AARHUS UNIVERSITET

Software Engineering and Architecture

Reflections on Mandatories

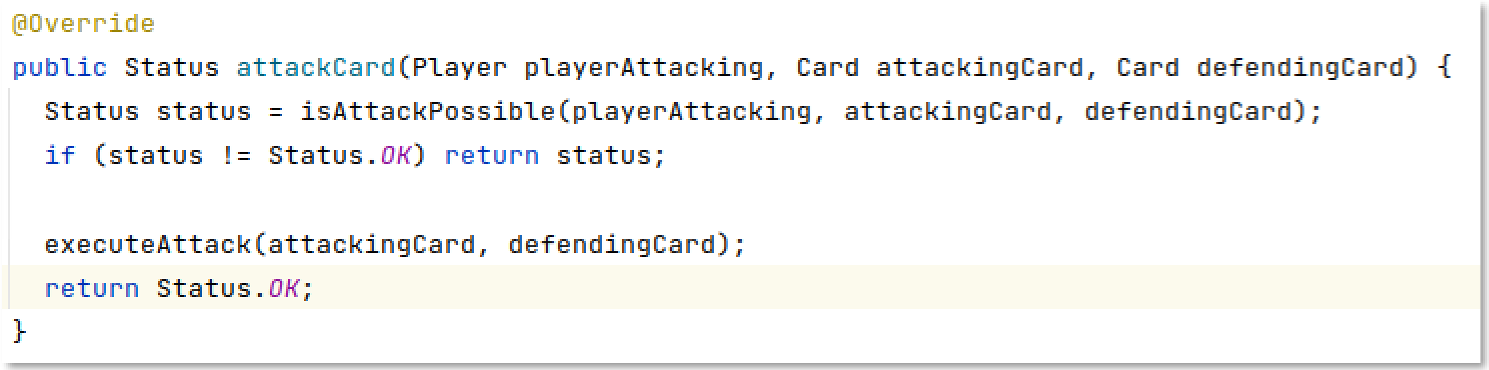
DOT & OLoA

- Two clean code principles often confuse
  - Do One Thing: “But it does multiple things…”
  - One Level of Abstraction: “Huh???”

- Do One Thing example
  - Game’s method attackCard()
    - “It does a lot, so it does not obey the ‘do one thing’ principle”
  - Yes, it does…
    - It does one thing: *it executes a card attack*
      - As seen from the perspective of the “Game”
  - [In my Clean Code slides, I am a bit ambiguous about that, sorry]

DOT

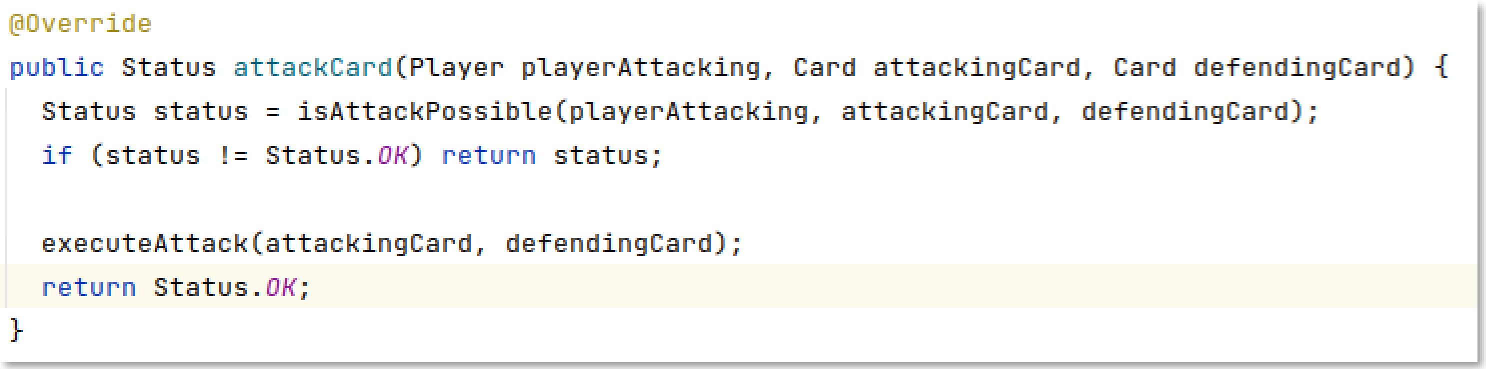
- So ‘Do One Thing’ depends on the perspective and has to be considered from the context
  - attackCard is a single function/operation from the ‘user of game’
  - But of course, internally (inside the method), it does quite a few things
  - **These ‘things’ can again be grouped into ‘units of doing one thing’**
    - Validate that an attack is possible; if so then do the attack



```java
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
  Status status = isAttackPossible(playerAttacking, attackingCard, defendingCard);
  if (status != Status.OK) return status;

  executeAttack(attackingCard, defendingCard);
  return Status.OK;
}
```

OLoA

- One Level of Abstraction
  - Tells us that these ‘next level things’ should also be grouped into ‘do one thing’ methods

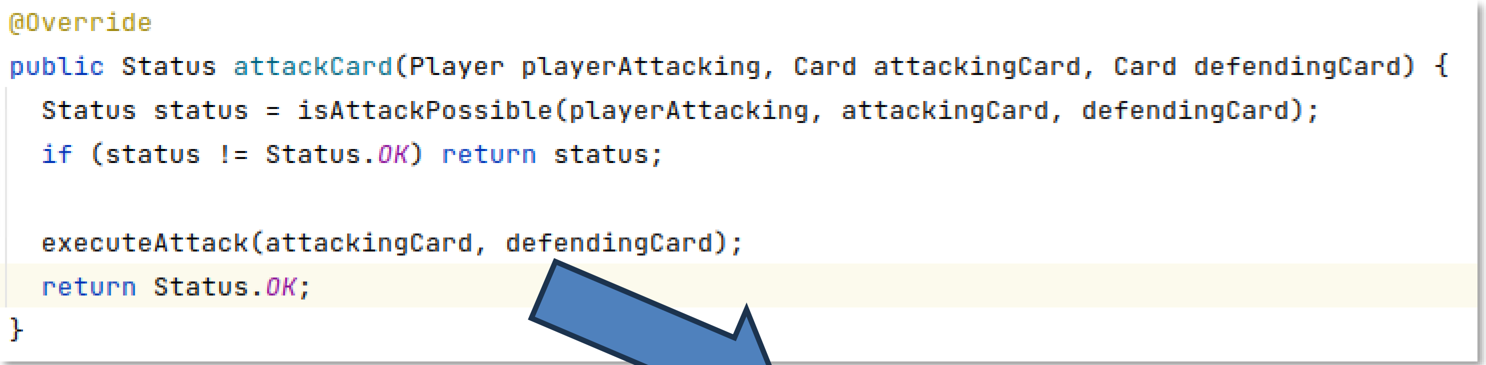


```java
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
  Status status = isAttackPossible(playerAttacking, attackingCard, defendingCard);
  if (status != Status.OK) return status;

  executeAttack(attackingCard, defendingCard);
  return Status.OK;
}
```

