Lecture 08

ECE 1145: Software Construction and Evolution

Coding Standards, Documentation, and Maintainability

(CH 3, 10)

Announcements

- Iteration 3 due Oct. 3
  - Refactoring, Strategy Pattern
  - Introduce Strategy in several places – make a refactoring/test list and divide up the work

- Resources:
  - Henrik Christensen ”Clean Code” reading on Canvas
  - Livable Code
    - Thread: https://twitter.com/i/events/843392359903649792
    - Video: https://www.youtube.com/watch?app=desktop&v=UDp6eDCi1Vo

Announcements

Looking ahead:

- Next week's assignment will be **code review**
  - Provide another group access to your Iteration 3 code (Release3)
  - Perform code review by Oct. 10
    - Meet with / contact the other group to clarify any questions, provide comments with appropriate context
    - Use time in class next week
- Then, Iteration 4 will be code quality improvements

Announcements

Iteration 3:
GammaCiv World
in color

Questions for Today

How do we make sure our code is understandable (and therefore maintainable)?

Why do we code?

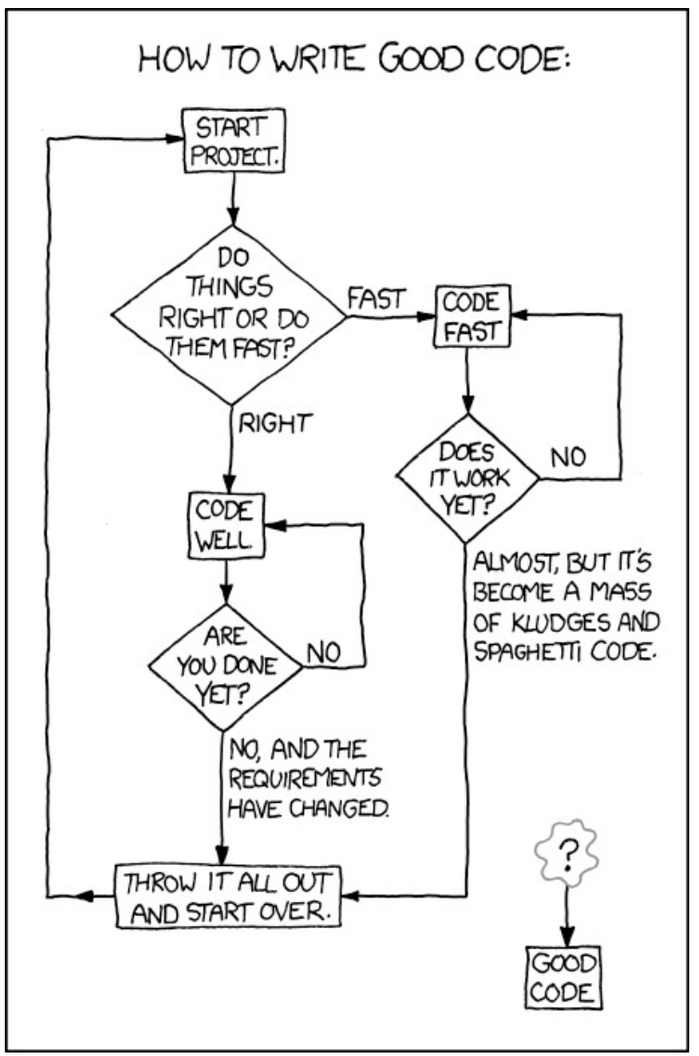
Code is written to:

- Be compiled, deployed, and executed by users in order to serve their need
- **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 if it is not well-structured / well-written / well-documented (/well-understood)…

Review: Analyzability

Recall: Analyzability is the capability of the software product to be **diagnosed** for deficiencies or causes of failures, or for the parts to be modified to be **identified**



Actual "rules" vary, depend on:

- Company culture
- Preferences
- Most important: Agreement among developers

Clean Code

Robert Martin

- Small
  - Make functions/methods do 1-5 logical steps
  - If more, consider breaking out into another function

Clean Code

Robert Martin

- Small
  - Make functions/methods do 1-5 logical steps
  - If more, consider breaking out into another function
- Do One Thing
  - Do one thing, do it well, do it only! Keep focus!

