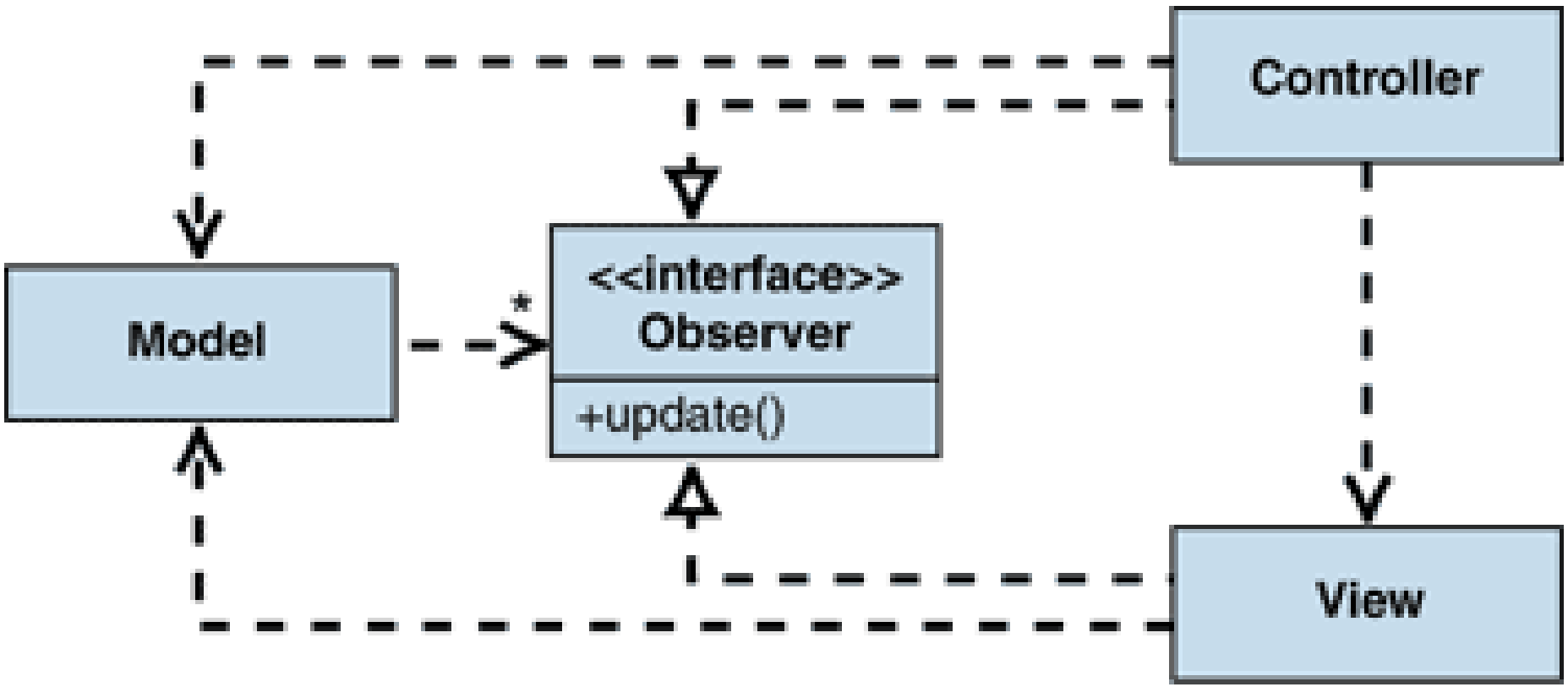
Model pasywny vs aktywny

- Model pasywny nie wie nic o widokach ani kontrolerach. Zmiany w danych są inicjowane przez kontroler, a widoki odpytywane są o stan modelu w odpowiedzi na akcje użytkownika.
- Model aktywny potrafi sam powiadamiać zainteresowane strony (widoki, czasem kontrolery) o zmianach w swoim stanie. W tym celu używa zwykle mechanizmu obserwatora (ang. observer pattern). Widoki lub kontrolery rejestrują się jako „słuchacze” modelu, a model wysyła im powiadomienia, gdy jego dane się zmienią.