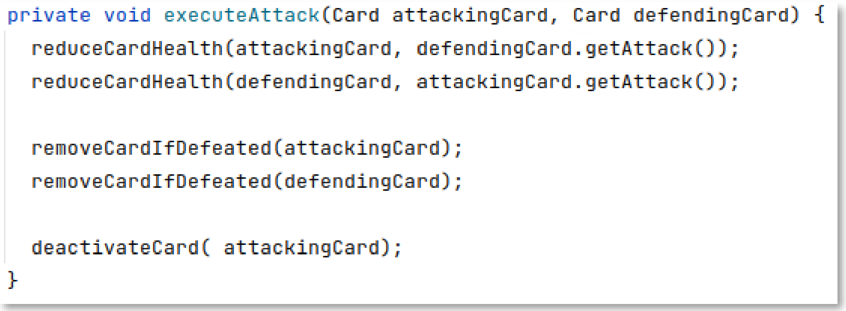
  - … and so on

OLoA

- As in…



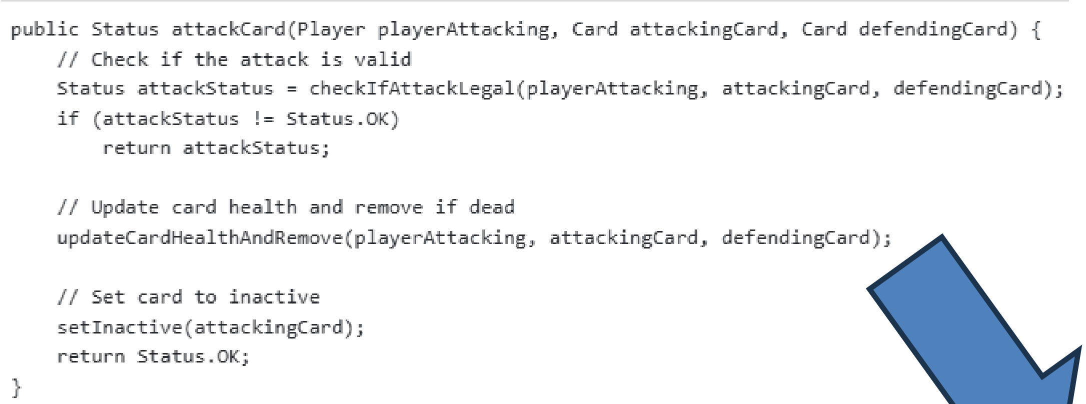
```java
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
  Status status = isAttackPossible(playerAttacking, attackingCard, defendingCard);
  if (status != Status.OK) return status;

  executeAttack(attackingCard, defendingCard);
  return Status.OK;
}
```



```java
private void executeAttack(Card attackingCard, Card defendingCard) {
  reduceCardHealth(attackingCard, defendingCard.getAttack());
  reduceCardHealth(defendingCard, attackingCard.getAttack());

  removeCardIfDefeated(attackingCard);
  removeCardIfDefeated(defendingCard);

  deactivateCard( attackingCard);
}
```

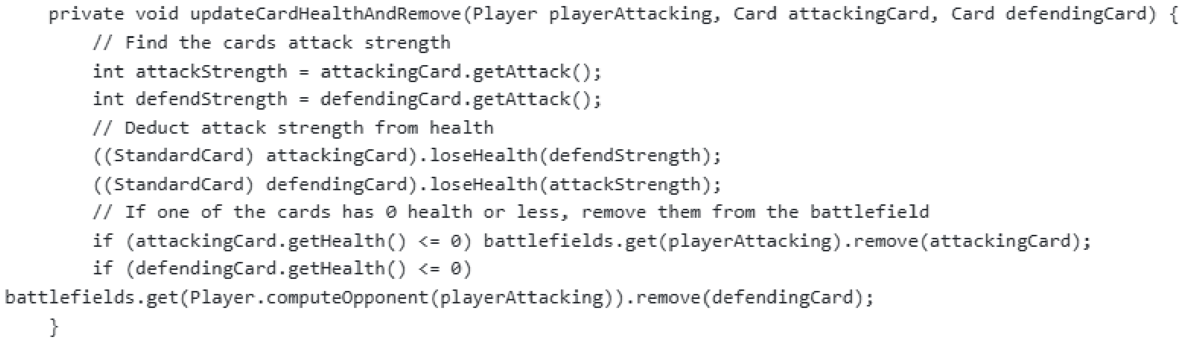
It does not stop at level 1

- TAs report multiple examples of ‘stopping at level 1’
  - Fine ‘next level of abstraction’



```
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    // Check if the attack is valid
    Status attackStatus = checkIfAttackLegal(playerAttacking, attackingCard, defendingCard);
    if (attackStatus != Status.OK)
        return attackStatus;

    // Update card health and remove if dead
    updateCardHealthAndRemove(playerAttacking, attackingCard, defendingCard);

    // Set card to inactive
    setInactive(attackingCard);
    return Status.OK;
}
```

