



AARHUS UNIVERSITET

# **Software Engineering and Architecture**

Writing Clean Code

- Code is written to
  - Be compiled, deployed, and executed by users in order to serve their need
    - So we make money, get salaries and can buy presents for our spouses (or children, or cats, or whatever...)
  - **Be maintained – that is - read and understood by humans so it can easily and correctly modified**

- *Functionally correct code can be next to impossible to maintain...*
  - *The code below correctly reflect a well known text book example. Which?*

```
public class X{private int y;public X(){y = 0;}public int z(){  
return y;}public void z1(int z0){y += z0;}public static void main(  
String[] args){X y=new X();y.z1(200);y.z1(3400);System.out.println  
("Result is "+ y.z());}}
```

- *Morale: Write Clean Code*

## Definition: Analyzability (ISO 9126)

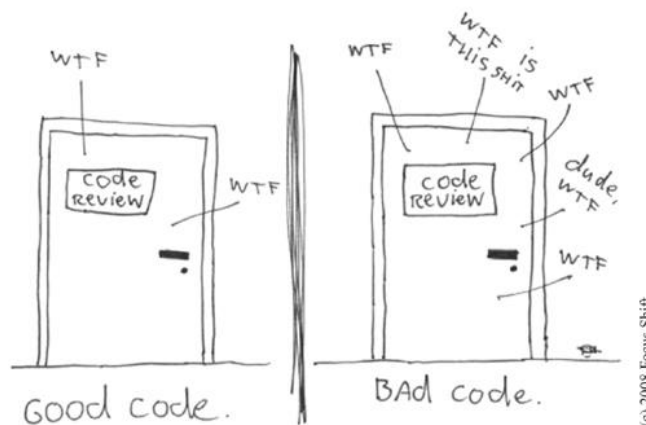
The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

- Basically:
  - can I *understand* the code *fast*?
  - Is my understanding *correct*?

# How to ?

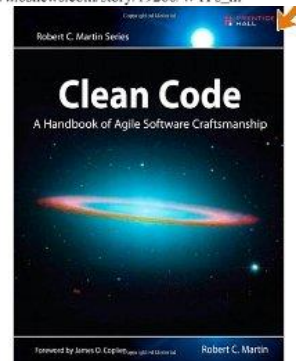
- Clean Code?
  - *Not an exact science !!!*
    - Kent Beck: "Code smell"
    - WTF measure
  - *Certainly partly matter of taste!*
    - *Not invented here syndrome...*
- Our Take at it
  - Uncle Bob "Clean Code"
    - On Functions and Comments
  - Uncle Henrik
    - My own prejudices and tastes ☺

The ONLY VALID MEASUREMENT  
OF CODE QUALITY: WTFs/MINUTE



Reproduced with the kind permission of Thom Holwerda.  
[http://www.osnews.com/story/19266/WTFs\\_m](http://www.osnews.com/story/19266/WTFs_m)