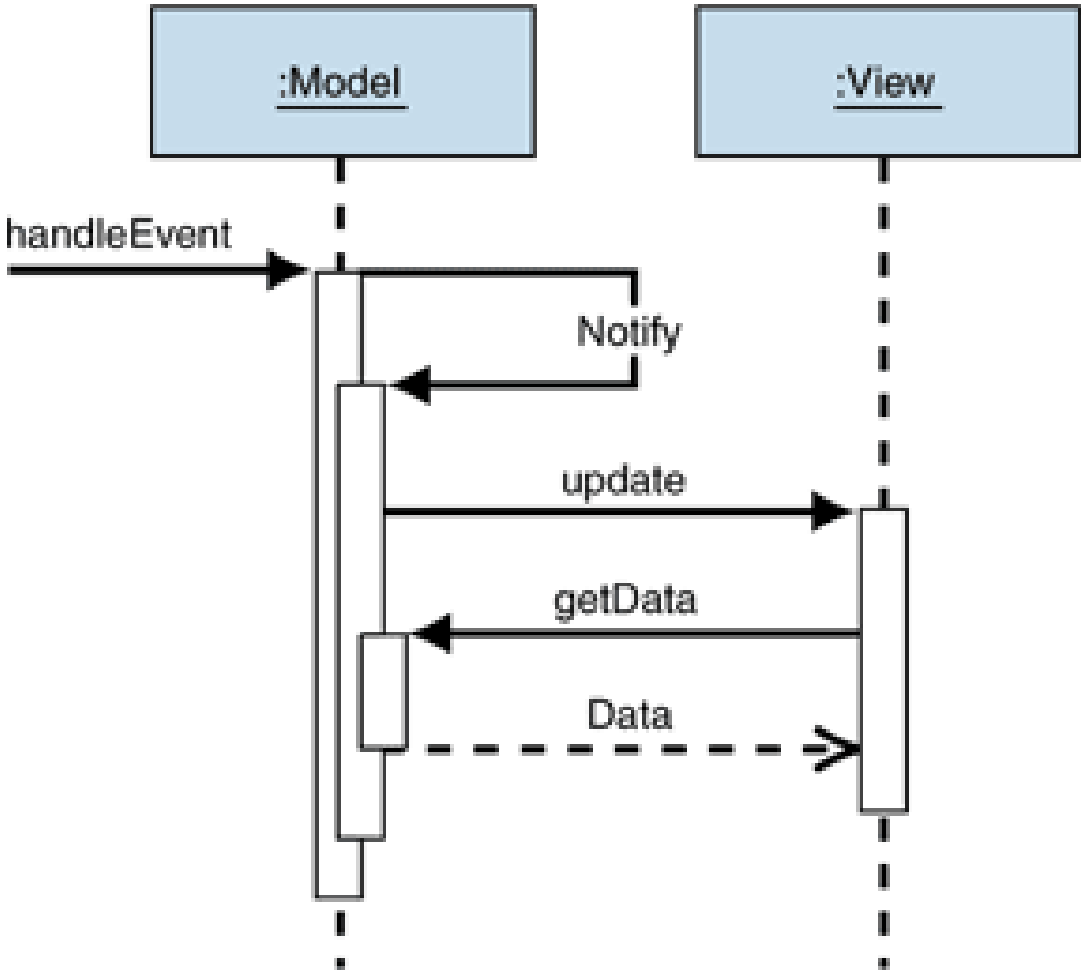
Behavior of the passive model



Using Observer to decouple the model from the view in the active model



Behavior of the active model



Model pasywny vs aktywny – podsumowanie

- **Model** w MVC odpowiada za przechowywanie danych i reguły zarządzania nimi. To on wie, *co* aplikacja przechowuje i *jak* powinny wyglądać operacje na tych danych.
- W **modelu pasywnym** zmiany w danych są inicjowane przez kontroler, a widoki odpytują model o jego stan w odpowiedzi na działania użytkownika. Model nie zna ani widoków, ani kontrolera.
- W **modelu aktywnym** model sam może informować zainteresowane strony (np. widoki) o zmianach swojego stanu. W tym celu widoki lub kontrolery rejestrują się jako „słuchacze” modelu, a model powiadamia je o każdej zmianie, dzięki czemu prezentacja danych może aktualizować się automatycznie.

MVC – podsumowanie

- Model – trzyma dane i reguły działania. Nie wie nic o widoku ani o kontrolerze.
- Widok – pokazuje dane użytkownikowi, nic nie zmienia w modelu i nie podejmuje decyzji.
- Kontroler – reaguje na akcje użytkownika i decyduje, co zrobić z modelem lub widokiem.

Zadanie

Napisz prostą aplikację - gra kółko i krzyżyk w konsoli, człowiek gra z komputerem. Komputer wykonuje losowe ruchy (do poprawienia później). Aplikacja ma wykorzystywać architekturę MVC.

Wskazówki

- W najprostszej wersji aplikacja powinna mieć trzy klasy:
  - `TicTacToeModel`
  - `TicTacToeView`
  - `TicTacToeController`
- W bardziej zaawansowanej wersji warto zamiast jednej klasy `TicTacToeModel` użyć kilku klas (one w sumie tworzą model):
  - `TicTacToeBoard`
  - `TicTacToeAI`
  - `TicTacToeGame`
- W dalszym ciągu omówimy tę bardziej zaawansowaną wersję.