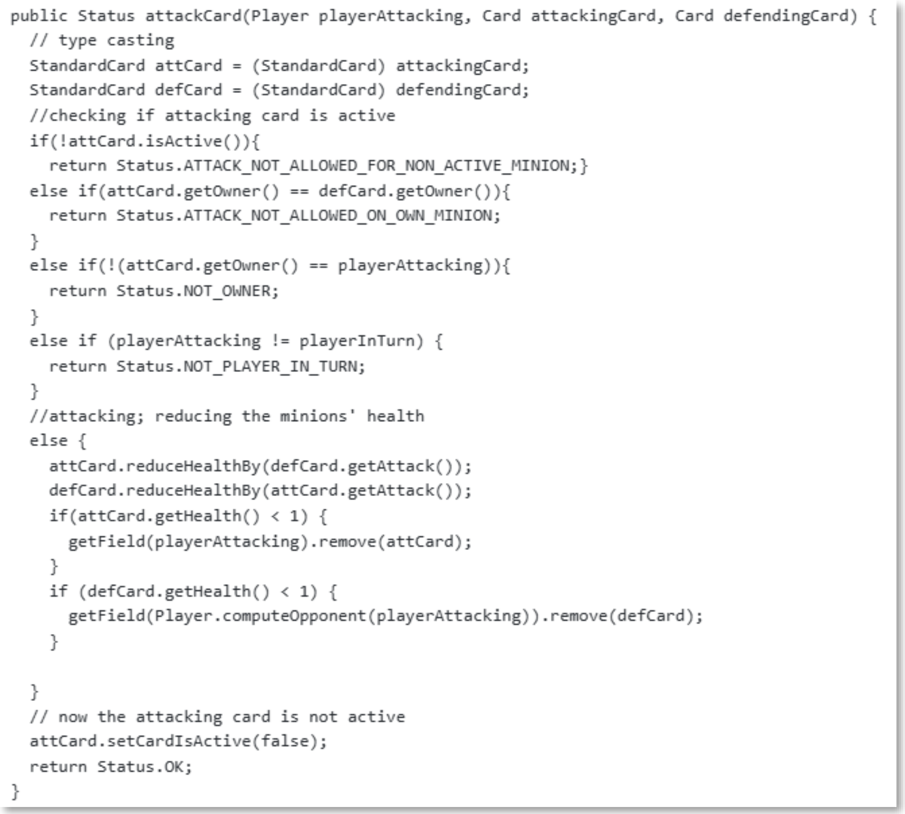
  - But next level is just ‘all of it’



```
    private void updateCardHealthAndRemove(Player playerAttacking, Card attackingCard, Card defendingCard) {
        // Find the cards attack strength
        int attackStrength = attackingCard.getAttack();
        int defendStrength = defendingCard.getAttack();
        // Deduct attack strength from health
        ((StandardCard) attackingCard).loseHealth(defendStrength);
        ((StandardCard) defendingCard).loseHealth(attackStrength);
        // If one of the cards has 0 health or less, remove them from the battlefield
        if (attackingCard.getHealth() <= 0) battlefields.get(playerAttacking).remove(attackingCard);
        if (defendingCard.getHealth() <= 0)
battlefields.get(Player.computeOpponent(playerAttacking)).remove(defendingCard);
    }
```

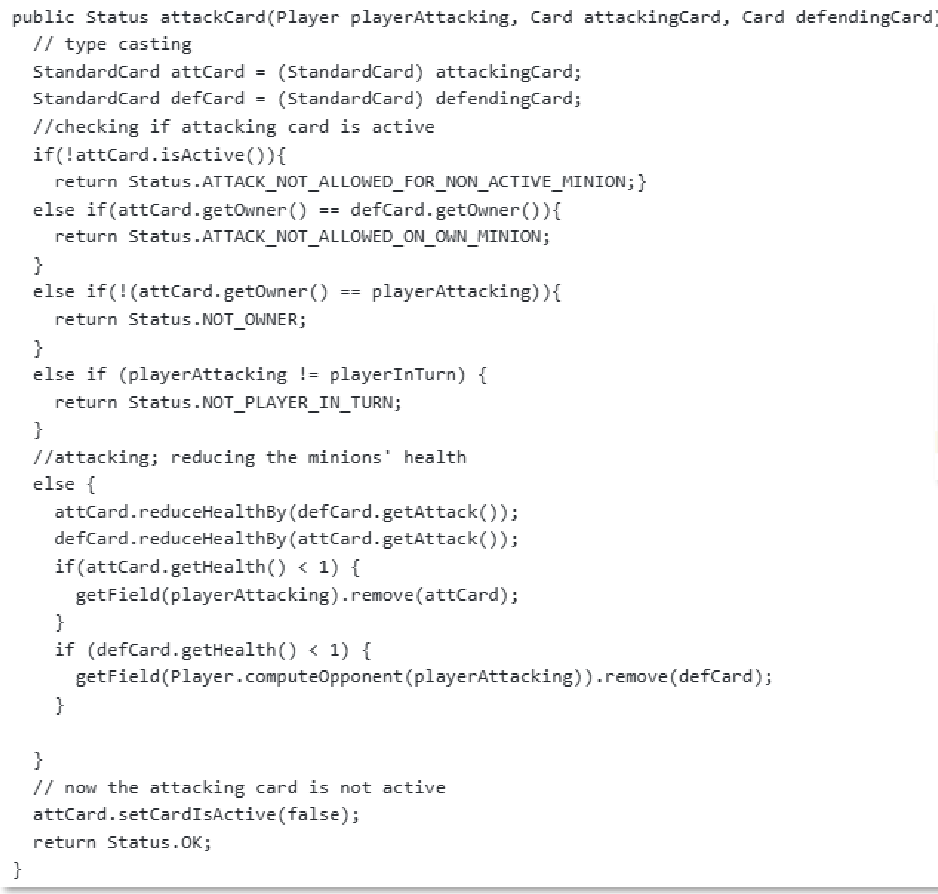
Not more to do?

- One group argues this is the final, cleanest code?

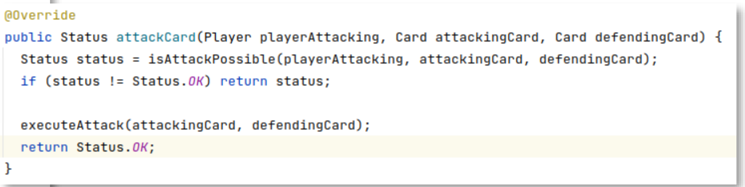


```
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
  // type casting
  StandardCard attCard = (StandardCard) attackingCard;
  StandardCard defCard = (StandardCard) defendingCard;
  //checking if attacking card is active
  if(!attCard.isActive()){
    return Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;}
  else if(attCard.getOwner() == defCard.getOwner()){
    return Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
  }
  else if(!(attCard.getOwner() == playerAttacking)){
    return Status.NOT_OWNER;
  }
  else if (playerAttacking != playerInTurn) {
    return Status.NOT_PLAYER_IN_TURN;
  }
  //attacking; reducing the minions' health
  else {
    attCard.reduceHealthBy(defCard.getAttack());
    defCard.reduceHealthBy(attCard.getAttack());
    if(attCard.getHealth() < 1) {
      getField(playerAttacking).remove(attCard);
    }
    if (defCard.getHealth() < 1) {
      getField(Player.computeOpponent(playerAttacking)).remove(defCard);
    }

  }
  // now the attacking card is not active
  attCard.setCardIsActive(false);
  return Status.OK;
}
```

Compare with Mine?



