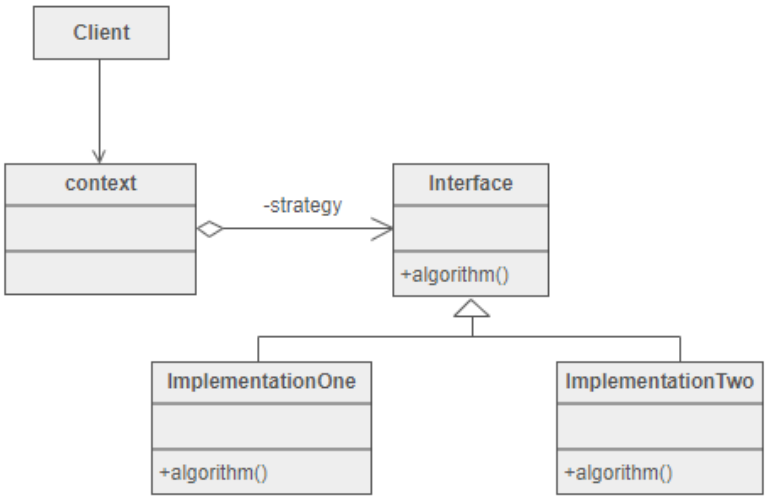
Projektni obrasci u aplikaciji Atomac

Pregled obrazaca:

- Strategy
- Decorator

Strategy:

Strategy obrazac spada u grupu obrazaca ponašanja. Njegova namena je da definiše familiju algoritama, enkapsulirajući svaki i čini ih međusobno zamenjivim. On omugućava jednostavnu promenu algoritma u vreme izvršenja. Dugo ime za Strategy obrazac je Policy.



Context – sadrži reference na interfejs koji može da ispolji promenljivo ponašanje

Interface – poseduje algoritam koji moze biti različito interpretiran

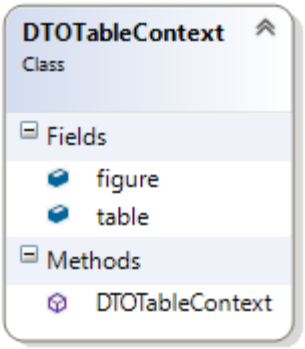
ImplementationOne/Two – definišu različita ponašanja algoritma koji zadovoljava neki problem

Client – poseduje kontekst trenutno izabranog ponašanja

Primena:

Različiti izgledi table, u zavisnotsti od konkretno izabrane strategije za iscrtavanje:

1. Boja na tabli
2. Izgleda figura



DTOTableContext – sadrži propertije koji definišu izgled table(boje) i figure. Pogodno za proširenje jer mogu da se dodaju novi dekoracioni elementi.

