AARHUS UNIVERSITET

Software Engineering and Architecture

Comments and Hints on

Mandatory Iteration 3 / Strategies

Beta/Gamma/Delta

- This is the **Strategy week (or 3-1-2 week)**

– BetaStone	Maintain BOTH variants Alpha+Beta
– GammaStone	Maintain ALL variants
– DeltaStone	Maintain ALL variants

- That is
  - The aspects that vary
    - Mana production
    - Winner determination
    - Hero Power
    - Deck building
  - … Must be ③①② processed ⇨ Several Strategy Patterns
    - Or rather examples of Compositional Design…

Refactor First / Add Features Then

- For all
  - Introduce the XStrategy *first* by making AlphaStone’s test cases pass *first*
  - Only *then* do you introduce the specific new tests to drive the new XStone specific behavior into existence

Unit / Integration Tests

- Some strategies can be tested in isolation – do it!
  - Just like ‘RateStrategies’ could be tested as a unit, so can *some of the HotStone strategies.*
  - It is often MUCH simpler!
- Example: DeltaStone is a variant with other cards (The ‘DishDeck’)
  - Encapsulate what varies (building a deck)
  - Program to an interface (define nice interface for that)
  - Favor object composition (game asks strategy to build a deck)

Unit/Integration Testing

- So there will be ‘an implementation of “build deck strategy” that create a deck’
  - Typical return some kind of List<Card> or array or …

- The Unit Testing Point
  - Does TDD/testing of that implementation rely on Game?
    - Most likely not! *It is just returning something*
  - Then test it in isolation!
    - *Given* strategy to create a dishdeck
    - *When* I create the strategy
    - *Then* it contains …

Unit/Integration Testing

- Not all aspects can be tested disjointly from Game
  - [At least not now, we will be able to do so, at a later point…]
- Example
  - GammaStone Hero Power – need to modify game's state
  - BetaStone Winner? Maybe – or maybe not?
    - Discuss in the SWEA Kata

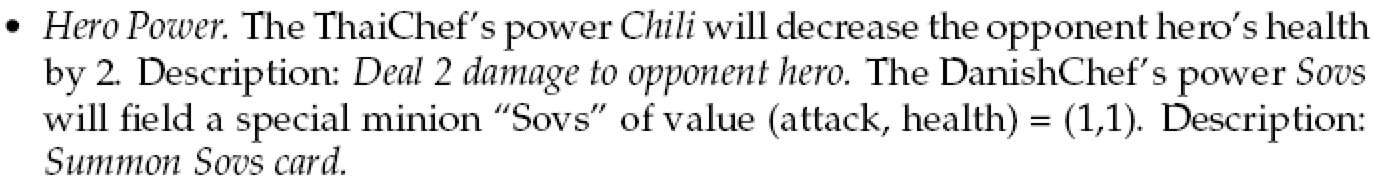
Strategies That Inspect and Modify State

Determine Winner/Stop Game

- Determining the winner is one thing…
- *But winning the game means no further method calls are allowed, right?*
  - Should we guard all calls to see if game is still ongoing?
- The reasoning is sound, but…
- **The UI will handle it!**
  - Down the road …
- So: KISS: *Keep it simple, stupid*

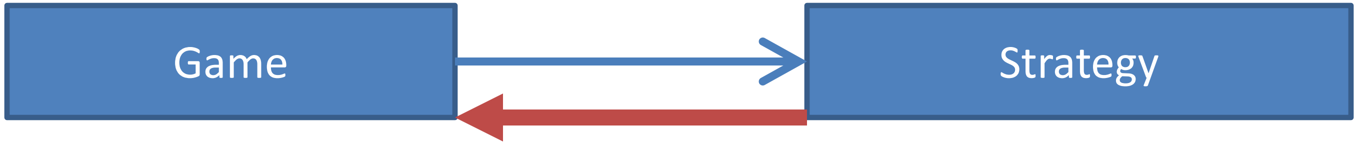
GammaStone HeroPower

- The specification:



- *Hero Power.* The ThaiChef's power *Chili* will decrease the opponent hero's health by 2. Description: *Deal 2 damage to opponent hero.* The DanishChef's power *Sovs* will field a special minion "Sovs" of value (attack, health) = (1,1). Description: *Summon Sovs card.*

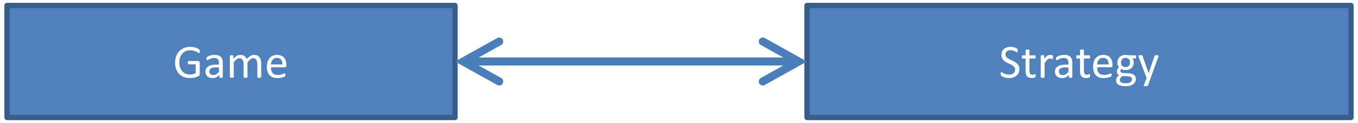
- That is
  - Mutation of game's state from another object than game!
    - Game will call *strategy* to tell that user wants to use hero power
    - *But…* The strategy then needs to *modify state of the calling game!*