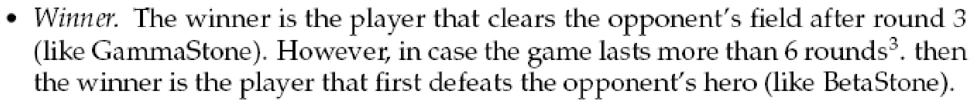
```
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
  // type casting
  StandardCard attCard = (StandardCard) attackingCard;
  StandardCard defCard = (StandardCard) defendingCard;
  //checking if attacking card is active
  if(!attCard.isActive()){
    return Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;}
  else if(attCard.getOwner() == defCard.getOwner()){
    return Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
  }
  else if(!(attCard.getOwner() == playerAttacking)){
    return Status.NOT_OWNER;
  }
  else if (playerAttacking != playerInTurn) {
    return Status.NOT_PLAYER_IN_TURN;
  }
  //attacking; reducing the minions' health
  else {
    attCard.reduceHealthBy(defCard.getAttack());
    defCard.reduceHealthBy(attCard.getAttack());
    if(attCard.getHealth() < 1) {
      getField(playerAttacking).remove(attCard);
    }
    if (defCard.getHealth() < 1) {
      getField(Player.computeOpponent(playerAttacking)).remove(defCard);
    }

  }
  // now the attacking card is not active
  attCard.setCardIsActive(false);
  return Status.OK;
}
```



```
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
  Status status = isAttackPossible(playerAttacking, attackingCard, defendingCard);
  if (status != Status.OK) return status;

  executeAttack(attackingCard, defendingCard);
  return Status.OK;
}
```

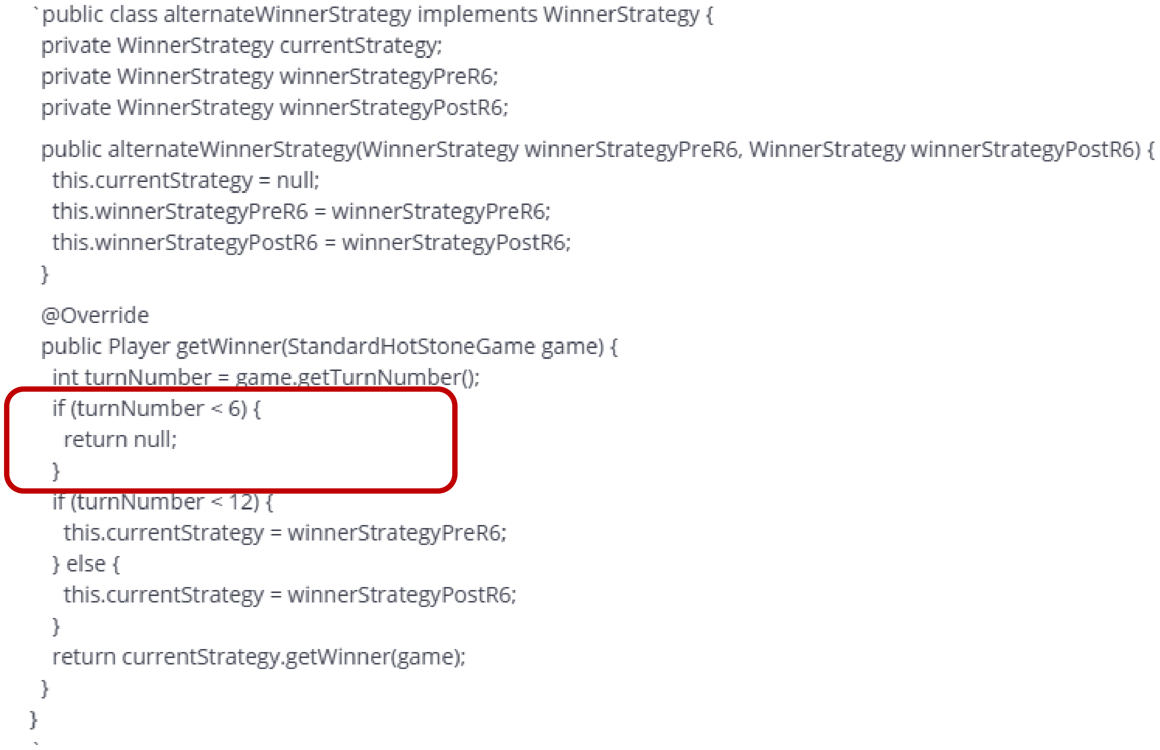
Zeta / State

- Mostly correct, but then..



- *Winner.* The winner is the player that clears the opponent's field after round 3 (like GammaStone). However, in case the game lasts more than 6 rounds[3]. then the winner is the player that first defeats the opponent's hero (like BetaStone).