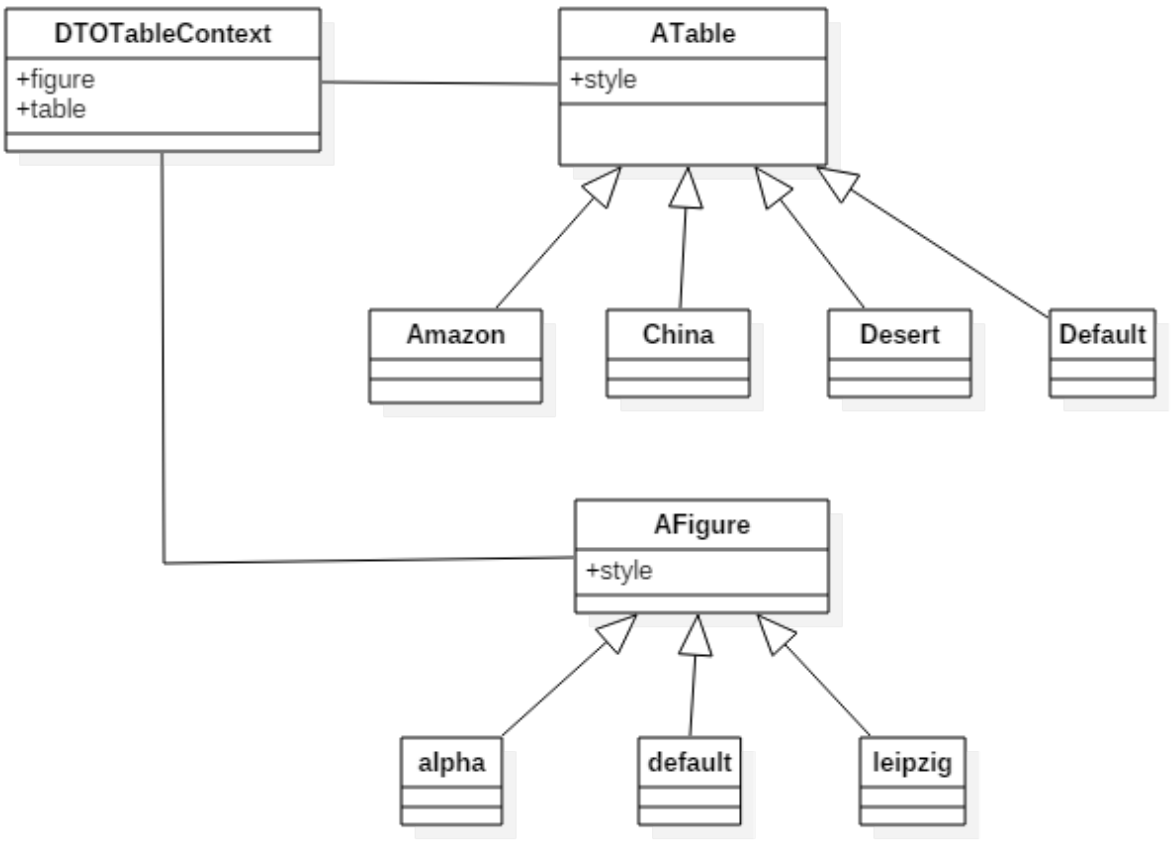
Objekat ove klase prenosi se na klijenta, gde se čuva i prilikom izmena menja izabranim vrednostima.

```
var CSS = {
  alpha: 'cbjs-alpha',
  black: cfg.tableContext.table+'-black',
  board: 'cbjs-board',
  chessboard: 'cbjs-chessboard',
  clearfix: 'cbjs-clearfix',
  highlight1: 'cbjs-highlight1',
  highlight2: 'cbjs-highlight2',
  notation: 'cbjs-notation',
  numeric: 'cbjs-numeric',
  piece: 'cbjs-piece',
  row: 'cbjs-row',
  sparePieces: 'cbjs-spare-pieces',
  sparePiecesRight: 'cbjs-spare-pieces-right',
  sparePiecesLeft: 'cbjs-spare-pieces-left',
  square: 'cbjs-square',
  white: cfg.tableContext.table+'-white'
};
```

Korišćenje propertija konteksta

```
var pom = JSON.parse('@Html.Raw(ViewBag.TableContext)');
sideBoard = new SideChessBoard('tableSide', '/Content/img/chesspieces/' + pom.figure + '/{piece}.png', [], [], 'black', pom);
```