Funkcja main

```
int main() {
    TicTacToeGame game;
    TicTacToeView view(game);
    TicTacToeController controller(game, view);

    controller.runGame();
}
```

Widok

```
class TicTacToeView {
private:
    TicTacToeGame &game;

public:
    TicTacToeView(TicTacToeGame &g) : game(g) {}
    void displayBoard() const {
        // wyświetla planszę za pomocą game.getCell(i,j)
        // TODO
    }
    void displayWinner(char player) const {
        cout << "Gracz " << player << " wygrał!\n";
    }
    void displayDraw() const {
        cout << "Remis!\n";
    }
    void displayMessage(const string &msg) const {
        cout << msg << endl;
    }
};
```

Kontroler

```
class TicTacToeController {
private:
    TicTacToeGame &game;
    TicTacToeView &view;

public:
    TicTacToeController(TicTacToeGame &g, TicTacToeView &v) : game(g), view(v) {}
    void runGame();

private:
    void playerMove();
};
```

Kontroler – metoda playerMove

```
void TicTacToeController::playerMove() {
    int row, col;
    bool valid = false;
    while (!valid) {
        cout << "Twój ruch (X). Podaj wiersz i kolumnę (0-2): ";
        cin >> row >> col;
        valid = game.makePlayerMove(row, col);
        if (!valid)
            view.displayMessage("Nieprawidłowy ruch, spróbuj ponownie.");
    }
}
```

```
void TicTacToeController::runGame() {
    while (true) {
        view.displayBoard();
        if (game.getCurrentPlayer() == 'X') {
            playerMove();
        } else {
            view.displayMessage("Ruch komputera (O):");
            game.makeComputerMove();
        }
        GameState state = game.getGameState();
        if (state == GameState::X_Won) {
            view.displayBoard(); view.displayWinner('X'); break;
        } else if (state == GameState::O_Won) {
            view.displayBoard(); view.displayWinner('O'); break;
        } else if (state == GameState::Draw) {
            view.displayBoard(); view.displayDraw(); break;
        }
        game.switchPlayer();
    }
}
```

metoda runGame

Model – GameState

- Na model składać będzie się kilka klas.
- Zaczniemy od najproszej klasy

```
enum class GameState { InProgress, Draw, X_Won, O_Won };
```