- *What is the issue with the marked code?*



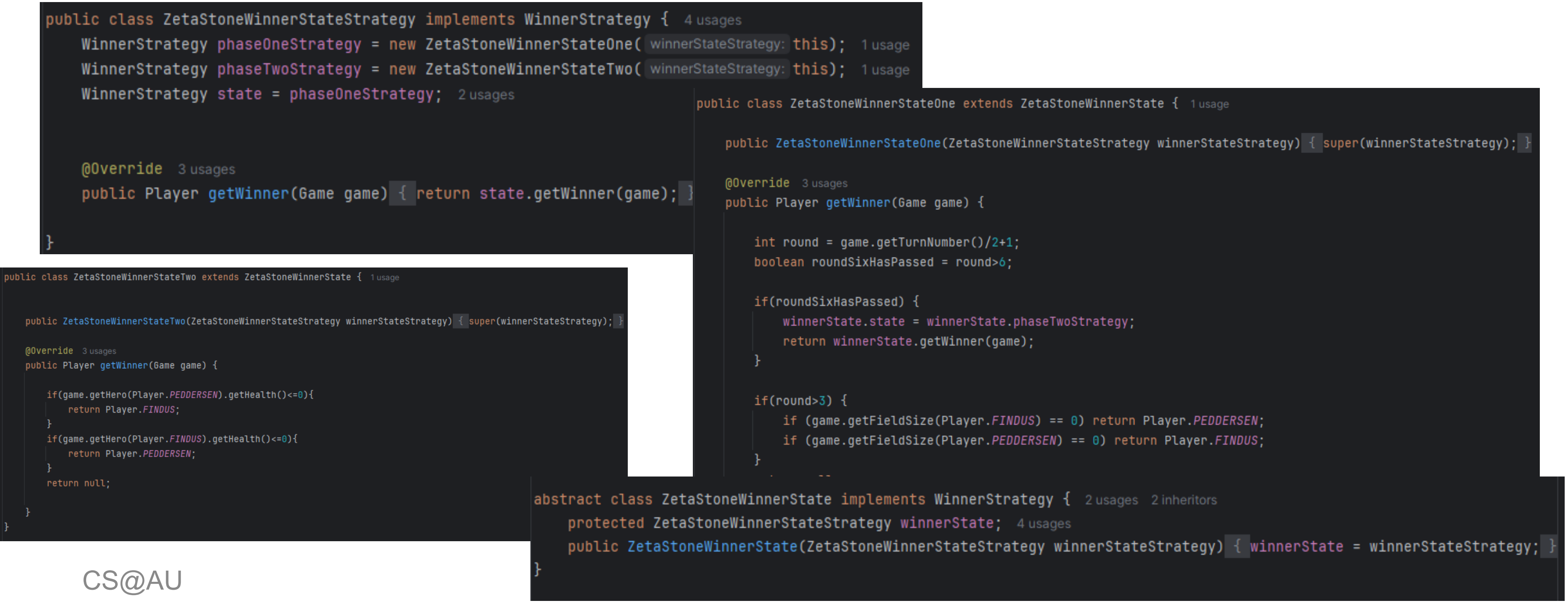
```
`public class alternateWinnerStrategy implements WinnerStrategy {
private WinnerStrategy currentStrategy;
private WinnerStrategy winnerStrategyPreR6;
private WinnerStrategy winnerStrategyPostR6;

public alternateWinnerStrategy(WinnerStrategy winnerStrategyPreR6, WinnerStrategy winnerStrategyPostR6) {
  this.currentStrategy = null;
  this.winnerStrategyPreR6 = winnerStrategyPreR6;
  this.winnerStrategyPostR6 = winnerStrategyPostR6;
}

@Override
public Player getWinner(StandardHotStoneGame game) {
  int turnNumber = game.getTurnNumber();
  if (turnNumber < 6) {
    return null;
  }
  if (turnNumber < 12) {
    this.currentStrategy = winnerStrategyPreR6;
  } else {
    this.currentStrategy = winnerStrategyPostR6;
  }
  return currentStrategy.getWinner(game);
}
}
`
```

- Take care ☺

AARHUS UNIVERSITET

Convoluted Code

- Sometimes (especially in a learning context!) we get functional code that is overly complex, and lacks *analyzability*. Try this:



```java
public class ZetaStoneWinnerStateStrategy implements WinnerStrategy {  4 usages
    WinnerStrategy phaseOneStrategy = new ZetaStoneWinnerStateOne( winnerStateStrategy: this);  1 usage
    WinnerStrategy phaseTwoStrategy = new ZetaStoneWinnerStateTwo( winnerStateStrategy: this);  1 usage
    WinnerStrategy state = phaseOneStrategy;  2 usages


    @Override  3 usages
    public Player getWinner(Game game) { return state.getWinner(game); }

}
```

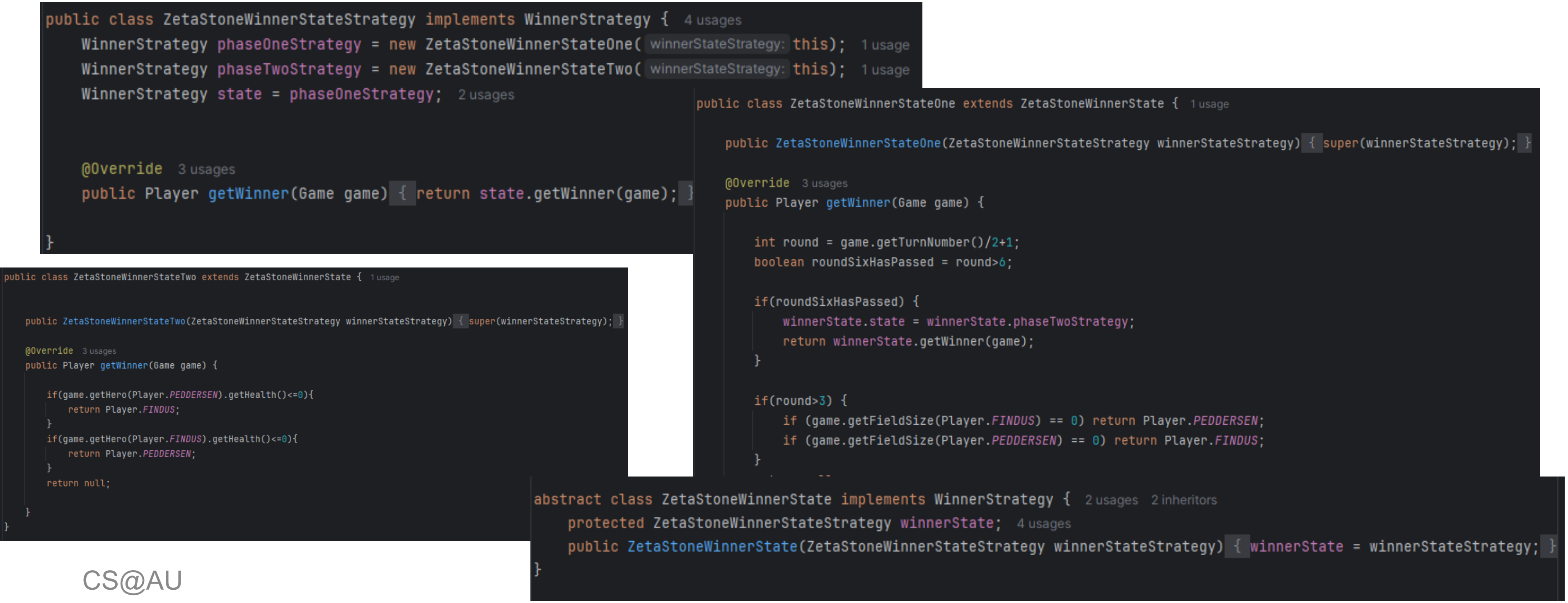
```java
public class ZetaStoneWinnerStateOne extends ZetaStoneWinnerState {  1 usage

    public ZetaStoneWinnerStateOne(ZetaStoneWinnerStateStrategy winnerStateStrategy) { super(winnerStateStrategy); }

    @Override  3 usages
    public Player getWinner(Game game) {

        int round = game.getTurnNumber()/2+1;
        boolean roundSixHasPassed = round>6;

        if(roundSixHasPassed) {
            winnerState.state = winnerState.phaseTwoStrategy;
            return winnerState.getWinner(game);
        }

        if(round>3) {
            if (game.getFieldSize(Player.FINDUS) == 0) return Player.PEDDERSEN;
            if (game.getFieldSize(Player.PEDDERSEN) == 0) return Player.FINDUS;
        }
```

```java
public class ZetaStoneWinnerStateTwo extends ZetaStoneWinnerState {  1 usage


    public ZetaStoneWinnerStateTwo(ZetaStoneWinnerStateStrategy winnerStateStrategy) { super(winnerStateStrategy); }

    @Override  3 usages
    public Player getWinner(Game game) {

        if(game.getHero(Player.PEDDERSEN).getHealth()<=0){
            return Player.FINDUS;
        }
        if(game.getHero(Player.FINDUS).getHealth()<=0){
            return Player.PEDDERSEN;
        }
        return null;

    }
}
```

```java
abstract class ZetaStoneWinnerState implements WinnerStrategy {  2 usages  2 inheritors
    protected ZetaStoneWinnerStateStrategy winnerState;  4 usages
    public ZetaStoneWinnerState(ZetaStoneWinnerStateStrategy winnerStateStrategy) { winnerState = winnerStateStrategy; }
}
```