(c) 2008 Focus Shift



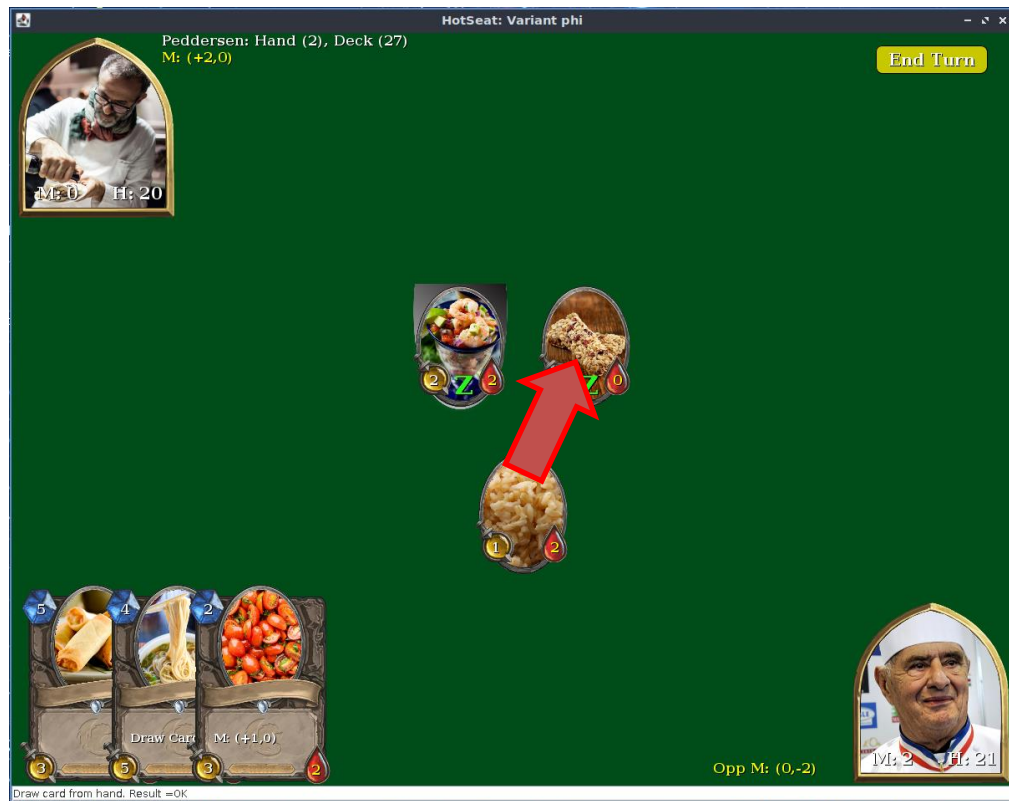
# Exercise

- We will be looking at *game.attackCard(p, aC, dC)*

- For AlphaStone

- My unclean impl:

- Used own test cases
- Developed attackCard (and other) methods from scratch
  - With inspiration from student's code 😊



- attackCard(Player attPlayer, Card attCard, Card defCard)
- (Weird) Choice of data structures in StandardGame
  - Battlefield is ARRAY with index 0 = findus and 1 = peddersen
  - Hand is Map with key = player and value = List<Card>

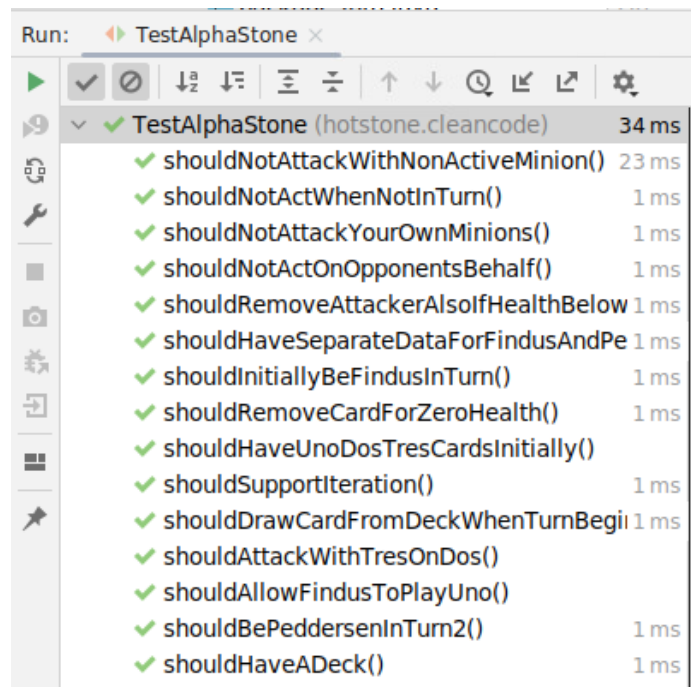
```
12 usages
private ArrayList<Card>[] field;
10 usages
private Map<Player, List<Card>> hand = new HashMap<>();
```

- The implementation ensures all Cards are instances of StandardCard, so casts are valid, to access mutators on cards ala

```
StandardCard atC = (StandardCard) attackingCard;
StandardCard defender = (StandardCard) defendingCard;
// Findus attacks the card
atC.lowerHealthBy(defender.getAttack());
defender.lowerHealthBy(atC.getAttack());
```