Inicijalizacija konteksta na klijentu

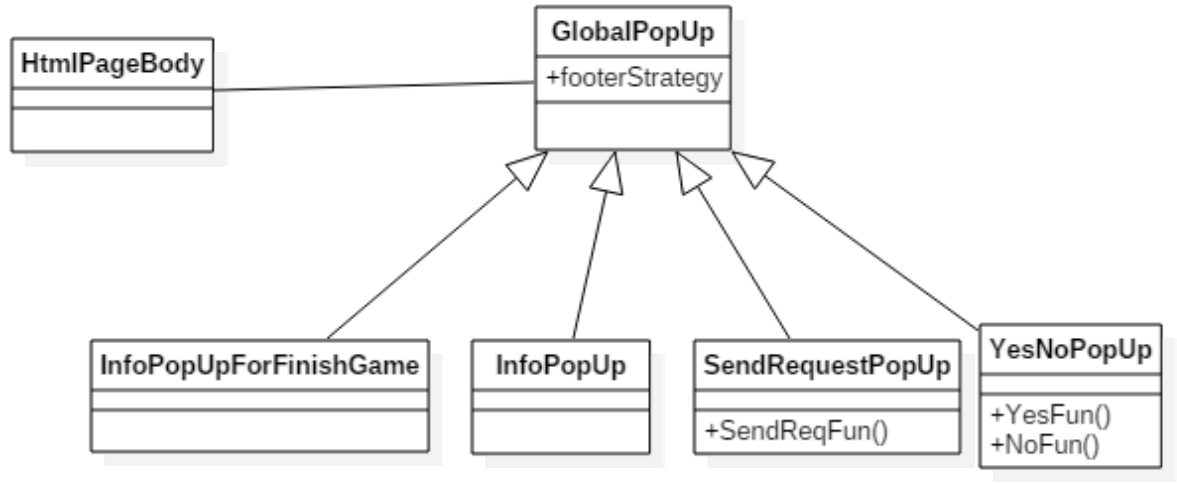


Podklase izvedene iz ATable I AFigure ne postoje jer bi doslo do eksplozije klasa. Podklase izvidene iz AFigure bi trebale da prestavljaju CSS klase (nalaze se u chessboard-0.3.0.css) za različite izglede crno-belih polja table. Primeri su klase:

- .Amazon-white i .Amazon-black
- .China-white i .China-black
- .Desert-white i .Desert-black

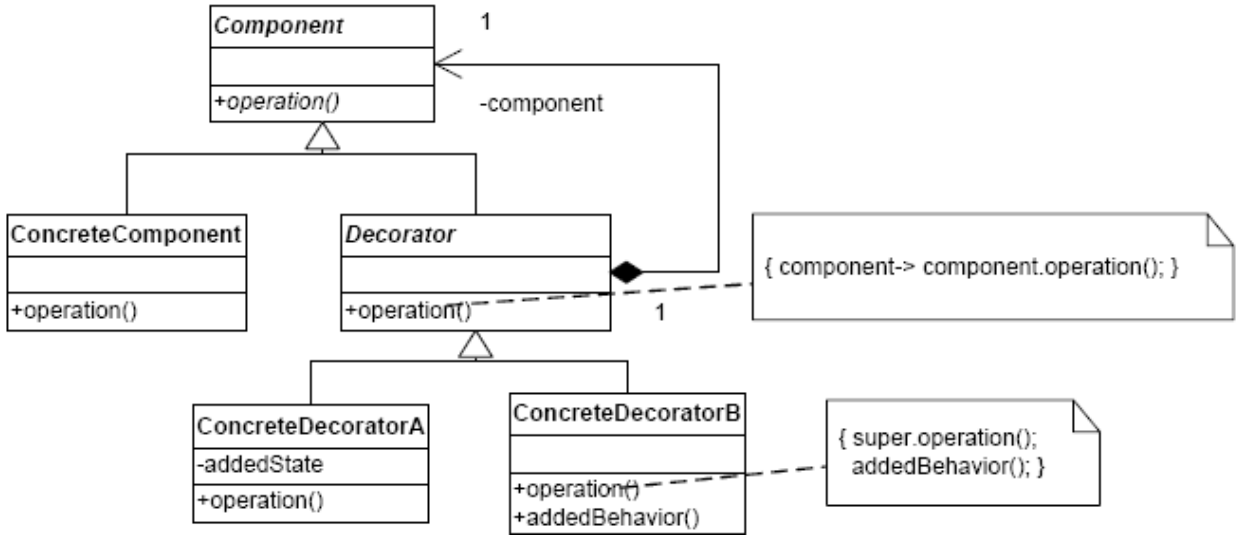
Primena 2:

U zavisnosti od potreba za razlicitim dugmicima i/ili delom u kojem se moze uneti tekst (naziv tima), prave se razliciti pop up modali. Osnovna ideja je bila da se dinamicki realizuje kreiranje modala. Kreiran je kroz funkciju **CreatePopUp**. Kasnije je uvidjena potreba za razlicitim modalima. Imajuci u vidu da deo modala ostaje nepromenjen, strategy obrazac se sam nametnuo za refaktorisanje. Kako je u pitanju javascript, strategija se bira kroz kreiranje kompozicije objekata. Sve ove “klase” (funkcije) su deo public-chat.js fajla.