Convoluted Code

- Abstract class? Source-code-copy? Four classes?
  - Think: **“Aah, this can’t be right???”** *Do Over*



```
public class ZetaStoneWinnerStateStrategy implements WinnerStrategy {  4 usages
    WinnerStrategy phaseOneStrategy = new ZetaStoneWinnerStateOne( winnerStateStrategy: this);  1 usage
    WinnerStrategy phaseTwoStrategy = new ZetaStoneWinnerStateTwo( winnerStateStrategy: this);  1 usage
    WinnerStrategy state = phaseOneStrategy;  2 usages


    @Override  3 usages
    public Player getWinner(Game game) { return state.getWinner(game); }

}
```

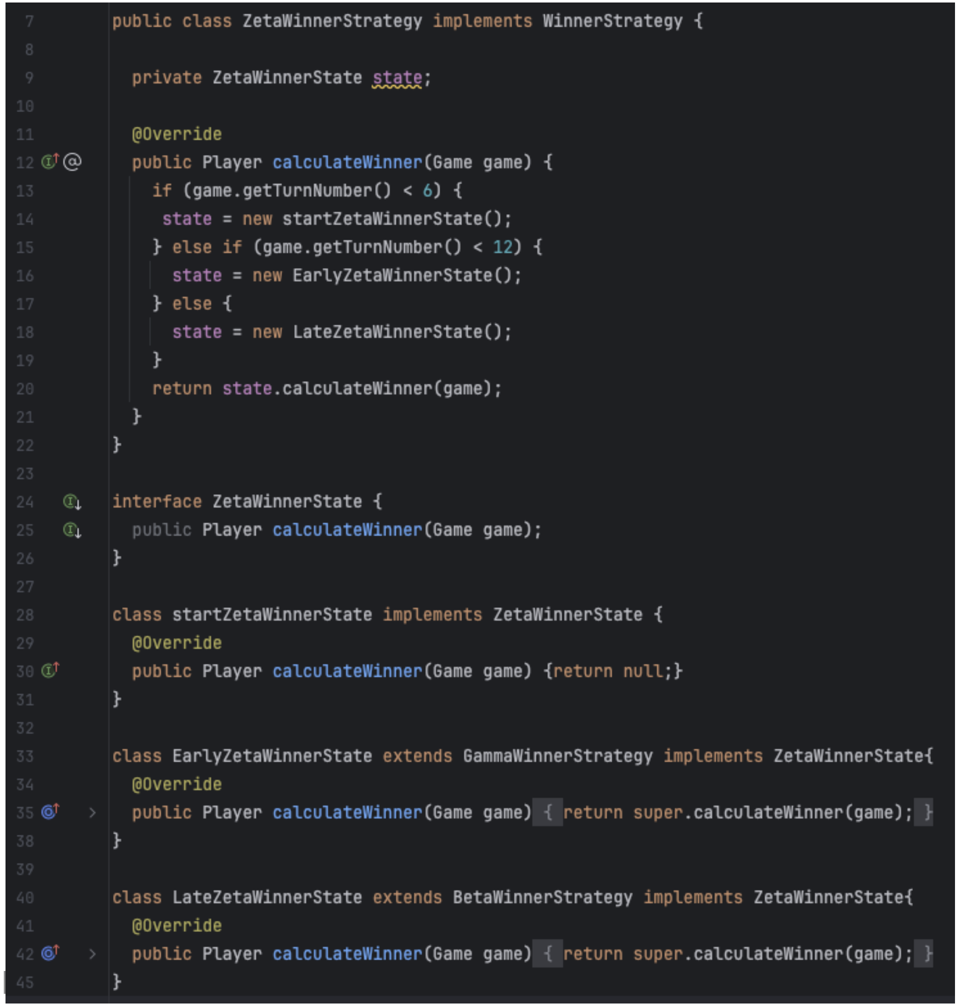
```
public class ZetaStoneWinnerStateOne extends ZetaStoneWinnerState {  1 usage

    public ZetaStoneWinnerStateOne(ZetaStoneWinnerStateStrategy winnerStateStrategy) { super(winnerStateStrategy); }

    @Override  3 usages
    public Player getWinner(Game game) {

        int round = game.getTurnNumber()/2+1;
        boolean roundSixHasPassed = round>6;

        if(roundSixHasPassed) {
            winnerState.state = winnerState.phaseTwoStrategy;
            return winnerState.getWinner(game);
        }

        if(round>3) {
            if (game.getFieldSize(Player.FINDUS) == 0) return Player.PEDDERSEN;
            if (game.getFieldSize(Player.PEDDERSEN) == 0) return Player.FINDUS;
        }
```

```
public class ZetaStoneWinnerStateTwo extends ZetaStoneWinnerState {  1 usage


    public ZetaStoneWinnerStateTwo(ZetaStoneWinnerStateStrategy winnerStateStrategy) { super(winnerStateStrategy); }

    @Override  3 usages
    public Player getWinner(Game game) {

        if(game.getHero(Player.PEDDERSEN).getHealth()<=0){
            return Player.FINDUS;
        }
        if(game.getHero(Player.FINDUS).getHealth()<=0){
            return Player.PEDDERSEN;
        }
        return null;

    }
}
```

```
abstract class ZetaStoneWinnerState implements WinnerStrategy {  2 usages  2 inheritors
    protected ZetaStoneWinnerStateStrategy winnerState;  4 usages
    public ZetaStoneWinnerState(ZetaStoneWinnerStateStrategy winnerStateStrategy) { winnerState = winnerStateStrategy; }
}
```