AARHUS UNIVERSITET

The “Back pointer”

- As the strategy object must be able to mutate game, it must be *provided with a reference to game*
- We can handle by a mutual reference or “back pointer”



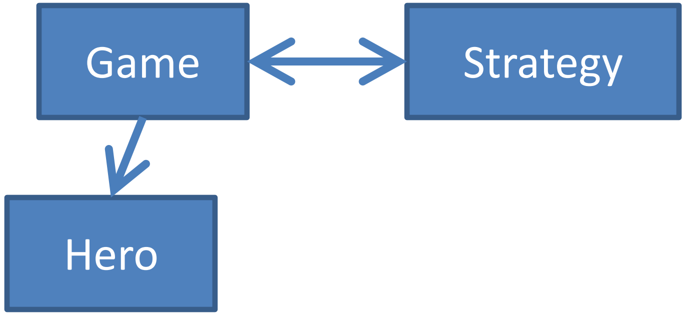
  - Ala calling from within the Game object (this):
    - myHeroPowerStrategy.useThePower(who, this); **or**
    - getHero(who).getHeroPowerStrategy().execute(this);
  - But – Game has no special mutator methods for, say, *decrease hero ‘who’s health by two*

Modifying Game’s State

- Ok, so Game has asked the strategy to ‘execute that hero power’ – what does that concrete strategy then do?
- Example: ThaiChef “Deal 2 damage to opp. hero”
- Either
  - A) Get the opponent hero (type: Hero), cast it to a (StandardHero), and call a mutator, ala
    - StandardHero hero = (StandardHero) game.getHero(who);
    - hero.changeHealth(-2);
  - B) Add a mutator method to game to *encapsulate* that, ala
    - game.changeHeroHealth(who, deltaValue);

Analysis

- Exercise: Which is better? Or are they equal?
  - A: “game.getHero(who).changeHealth(-2)”
  - B: “game.changeHeroHealth(who, -2)”



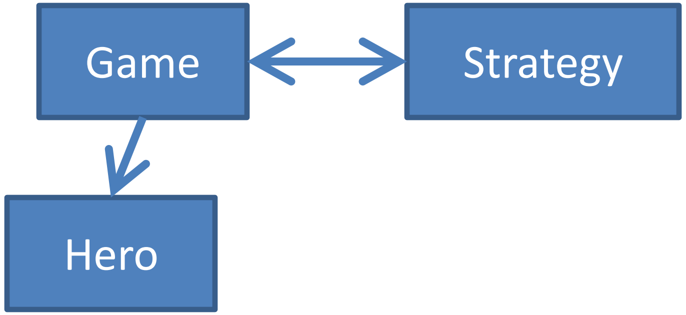
- Arguments? Pros and Cons?

- Remember previous courses’ discussion on *coupling and cohesion?*
  - Low coupling = “do not talk to strangers”
  - High cohesion = “I handle all related to me”

Analysis

- Low coupling = "do not talk to strangers"
- High cohesion = "I handle all related to me"

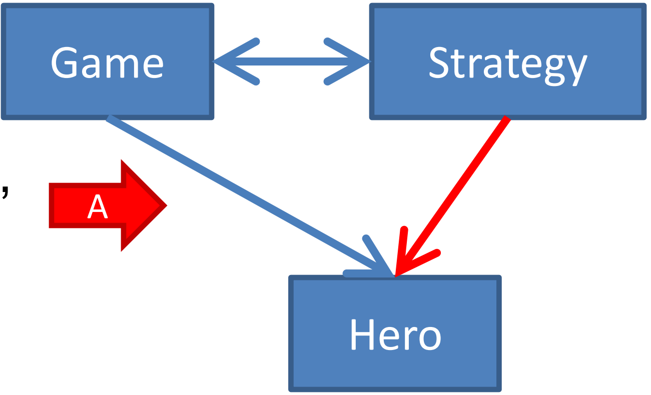
- Which is more favorable?
  - A: "game.getHero(who).changeHealth(-2)"
  - B: "game.changeHeroHealth(who, -2)"



Analysis

- Low coupling = "do not talk to strangers"
- High cohesion = "I handle all related to me"

- Which is more favorable?
  - A: "game.getHero(who).changeHealth(-2)"
  - B: "game.changeHeroHealth(who, -2)"



- *B is:*
  - *Strategy does not talk to Hero, only Game (no strangers)*
  - *Strategy does not change Game's state 'behind the scene', (game handles everything related to its state.)*

General Rule

- *Do* ***not*** *let any strategy…*
  - *Get hero/card and modify state on them (talking to strangers)*
  - *Get game’s internal data structures and modify state on them*
    - *Both talking to strangers and breaking encapsulation!*

- ***Instead Do let any strategy that needs to modify game state:***
  - ***Get a reference to game (implementation), and call (new) mutator methods that let game itself modify internal state and data structures!***

Hero Subclasses

Why not just subclass to handle GammaStone Hero ?