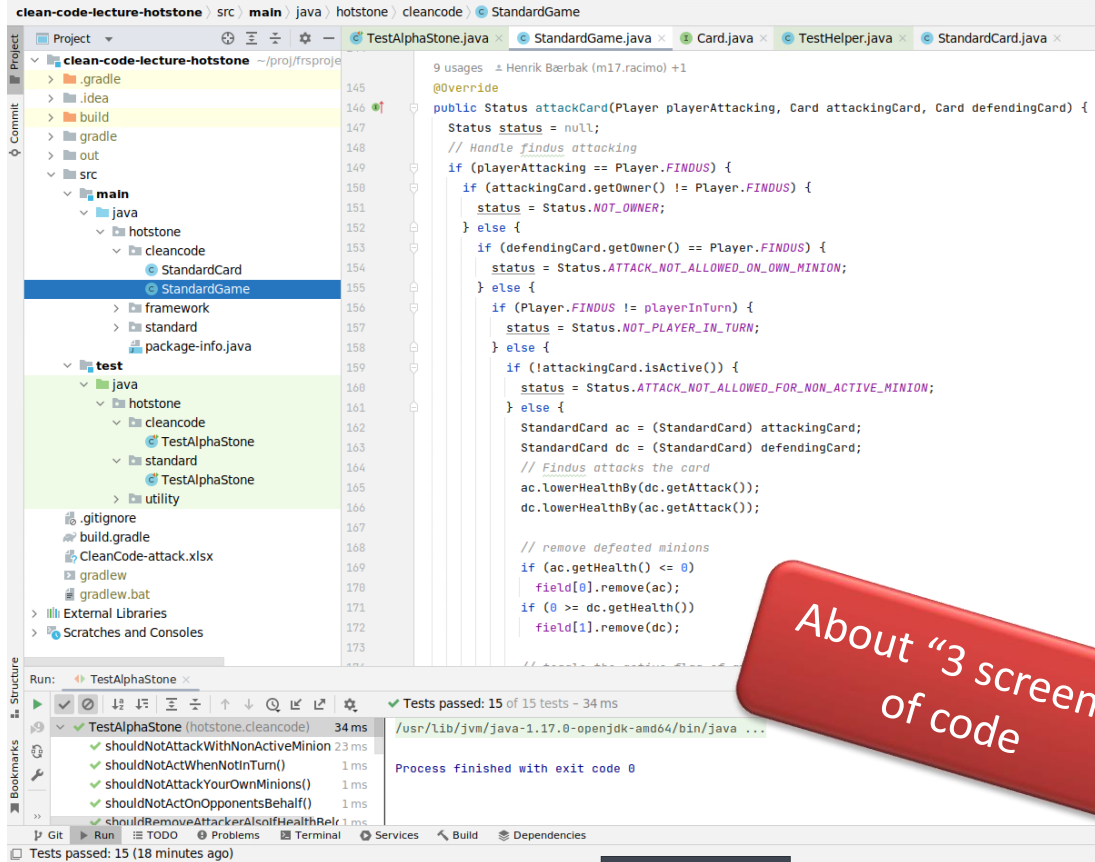
# TDD TestList => Test Cases

- My own test suite of AlphaStone, however, removed all 'non attack card' related stuff...





# The Method



```

clean-code-lecture-hotstone | src | main | java | hotstone | cleancode | StandardGame
Project
  clean-code-lecture-hotstone ~/proj/frsproj
    .gradle
    .idea
    build
    gradle
    out
    src
      main
        java
          hotstone
            cleancode
              StandardCard
              StandardGame
            framework
            standard
            package-info.java
          test
            java
              hotstone
                cleancode
                  TestAlphaStone
                standard
                  TestAlphaStone
                utility
            gitignore
            build.gradle
            CleanCode-attack.xlsx
            gradlew
            gradlew.bat
            External Libraries
            Scratches and Consoles
Run: TestAlphaStone
Tests passed: 15 of 15 tests - 34 ms
TestAlphaStone (hotstone.cleancode) 34 ms
  shouldNotAttackWithNonActiveMinion 23 ms
  shouldNotActWhenNotInTurn() 1 ms
  shouldNotAttackYourOwnMinions() 1 ms
  shouldNotActOnOpponentsBehalf() 1 ms
  shouldRemoveAttackerAndMinionHealth 1 ms
Process finished with exit code 0

```

About "3 screens"  
of code

```

@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    Status status = null;
    // Handle Findus attacking
    if (playerAttacking == Player.FINDUS) {
        if (attackingCard.getOwner() != Player.FINDUS) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.FINDUS) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
                if (Player.FINDUS != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard dc = (StandardCard) defendingCard;
                        // Findus attacks the card
                        dc.lowerHealthBy(dc.getAttack());
                        dc.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0) {
                            field[0].remove(ac);
                        }
                        if (0 >= dc.getHealth()) {
                            field[1].remove(dc);
                        }

                        // toggle the active flag of attacker
                        ac.setActive(false);
                        status = Status.OK;
                    }
                }
            }
        }
    } else { // ===== it is pedersen attacking
        if (attackingCard.getOwner() != Player.PEDERSEN) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.PEDERSEN) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
                if (Player.PEDERSEN != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard defender = (StandardCard) defendingCard;
                        // Findus attacks the card
                        ac.lowerHealthBy(defender.getAttack());
                        defender.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0) {
                            field[1].remove(ac);
                        }
                        if (defender.getHealth() <= 0) {
                            field[0].remove(defender);
                        }

                        // toggle the active flag of attacker
                        ac.setActive(false);
                        status = Status.OK;
                    }
                }
            }
        }
    }
    return status;
}

```

Functionally correct  
AlphaStone attack!  
(Also on BS.)

# Clean Code Properties

A Classification Template

# Classification Scheme

- An attempt at systematics...