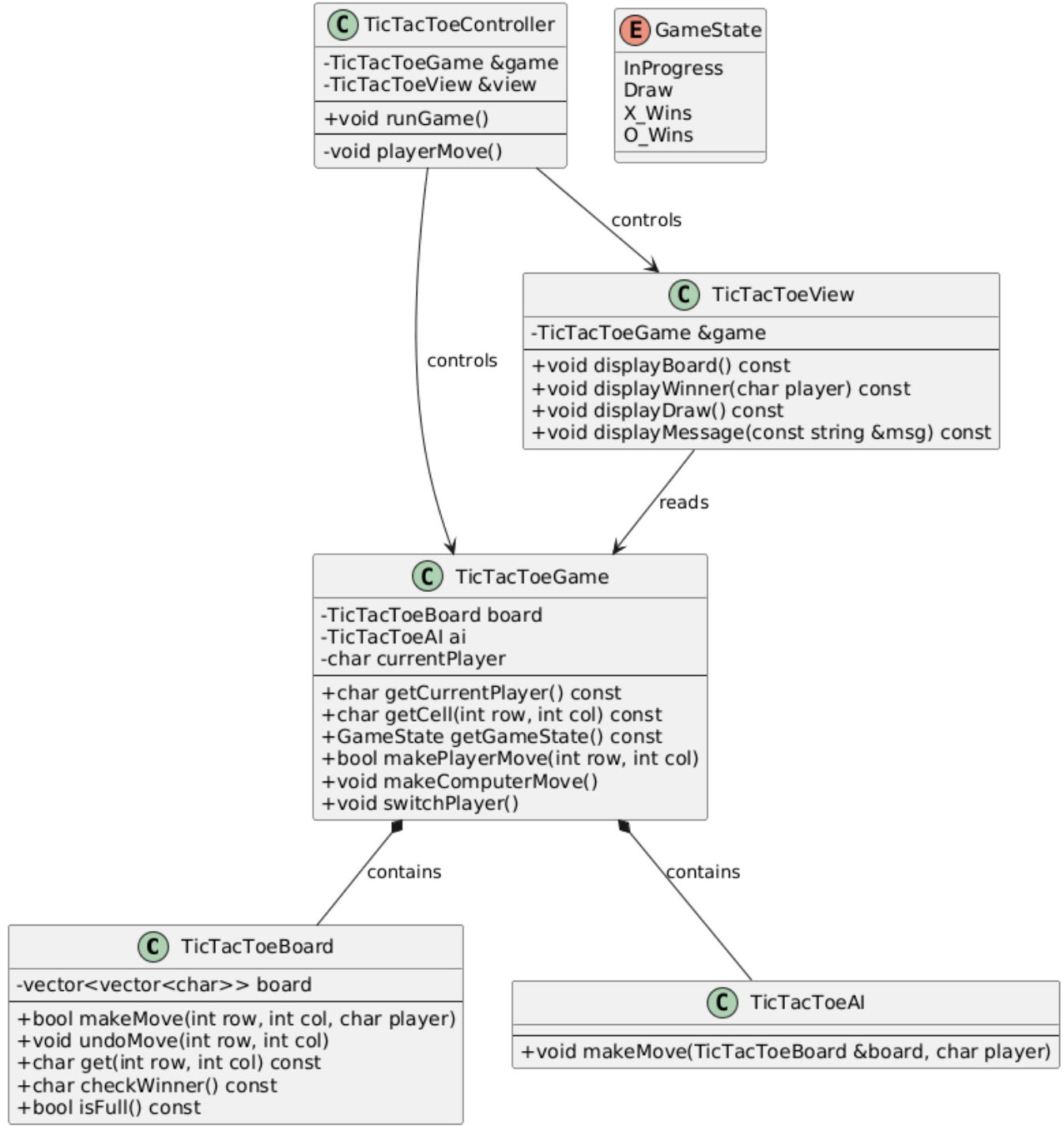
```
class TicTacToeGame {
private:
    TicTacToeBoard board;
    TicTacToeAI ai;
    char currentPlayer;
public:
    TicTacToeGame() : currentPlayer('X') {}
    char getCurrentPlayer() const { return currentPlayer; }
    char getCell(int row, int col) const { return board.get(row, col); }
    GameState getGameState() const;

    void switchPlayer() {
        currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
    }
    bool makePlayerMove(int row, int col) {
        return board.makeMove(row, col, currentPlayer);
    }
    void makeComputerMove() {
        ai.makeMove(board, currentPlayer);
    }
};
```

```
GameState TicTacToeGame::getGameState() const {
    char winner = board.checkWinner();
    if (winner == 'X') return GameState::X_Won;
    if (winner == 'O') return GameState::O_Won;
    if (board.isFull()) return GameState::Draw;
    return GameState::InProgress;
}
```

```
class TicTacToeBoard {
private:
    vector<vector<char>> board;
public:
    TicTacToeBoard() : board(3, vector<char>(3, ' ')) {}
    bool makeMove(int row, int col, char player) {
        if (row < 0 || row > 2 || col < 0 || col > 2) return false;
        if (board[row][col] != ' ') return false;
        board[row][col] = player;
        return true;
    }
    void undoMove(int row, int col) {
        board[row][col] = ' ';
    }
    char get(int row, int col) const {
        return board[row][col];
    }
    char checkWinner() const { // zwraca 'X' lub 'O' lub ' ' (brak zwycięzcy); TODO }
    bool isFull() const { // zwraca true jeśli na plansza jest pełna; TODO }
};
```

```cpp
class TicTacToeAI {
public:
    void makeMove(TicTacToeBoard &board, char player) {
        // Losowy ruch
        int row, col;
        do {
            row = rand() % 3;
            col = rand() % 3;
        } while (!board.makeMove(row, col, player));
    }
};
```


TicTacToeController
-TicTacToeGame &game
-TicTacToeView &view
+void runGame()
-void playerMove()
GameState
InProgress
Draw
X_Wins
O_Wins
controls
controls
TicTacToeView
-TicTacToeGame &game
+void displayBoard() const
+void displayWinner(char player) const
+void displayDraw() const
+void displayMessage(const string &msg) const
reads
TicTacToeGame
-TicTacToeBoard board
-TicTacToeAI ai
-char currentPlayer
+char getCurrentPlayer() const
+char getCell(int row, int col) const
+GameState getGameState() const
+bool makePlayerMove(int row, int col)
+void makeComputerMove()
+void switchPlayer()
contains
contains
TicTacToeBoard
-vector<vector<char>> board
+bool makeMove(int row, int col, char player)
+void undoMove(int row, int col)
+char get(int row, int col) const
+char checkWinner() const
+bool isFull() const
TicTacToeAI
+void makeMove(TicTacToeBoard &board, char player)