Subclassing

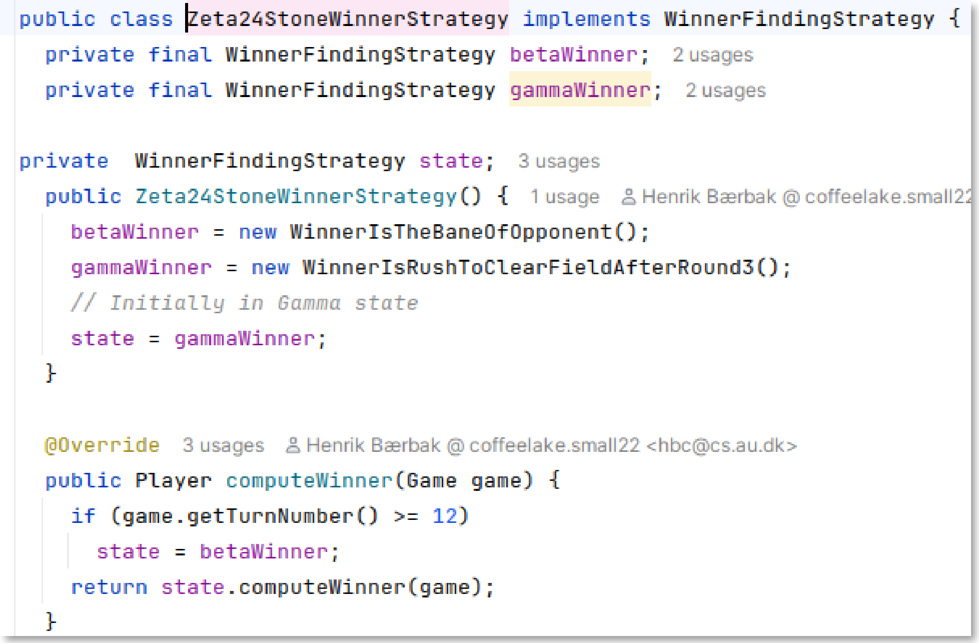
- … needed?
  - Inheritance used just to rename a class?



```
public class ZetaWinnerStrategy implements WinnerStrategy {

  private ZetaWinnerState state;

  @Override
  public Player calculateWinner(Game game) {
    if (game.getTurnNumber() < 6) {
     state = new startZetaWinnerState();
    } else if (game.getTurnNumber() < 12) {
      state = new EarlyZetaWinnerState();
    } else {
      state = new LateZetaWinnerState();
    }
    return state.calculateWinner(game);
  }
}

interface ZetaWinnerState {
  public Player calculateWinner(Game game);
}

class startZetaWinnerState implements ZetaWinnerState {
  @Override
  public Player calculateWinner(Game game) {return null;}
}

class EarlyZetaWinnerState extends GammaWinnerStrategy implements ZetaWinnerState{
  @Override
  public Player calculateWinner(Game game) { return super.calculateWinner(game); }
}

class LateZetaWinnerState extends BetaWinnerStrategy implements ZetaWinnerState{
  @Override
  public Player calculateWinner(Game game) { return super.calculateWinner(game); }
}
```

Not Quite as Convoluted

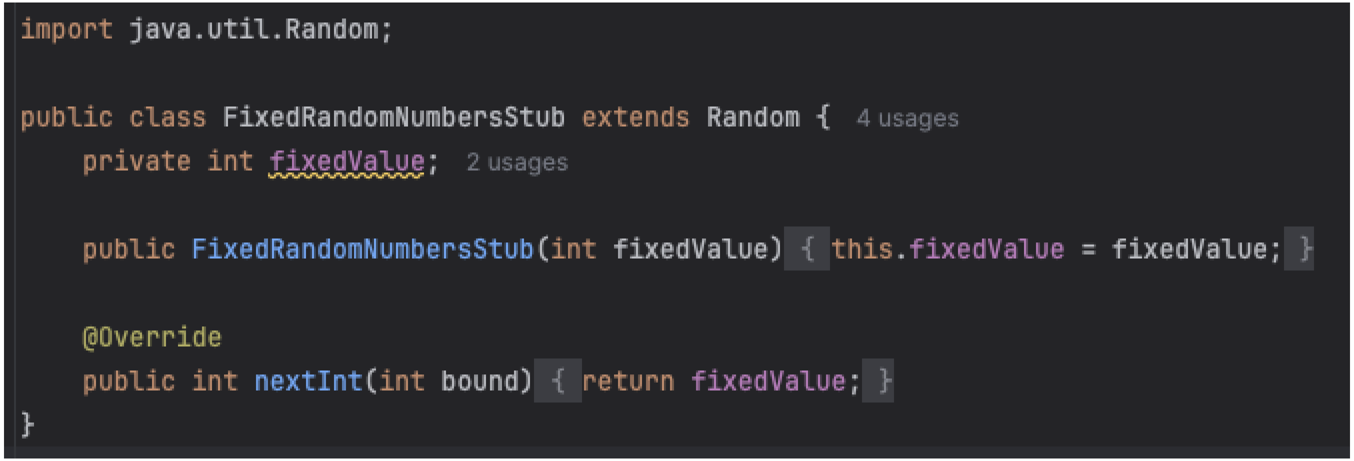
- State pattern just delegates to ConcreteState objects
  - Store the two strategies, set the starting one
  - Switch in case of turn passing 12
  - *Return delegate's opinion on who has won…*
    - Nothing else…
  - **Compositional Design:**
    - ***Let someone else do the job…***



```
public class Zeta24StoneWinnerStrategy implements WinnerFindingStrategy {
  private final WinnerFindingStrategy betaWinner;  2 usages
  private final WinnerFindingStrategy gammaWinner;  2 usages

private  WinnerFindingStrategy state;  3 usages
  public Zeta24StoneWinnerStrategy() {  1 usage  Henrik Bærbak @ coffeelake.small22
    betaWinner = new WinnerIsTheBaneOfOpponent();
    gammaWinner = new WinnerIsRushToClearFieldAfterRound3();
    // Initially in Gamma state
    state = gammaWinner;
  }

  @Override  3 usages  Henrik Bærbak @ coffeelake.small22 <hbc@cs.au.dk>
  public Player computeWinner(Game game) {
    if (game.getTurnNumber() >= 12)
      state = betaWinner;
    return state.computeWinner(game);
  }
```