Method Name:		
Wanted property	is OK	Example/argument
Small		
Do One Thing		
One Level of Abstraction		
Use Descriptive Names		
Keep Number of Arguments Low		
Avoid Flag Arguments		
Have No Side Effects		
Command Query Separation		
Prefer Exceptions to Error Codes		
Don't Repeat Yourself		
Do the Same Thing, the Same Way		
Name Boolean Expressions		
Bail out Fast		
Arguments in Argument Lists		

# Bob's Properties

- Small
  - Make it do 1-5 logical steps / 'functional nuggets'
- Do One Thing
  - Do one thing, do it well, do it only! *Keep focus!*
- One Level of Abstraction
  - The dean does not edit spelling errors in my slides!
- Use Descriptive Names
  - Tell the one thing it does! Do not tell anything else
- Keep Number of Arguments low
  - 0-1-2 maybe three arguments. No more. If more, your method does *more than one thing!*

- Think code as a military hierarchy
  - *General*: overall movement of armies
    - *Major*: executive of battalions
      - *Private*: wading through mud
- Example: A Point-of-Sales system / "Føtex kasse"
  - Goal:
    - Scan purchased items
    - Produce till receipt
    - Update warehouse inventory



# A Properly Leveled Implementation

AARHUS UNIVERSITET

```
// Highest level of abstraction
public void processItem(BarCode scannedCode) {
    Item itemScanned = barcodeSystem.lookup(scannedCode);
    addItemToReceipt(itemScanned);
    updateRunningTotal(item.cost());
    decrementWarehouseInventory(item);
}

TillReceipt currentReceipt;
// Second level of abstraction
public void addItemToReceipt(Item itemToAdd) {
    currentReceipt.addItem(itemToAdd);
    if (isBottleWithDeposit(itemToAdd)) {
        Item bottleDeposit = computeDepositFor(itemToAdd);
        currentReceipt.addItem(bottleDeposit);
    }
}

// Third level of abstraction
Item computeDepositFor(Item bottleItem) {
    if (bottleItem.getType() == BOTTLES.MineralWater) {
        return new Item("Pant", 0.50);
    }
    if (bottleItem.getType() == BOTTLES.Soda) {
        ...
    }
    ...
}
```

Overall Algorithm level  
'General'

TillReceipt handling level  
'Major'

Low level computation  
level  
'Private'

LIDL	
Heegermühlerstr. 1	
15225 Eberswalde	
Mo-Sa 8-21 Uhr So geschlossen	
Milchmischgetränk	0,49 A
Bio Haferdrink	1,19 B
Jacobs Krön. Kräftig	7,98 A
2 x	3,55
Mineralwasser still	0,38 B
2 x	0,19
Pfand 0,25 EM	0,50 B
2 x	0,25
zu zahlen	
10,54	

# Bob's Properties

- Avoid Flag Arguments
  - `produceWebPage(true, true, false);`      Huh???
  - *Boolean arguments says "do **more than one thing!**"*      Right?
- Have No Side Effects
  - Do not do *hidden things / hidden state changes*
    - It will fool the client; and will hide weird bugs
  - Ex: init a session; modify the object passed as argument
    - If it does, the **descriptive name**, should reflect it!

# Bob's Properties

- Command Query Separation
  - Setters and Getters          Accessors and Mutators
  - Query:                      no state change!!!          Return a value.
  - Command:                  no return value(hm)          Change the state
- Prefer Exceptions to Error Codes
  - Not 'int addPayment(int amount)' that returns error code 17 in case it is an illegal coin
    - Networking is an exception – it cannot propagate exceptions
- Don't Repeat Yourself
  - Avoid Duplicated Code
    - To avoid *multiple maintenance problem*



# Don't Repeat Yourself

- That is

Avoid Duplication

- Why?
  - Multiple maintenance problem !!!
  - *Root of (almost) all Evil*

```
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    Status status = null;
    // Handle Findus attacking
    if (playerAttacking == Player.FINDUS) {
        if (attackingCard.getOwner() != Player.FINDUS) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.FINDUS) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
                if (Player.FINDUS != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard dc = (StandardCard) defendingCard;
                        // Findus attacks the card
                        ac.lowerHealthBy(dc.getAttack());
                        dc.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0)
                            field[0].remove(ac);
                        if (0 != dc.getHealth())
                            field[1].remove(dc);

                        // toggle the active flag of attacker
                        ac.setActive(false);
                        status = Status.OK;
                    }
                }
            }
        }
    } else { // ===== it is pedersen attacking
        if (attackingCard.getOwner() != Player.PEDERSEN) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.PEDERSEN) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
                if (Player.PEDERSEN != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                    } else {
                        StandardCard attc = (StandardCard) attackingCard;
                        StandardCard defender = (StandardCard) defendingCard;
                        // Findus attacks the card
                        attc.lowerHealthBy(defender.getAttack());
                        defender.lowerHealthBy(attc.getAttack());

                        // remove defeated minions
                        if (attc.getHealth() <= 0)
                            field[1].remove(attc);
                        if (defender.getHealth() <= 0)
                            field[0].remove(defender);

                        // toggle the active flag of attacker
                        attc.setActive(false);
                        status = Status.OK;
                    }
                }
            }
        }
    }
    return status;
}
```

```

@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    Status status = null;
    // Handle findus attacking
    if (playerAttacking == Player.FINDUS) {
        if (attackingCard.getOwner() != Player.FINDUS) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.FINDUS) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
                if (Player.FINDUS != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (!attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard dc = (StandardCard) defendingCard;
                        // Findus attacks the card
                        ac.lowerHealthBy(dc.getAttack());
                        dc.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0)
                            field[0].remove(ac);
                        if (0 >= dc.getHealth())
                            field[1].remove(dc);

                        // toggle the active flag of attacker
                        ac.setActive(false);

                        status = Status.OK;
                    }
                }
            }
        }
    } else { // ===== it is peddersen attacking
        if (attackingCard.getOwner() != Player.PEDDERSEN) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.PEDDERSEN) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {

```



AARHUS UNIVERSITET

# Clean Code

Additions by Uncle Henrik

# Arguments in Argument Lists

- One symptom on duplicated code is the use of 'arguments' in the method/class **names**
  - addOneToX(int x); addTwoToX(int x); addThreeToX(int x); ???

```
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    Status status = null;
    // Handle findus attacking
    if (playerAttacking == Player.FINDUS) {
        return handleAttackForFindus(attackingCard, defendingCard);
    } else { // ===== it is peddersen attacking
        return peddersensAttacking(defendingCard, attackingCard);
    }
}
```

An argument  
appears as part of  
the method name

Important exception!  
Test case methods often  
hardcode values and  
parameters in the method  
names



# Do the Same Thing, the Same Way

- Akin to Uncle Bob's *Do One Thing* but not quite...

Do the same thing, the same way

- Why?

- Analyzability

- WarStory

- DSE 'string copy'
- *Changed the ordering of arguments compared to standard !?!?!?*

```
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    Status status = null;
    // Handle Findus attacking
    if (playerAttacking == Player.FINDUS) {
        return handleAttackForFindus(attackingCard, defendingCard);
    } else { // ===== it is peddersen attacking
        return peddersenAttack(attackingCard, defendingCard);
    }
}
```

# Do the Same Thing, the Same Way

Do the same thing, the same way

- Within classes you have the option to either

- Use accessor method
- Or direct data structure access

- *Do same thing same way...*

- *Use the access method*
  - *Allows changing datastructure!*

```
private boolean moveRedUnit( Position from,
    i → (unitMap.get(from) != null &&
        !world.get(to).getTypeString().eq
        unitMap.get(from).getOwner() == P
        !getTileAt(to).getTypeString().eq
        getUnitAt(from).getMoveCount() >=
        Math.abs(from.getColumn() - to.ge
        Math.abs(from.getRow() - to.getRo
        ! to.equals(from) ) {
```

```
// System.out.println("--> "+ getUnitAt
```

```
// if the to tile is free or its unit i
if ( getUnitAt(to) == null ) {
```

```
// move is allowed to proceed to move
unitMap.put(to, unitMap.get(from));
```

- *Do the same thing the same way...*
  - *Global consistency in all respects is difficult 😊*

```
csdev@m33:~/proj/privatekata/gfxtestbed$ git pull
Already up to date.
csdev@m33:~/proj/privatekata/gfxtestbed$ git push
Everything up-to-date
```

- *Do statements end in a dot or not?*
- *Is 'up-to-date' with or without dashes..*

# Name Boolean Expressions

- Boolean expressions are difficult to read!

```
// System.out.println("--> move from " + from + " to "+to);
if (unitMap.get(from) != null &&
    !world.get(to).getTypeString().equals(GameConstants.OCEANS) &&
    unitMap.get(from).getOwner() == Player.RED &&
    !getTileAt(to).getTypeString().equals(GameConstants.MOUNTAINS) &&
    getUnitAt(from).getMoveCount() >= 1 &&
    Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
    Math.abs(from.getRow() - to.getRow()) <= 1 &&
    ! to.equals(from) ) {

    // System.out.println("--> "+ getUnitAt(to) + " / " + getPlayerInTurn());

    // if the to tile is free or its unit is not my own (not stacking)
    if ( getUnitAt(to) == null ) {
```