XHero

- IntProg taught about subclassing, so why not use it here?
  - A hero has a power, so why not add 'usePower(…)' method to Hero, and use subclassing
    - BabyHero::usePower(Game g) { // do nothing }
    - ThaiHero extends BabyHero
    - usePower(Game g) { [cast g to StandardGame, call mutators;] }
- **This is a correct solution to the functional requirement – and design-wise a sound one.**
- **But in SWEA we train a compositional approach! So to train that, you should avoid inheritance.**

Compositional Hero

- Thus, your options to train are *Strategy* based approach
  - Either a Strategy in the Game or in the Hero
- Ala
  - Game::usePower() { heroStrategy.execPower(who, this); }
  - Or hero has a strategy
  - Game::usePower() { getHero(who).getPowerStrategy().execPower(this);}
  - Or a pure ‘lambda function’ approach:
    - Game/hero has a Consumer<Game> functional interface/lambda expression

Compositional Hero

- Thus your options to train are *Strategy* based approach
  - Either in Game or in the Hero
- Ala
  - Game::usePower() { heroStrat
  - Or hero has a strategy
  - Game::usePower() { getHero(who).getPowerStrat
  - Or a pure ‘lambda function’ approach:
    - Game/hero has a Consumer<Game> functional interface/lambda expression

One will lead to a lot of ‘if’s and the other will not. Analyze and pick your favorite. No ‘really wrong’ solution though.

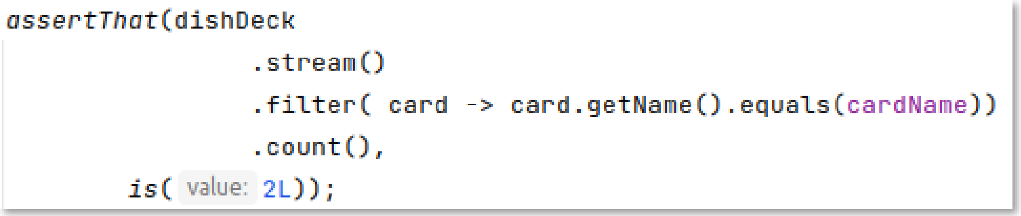
DeltaStone

- Strategy = algorithm that varies
- Here: Game’s algorithm to *build a deck* varies

- One note
  - The deck is shuffled – how to TDD that???
    - i.e. You cannot assertThat card 17 is Poke Bowl because it may be Green Salad ☹
  - (We will come back to a solution to this later, but we *can do* something now)
  - Any ideas?
    - Hint – look at the specifications – we know *something*

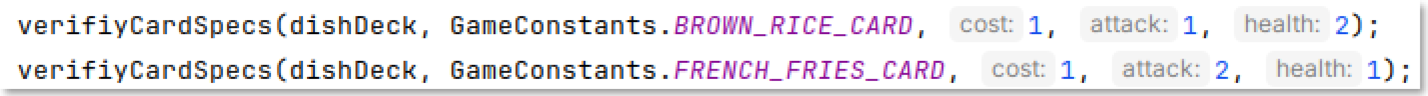
DishDeck

- How to TDD a deck which is shuffled?
- TDD the aspects that you know about
  - First card has mana 1 = …, is(1)
  - Second card has mana 1 or 2 = …, lessThanOrEqualTo(2)
- There are exactly two Noodle Soup Cards in deck



```
assertThat(dishDeck
                .stream()
                .filter( card -> card.getName().equals(cardName))
                .count(),
        is( value: 2L));
```

- Make a private test method to verify the exact cost, attack, health of a given card
  - And use it on each card type: Tomato Salad, Brown Rice, …



```
verifiyCardSpecs(dishDeck, GameConstants.BROWN_RICE_CARD,  cost: 1,  attack: 1,  health: 2);
verifiyCardSpecs(dishDeck, GameConstants.FRENCH_FRIES_CARD,  cost: 1,  attack: 2,  health: 1);
```

Randomness?

- How to test that the deck is really random?
  - Generate N decks, assertThat(“they are different”)
    - Ahem – with a probability higher than …
      - (We could encounter two random runs that generate same deck!)

- Evident test
  - *If you write 10 lines of test code to test 1 line of production code your bug will be in the test code* ☹
- How to shuffle a deck in Java?
  - 1 line: Collections.shuffle(myDeck);
  - It is TDD principle ”obvious implementation”