Test Stub for Randomness

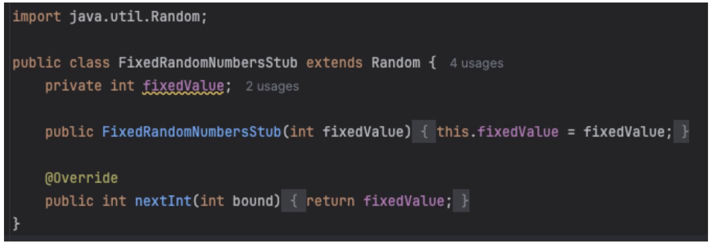
- Some subclass the Java library ‘Random’ class
  - A Double is a *replacement* of a Depended-On Unit



```java
import java.util.Random;

public class FixedRandomNumbersStub extends Random {  4 usages
    private int fixedValue;  2 usages

    public FixedRandomNumbersStub(int fixedValue) { this.fixedValue = fixedValue; }

    @Override
    public int nextInt(int bound) { return fixedValue; }
}
```

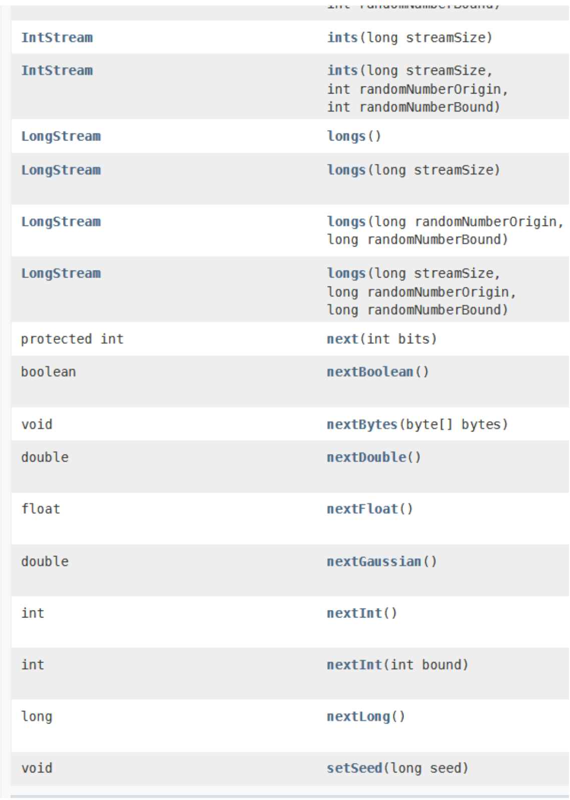
Code Stability

- Random contains many methods!
  - But only one overridden…



```
import java.util.Random;

public class FixedRandomNumbersStub extends Random {  4 usages
    private int fixedValue;  2 usages

    public FixedRandomNumbersStub(int fixedValue) { this.fixedValue = fixedValue; }

    @Override
    public int nextInt(int bound) { return fixedValue; }
}
```

  - *What if I rewrite my EpsilonStone to use nextFloat() instead?*
  - Answer: Unexpected effects!



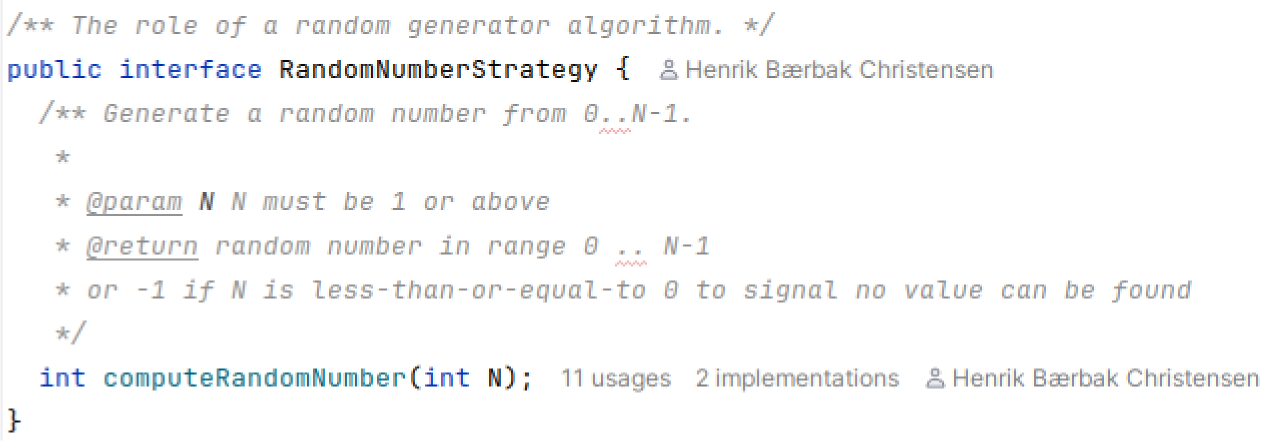
	int randomNumberBound)
IntStream	ints(long streamSize)
IntStream	ints(long streamSize, int randomNumberOrigin, int randomNumberBound)
LongStream	longs()
LongStream	longs(long streamSize)
LongStream	longs(long randomNumberOrigin, long randomNumberBound)
LongStream	longs(long streamSize, long randomNumberOrigin, long randomNumberBound)
protected int	next(int bits)
boolean	nextBoolean()
void	nextBytes(byte[] bytes)
double	nextDouble()
float	nextFloat()
double	nextGaussian()
int	nextInt()
int	nextInt(int bound)
long	nextLong()
void	setSeed(long seed)

Definition: Stability (ISO 9126)

The capability of the software product to avoid unexpected effects from modifications of the system.

Code Stability

- Morale: Let your Test Double define only the *single responsibility* of *‘make a random index of who to effect on the battle field of size n’*
  - *High cohesion, low coupling*



```
/** The role of a random generator algorithm. */
public interface RandomNumberStrategy {  Henrik Bærbak Christensen
  /** Generate a random number from 0..N-1.
   *
   * @param N N must be 1 or above
   * @return random number in range 0 .. N-1
   * or -1 if N is less-than-or-equal-to 0 to signal no value can be found
   */
  int computeRandomNumber(int N);  11 usages  2 implementations  Henrik Bærbak Christensen
}
```