- One big ball of mud of && between boolean computations ☹



# Name Boolean Expressions

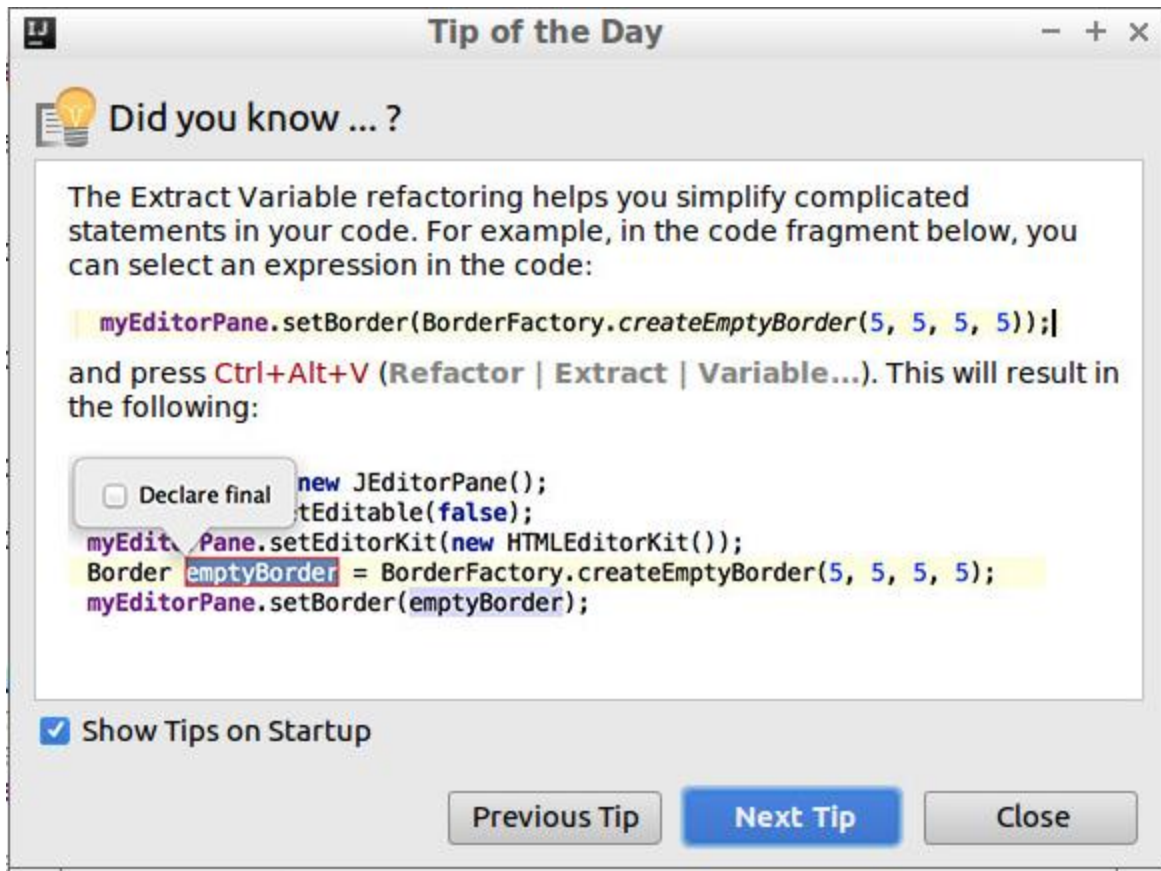
- Name each subexpression so we can read what it is!

Give each boolean expression a name.  
Make name *positive*!

```
boolean isOwningAttackingCard =  
    attackingCard.getOwner() == playerAttacking;  
if (!isOwningAttackingCard) return Status.NOT_OWNER;  
  
boolean isOwningDefendingCard =  
    defendingCard.getOwner() == playerAttacking;  
if (isOwningDefendingCard) return Status.ATTACK_NOT_ALLOWED_ON_OWN;  
  
boolean isItAttackersTurn =  
    getPlayerInTurn() == playerAttacking;  
if (!isItAttackersTurn) return Status.NOT_PLAYER_IN_TURN;
```

Do not put 'not' into the  
Boolean name:  
*isNotMyCard*  
then what is  
if (! isNotMyCard) ... ???

# And Help from My Friends



# Bail out fast

- When lots of conditions must be checked, I often see *deep nesting of if*

Empirical studies show that humans cannot cope well with nesting levels deeper than 1-2 !

```
if (playerAttacking == Player.FINDUS) {
    if (attackingCard.getOwner() != Player.FINDUS) {
        status = Status.NOT_OWNER;
    } else {
        if (defendingCard.getOwner() == Player.FINDUS) {
            status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
        } else {
            if (Player.FINDUS != playerInTurn) {
                status = Status.NOT_PLAYER_IN_TURN;
            } else {
                if (!attackingCard.isActive()) {
                    status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                } else {
                    StandardCard ac = (StandardCard) attackingCard;
                    StandardCard dc = (StandardCard) defendingCard;
                    // Findus attacks the card
                    ac.lowerHealthBy(dc.getAttack());
                    dc.lowerHealthBy(ac.getAttack());

                    // remove defeated minions
                    if (ac.getHealth() <= 0)
                        field[0].remove(ac);
                    if (0 >= dc.getHealth())
                        field[1].remove(dc);

                    // toggle the active flag of attacker
                    ac.setActive(false);

                    status = Status.OK;
                }
            }
        }
    }
}
```