Decorator:

Decorator obrazac spada u grupu strukturnih obrazaca. Njegova namena je dinamičko dodavanje novih osobina objektu. Time se dobija na fleksibilnosti, jer se u zavisnosti od prisutnih dekoratora može vršiti dodatna obrada, ili objekat može imati dodatne osobine, pri čemu zadržava isti interfejs kao i originalni objekat.



Component - interfejs koji implementiraju klase objekata koje dekorišemo, kao i dekorator

ConcreteComponent - konkretna klasa čiji objekti mogu biti dekorisani

Decorator - apstraktna klasa koja implementira isti interfejs kao i klase objekata kojima je dekorator namenjen

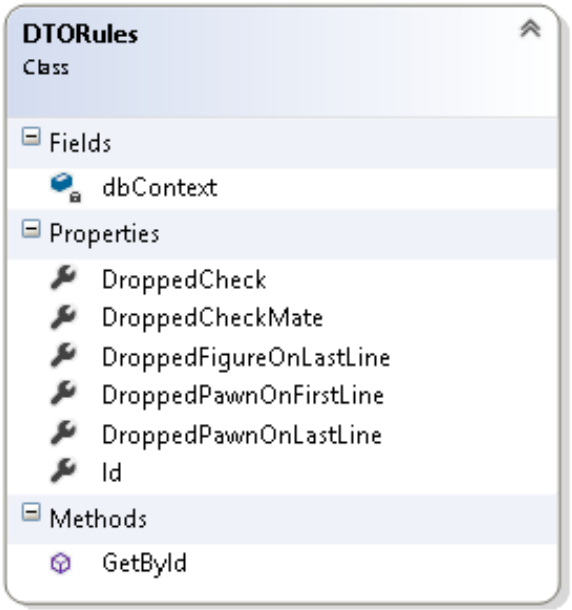
ConcreteDecoratorA/**B** - konkretni dekorator koji može imati dodatne podatke i/ili ponašanje

Primena:

Različita pravila vezana za dodatne figure, koja mogu biti aktivna u toku partije „Atomca" - dodatne figure su figure koje igrač dobija od svog saigrača kada saigrač pojede protivničku figuru, i može ih umetnuti na polje umesto regularnog poteza. Za sada postoje 5 pravila oblika DA/NE:

- Da li je moguće umetnuti figuru na polje tako da daje šah protivniku (**droppedCheck**)
- Da li je moguće umetnuti figuru na polje tako da daje mat protivniku (**droppedCheckMate**)
- Da li je moguće umetnuti pešaka na poslednju vrstu polja (**droppedPawnOnLastLine**)
- Da li je moguće umetnuti pešaka na prvu liniju polja (**droppedPawnOnFirstLine**)
- Da li je moguće umetnuti bilo koju figuru na poslednju liniju polja (**droppedFigureOnLastLine**)

Ova pravila se pamte u DTORules objektu koji sadrži po jedan boolean property za svako pravilo (imena property-ja su navedena u zagradi u listi iznad). Na osnovu podataka o izabranim pravilima za novu partiju kreira se objekat koji dekoriše igru (tj. stanje table) i služi za validaciju poteza u skladu sa odabranim pravilima.



U zavisnosti od odabranih pravila za partiju, zajedno sa igrom (i tablom) se kreira i moveValidator objekat, koji formira lančanu listu metoda koje po odigravanju poteza ispituju status igre, i ukoliko se odigrani potez ne slaže sa izabranim pravilima vraćaju staro stanje igre, time ne dozvoljavajući igraču da odigra sporni potez. Formira se lančana lista metoda od kojih svaka vrši svoj deo obrade (provere stanja) i vraća *false* ukoliko je stanje nedozvoljeno, ili poziva sledeću metodu u validatoru. Ukoliko su sva pravila zadovoljena, poslednja metoda će vratiti *true* i potez može biti odigran.