You can write it, but cannot maintain it

# Bail out fast

- Flatten it, by *bailing out as soon as an answer can be computed*

## Bail out fast

```
boolean isOwningAttackingCard =  
    attackingCard.getOwner() == playerAttacking;  
if (!isOwningAttackingCard) return Status.NOT_OWNER;  
  
boolean isOwningDefendingCard =  
    defendingCard.getOwner() == playerAttacking;  
if (isOwningDefendingCard) return Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;  
  
boolean isItAttackersTurn =  
    getPlayerInTurn() == playerAttacking;  
if (!isItAttackersTurn) return Status.NOT_PLAYER_IN_TURN;
```

# Agile Method Development

- *Like any other creative process, you do not do it in one brilliant step!*
- Kent Beck:
  - Clean code that works *but not in that order*
    - *Make it work, then make it clean*
  - *Commit horrible sins, and then clean up the mess*



AARHUS UNIVERSITET

# Analysis



# Five minutes: Spot properties

Method Name:		
Wanted property	is OK	Example/argument
Small		
Do One Thing		
One Level of Abstraction		
Use Descriptive Names		
Keep Number of Arguments Low		
Avoid Flag Arguments		
Have No Side Effects		
Command Query Separation		
Prefer Exceptions to Error Codes		
Don't Repeat Yourself		
Do the Same Thing, the Same Way		
Name Boolean Expressions		
Bail out Fast		
Arguments in Argument Lists		

```
@Override
public Status attackCard(Player playerAttacking, Card attackingCard, Card defendingCard) {
    Status status = null;
    // Handle findus attacking
    if (playerAttacking == Player.FINDUS) {
        if (attackingCard.getOwner() != Player.FINDUS) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.FINDUS) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
                if (Player.FINDUS != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (!attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard dc = (StandardCard) defendingCard;
                        // Findus attacks the card
                        ac.lowerHealthBy(dc.getAttack());
                        dc.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0)
                            field[0].remove(ac);
                        if (0 >= dc.getHealth())
                            field[1].remove(dc);

                        // toggle the active flag of attacker
                        ac.setActive(false);

                        status = Status.OK;
                    }
                }
            }
        }
    }
    } else { // ===== it is peddersen attacking
        if (attackingCard.getOwner() != Player.PEDDERSEN) {
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.PEDDERSEN) {
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
            } else {
```



# My Analysis

Method Name: attackCard		
Wanted property	is OK	Example/argument
Small	No	About 3 full screens of code is quite a lot
Do One Thing	No (yes)	Does validation, attacks, minion removal (but supposed to!)
One Level of Abstraction	No	It does it all! Like data structure manipulations directly
Use Descriptive Names	No	ac and dc are short, undescriptive names, (currents?)
Keep Number of Arguments Low	Yes	
Avoid Flag Arguments	Yes	
Have No Side Effects	Yes	No side effect except given by specification, which is Ok
Command Query Separation	Yes (mutator)	Returns a value, but it is the outcome of the mutation
Prefer Exceptions to Error Codes	n/a	
Don't Repeat Yourself	No	A horrible src-code-copy of the full algorithm! Removal code duplicated
Do the Same Thing, the Same Way	No	
Name Boolean Expressions	No	No expression is named. Not soo bad here, as they are all one-liners
Bail out Fast	No	Deep nesting level lowers analyzability
Arguments in Argument Lists	Yes	No 'attackByFindus(...)' and 'attackByPeddersen(...)' methods

```
@Override
public Status attackCard(Player playerAttacking,
    Status status = null;
    // Handle findus attacking
    if (playerAttacking == Player.FINDUS) {
        if (attackingCard.getOwner() != Player.FINDUS)
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.FINDUS)
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_TEAM;
            } else {
                if (Player.FINDUS != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (!attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_ON_DEAD_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard dc = (StandardCard) defendingCard;
                        // Findus attacks the card
                        ac.lowerHealthBy(dc.getAttack());
                        dc.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0)
                            field[0].remove(ac);
                        if (0 >= dc.getHealth())
                            field[1].remove(dc);

                        // toggle the active flag of attacker
                        ac.setActive(false);

                        status = Status.OK;
                    }
                }
            }
        }
    } else { // ===== it is peddersen attacking
        if (attackingCard.getOwner() != Player.PEDDERSEN)
            status = Status.NOT_OWNER;
        } else {
            if (defendingCard.getOwner() == Player.PEDDERSEN)
                status = Status.ATTACK_NOT_ALLOWED_ON_OWN_TEAM;
            } else {
                if (Player.PEDDERSEN != playerInTurn) {
                    status = Status.NOT_PLAYER_IN_TURN;
                } else {
                    if (!attackingCard.isActive()) {
                        status = Status.ATTACK_NOT_ALLOWED_ON_DEAD_MINION;
                    } else {
                        StandardCard ac = (StandardCard) attackingCard;
                        StandardCard dc = (StandardCard) defendingCard;
                        // Peddersen attacks the card
                        ac.lowerHealthBy(dc.getAttack());
                        dc.lowerHealthBy(ac.getAttack());

                        // remove defeated minions
                        if (ac.getHealth() <= 0)
                            field[0].remove(ac);
                        if (0 >= dc.getHealth())
                            field[1].remove(dc);

                        // toggle the active flag of attacker
                        ac.setActive(false);

                        status = Status.OK;
                    }
                }
            }
        }
    }
}
```