Ovakva realizacija praktično simulira dekorator, zbog velikih ograničenja JavaScripta u pogledu tipova podataka i njihovih hijerarhija (nemogućnost definisanja interfejsa/apstraktne klase). Pošto pravi dekorator u izvršenju ima formu lančane liste (redom se pozivaju metode svih dekoratora), smatramo da je ovo adekvatna zamena, tj. ponašanje je identično ponašanju pravog dekoratora.

Sve ovo je implementirano u fajlu *main-chess-board.js*, i prikazano je na slikama ispod:

Metoda koja u zavisnosti od odabranih pravila kreira validator.

```
let moveValidator = createValidator();

function createValidator() {
    let rules = [];
    let ruleInd = 0;

    rules[ruleInd++] = new EmptyRule(null);

    if (ruleSet.droppedCheck === false) {
        rules[ruleInd++] = new DroppedCheckRule(rules[ruleInd - 1]);
    }

    if (ruleSet.droppedCheckMate === false) {
        rules[ruleInd++] = new DroppedCheckMateRule(rules[ruleInd - 1]);
    }

    if (ruleSet.droppedPawnOnFirstLine === false) {
        rules[ruleInd++] = new DroppedPawnOnFirstLineRule(rules[ruleInd - 1]);
    }

    if (ruleSet.droppedPawnOnLastLine === false) {
        rules[ruleInd++] = new DroppedPawnOnLastLineRule(rules[ruleInd - 1]);
    }

    if (ruleSet.droppedFigureOnLastLine === false) {
        rules[ruleInd++] = new DroppedFigureOnLastLineRule(rules[ruleInd - 1]);
```

```
        function DroppedPawnOnLastLineRule(next) {
            return new Rule(() => {
                let moveHistory = game.history({ verbose: true });
                let lastMove = moveHistory[moveHistory.length - 1];
                let lastMoveTo = lastMove.to[1];

                if (lastMove.piece[0] === 'p'
                    && (lastMove.color === 'w' && lastMoveTo === '8'
                        || lastMove.color === 'b' && lastMoveTo === '1')) {
                    game.undo();
                    return false;
                } else {
                    return true;
                }
            }, next);
        }

        function DroppedFigureOnLastLineRule(next) {
            return new Rule(() => {
                let moveHistory = game.history({ verbose: true });
                let lastMove = moveHistory[moveHistory.length - 1];
                let lastMoveTo = lastMove.to[1];

                if (lastMove.piece[0] !== 'p'
                    && (lastMove.color === 'w' && lastMoveTo === '8'
                        || lastMove.color === 'b' && lastMoveTo === '1')) {
                    game.undo();
                    return false;
                } else {
                    return true;
                }
            }, next);
        }
```

*Klase pravila **droppedPawnOnLastLine** i **droppedFigureOnLastLine**.*

```
function Rule(validator, next) {
    this.validate = () => {
        if (validator() === false) {
            return false;
        }

        if (this.next !== null && this.next !== undefined) {
            return this.next.validate();
        }
    };
    this.next = next;
};

function EmptyRule(next) {
    return new Rule(() => {
        return true;
    }, next);
}

function DroppedCheckRule(next) {
    return new Rule(() => {
        if (game.in_check()) {
            game.undo();
            return false;
        } else {
            return true;
        }
    }, next);
}
```

*Apstraktna klasa za predstavljanje pravila **Rule**, i jedan deo konkretnih pravila. Prazno pravilo je uvek prisutno u slučaju da igrači nisu odabrali nijedno ograničenje (sva pravila su true).*

```
function DroppedCheckMateRule(next) {
    return new Rule(() => {
        if (game.in_checkmate()) {
            game.undo();
            return false;
        } else {
            return true;
        }
    }, next);
}

function DroppedPawnOnFirstLineRule(next) {
    return new Rule(() => {
        let moveHistory = game.history({ verbose: true });
        let lastMove = moveHistory[moveHistory.length - 1];
        let lastMoveTo = lastMove.to[1];

        if (lastMove.piece[0] === 'p'
            && (lastMove.color === 'w' && lastMoveTo === '1'
                || lastMove.color === 'b' && lastMoveTo === '8')) {
            game.undo();
            return false;
        } else {
            return true;
        }
    }, next);
}
```

*Klase pravila **droppedCheckMate** i **droppedPawnOnFirstLine**.*