AARHUS UNIVERSITET

# **Refactoring Session**

Screencasted Coding Session



# Let us clean up...

```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
clean-code-lecture-hotstone | src | main | java | hotstone | cleancode | StandardGame | attackCard

Project
clean-code-lecture-hotstone ~ /proj/hrsproj
  .gradle
  .idea
  build
  gradle
  out
  src
    main
      java
        hotstone
          cleancode
            StandardCard
            StandardGame
        framework
        standard
        package-info.java
    test
      java
        hotstone
          cleancode
            TestAlphaStone
          standard
            TestAlphaStone
          utility
  .gitignore
  build.gradle
  CleanCode-attack.xlsx
  gradlew
  gradlew.bat
  External Libraries
  Scratches and Consoles

Run: TestAlphaStone x
Tests passed: 15 of 15 tests - 30 ms
TestAlphaStone (hotstone.cleancode) 30 ms /usr/lib/jvm/java-1.17.0-openjdk-amd64/bin/java ...
  shouldNotAttackWithNonActiveMinion 22 ms
  shouldNotActWhenNotInTurn() 2 ms
  shouldNotAttackYourOwnMinions()
Process finished with exit code 0
```

Find screencasts on the week schedule!

# 45-60 Minutes Later

```
@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;
}
```

Method Name:		
Wanted property	is OK	Example/argument
Small		
Do One Thing		
One Level of Abstraction		
Use Descriptive Names		
Keep Number of Arguments Low		
Avoid Flag Arguments		
Have No Side Effects		
Command Query Separation		
Prefer Exceptions to Error Codes		
Don't Repeat Yourself		
Do the Same Thing, the Same Way		
Name Boolean Expressions		
Bail out Fast		
Arguments in Argument Lists		

# Next level methods

```
private Status isAttackPossible(Player playerAttacking, Card attackingCard, Card defendingCard) {
    boolean isOwningAttackingCard =
        attackingCard.getOwner() == playerAttacking;
    if (!isOwningAttackingCard) {
        return Status.NOT_OWNER;
    }

    boolean isOwningDefendingCard =
        defendingCard.getOwner() == playerAttacking;
    if (isOwningDefendingCard) {
        return Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
    }

    boolean isItAttackersTurn =
        getPlayerInTurn() == playerAttacking;
    if (!isItAttackersTurn) {
        return Status.NOT_PLAYER_IN_TURN;
    }

    boolean isAttackingCardActive =
        attackingCard.isActive();
    if (!isAttackingCardActive) {
        return Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
    }
    return Status.OK;
}
```

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

    removeCardIfDefeated(attackCard);
    removeCardIfDefeated(defendingCard);

    deactivateCard(attackCard);
}
```

```
private void deactivateCard(Card attackingCard) {
    ((StandardCard) attackingCard).setActive(false);
}
```

```
private void reduceCardHealth(Card card, int attackValue) {
    StandardCard stdCard = (StandardCard) card;
    stdCard.lowerHealthBy(attackValue);
}
```

```
private void removeCardIfDefeated(Card card) {
    if (card.getHealth() <= 0) {
        field[card.getOwner().ordinal()].remove(card);
    }
}
```



AARHUS UNIVERSITET

# Afterthoughts

# The Two Codebases

- TDD produce two large codebases
  - The production code
  - The test code
- The properties of clean code apply to both
  - You want to be able to read test code also! (Analyzability)
  - **Except one property: Arguments in Argument Lists**
    - In test code, you often ‘hardwire parameters’ / no abstraction
    - Example: `private void moveRedArcherToPosition45()`
      - Encapsulates the moves of particular unit to particular position in order to do testing!