Clean Code

Robert Martin

- Small
  - Make functions/methods do 1-5 logical steps
  - If more, consider breaking out into another function
- Do One Thing
  - Do one thing, do it well, do it only! Keep focus!
- One Level of Abstraction
  - Don’t break encapsulation, hide implementation details

Clean Code

Robert Martin

- Small
  - Make functions/methods do 1-5 logical steps
  - If more, consider breaking out into another function
- Do One Thing
  - Do one thing, do it well, do it only! Keep focus!
- One Level of Abstraction
  - Don’t break encapsulation, hide implementation details
- Use Descriptive Names
  - Describe the one thing it does! Do not describe anything else

Clean Code

Robert Martin

- Small
  - Make functions/methods do 1-5 logical steps
  - If more, consider breaking out into another function
- Do One Thing
  - Do one thing, do it well, do it only! Keep focus!
- One Level of Abstraction
  - Don’t break encapsulation, hide implementation details
- Use Descriptive Names
  - Describe the one thing it does! Do not describe anything else
- Keep Number of Arguments low
  - Generally <= 3 arguments
  - If you need more, your method likely does **more than one thing!**

Clean Code

Robert Martin

- Avoid Flag Arguments
  - produceWebPage(true, true, false); ???
  - Indication that a method is doing **more than one thing**

Clean Code

Robert Martin

- Avoid Flag Arguments
  - produceWebPage(true, true, false); ???
  - Indication that a method is doing **more than one thing**

- Have No Side Effects
  - Do not do hidden things / have hidden state changes
    - Can fool the client and will hide weird bugs
  - ex: modifying an object passed as argument
    - If a method does this, the **descriptive name** should reflect it!

Clean Code

Robert Martin

- Command Query Separation
  - Getters and Setters / Accessors and Mutators
  - Query: no state change Return a value
  - Command: no return value Change the state

Clean Code

Robert Martin

- Command Query Separation
  - Getters and Setters / Accessors and Mutators
  - Query: no state change Return a value
  - Command: no return value Change the state
- Prefer Exceptions to Error Codes
  - Not ‘int addPayment(int amount)’ that returns error code 17 if it is an illegal coin
    - Unless you have modules that cannot propagate exceptions (e.g., networking)

Clean Code

Robert Martin

- Command Query Separation
  - Getters and Setters / Accessors and Mutators
  - Query: no state change Return a value
  - Command: no return value Change the state
- Prefer Exceptions to Error Codes
  - Not ‘int addPayment(int amount)’ that returns error code 17 if it is an illegal coin
    - Unless you have modules that cannot propagate exceptions (e.g., networking)
- Don’t Repeat Yourself
  - Avoid Duplicated Code
  - Recall: avoid **multiple maintenance problem**

More Rules

Henrik Christensen

- No arguments in method/class names
  - A symptom of duplicated code
  - addOneToX(int x); addTwoToX(int x); addThreeToX(int x); ???
  - Exception: test case methods

More Rules

Henrik Christensen

- No arguments in method/class names
  - A symptom of duplicated code
  - addOneToX(int x); addTwoToX(int x); addThreeToX(int x); ???
  - Exception: test case methods
- Do the same thing the same way
  - Analyzability

More Rules

Henrik Christensen

- No arguments in method/class names
  - A symptom of duplicated code
  - addOneToX(int x); addTwoToX(int x); addThreeToX(int x); ???
  - Exception: test case methods
- Do the same thing the same way
  - Analyzability
- Name Boolean sub-expressions

```
// 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 ) {
```

VS

```
boolean isOwnUnit = getUnitAt(from).getOwner() == getPlayerInTurn();
if (! isOwnUnit) return false;

boolean hasMovesLeft = getUnitAt(from).getMoveCount() >= 1;
if (! hasMovesLeft) return false;

boolean moveDistanceIsLessThanOne =
        Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
        Math.abs(from.getRow() - to.getRow()) <= 1;
if (! moveDistanceIsLessThanOne) return false;
```

More Rules

Henrik Christensen

- No arguments in method/class names
  - A symptom of duplicated code
  - addOneToX(int x); addTwoToX(int x); addThreeToX(int x); ???
  - Exception: test case methods
- Do the same thing the same way
  - Analyzability
- Name Boolean sub-expressions
- Avoid lots of nesting
  - Flatten by bailing out as soon as an answer can be computed

```
boolean isOwnUnit = getUnitAt(from).getOwner() == getPlayerInTurn();
if (! isOwnUnit) return false;

boolean hasMovesLeft = getUnitAt(from).getMoveCount() >= 1;
if (! hasMovesLeft) return false;

boolean moveDistanceIsLessThanOne =
        Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
        Math.abs(from.getRow() - to.getRow()) <= 1;
if (! moveDistanceIsLessThanOne) return false;
```

More Rules

Kent Beck

Make it **work**, then make it clean.

Refactoring: Improving the Design of Existing Code

Martin Fowler, Kent Beck

Code Smells

"Code smell" is a term for a surface indication that usually corresponds to a deeper problem in the system (implied that smell is bad).

Refactoring: Improving the Design of Existing Code
Martin Fowler, Kent Beck

Code Smells

"Code smell" is a term for a surface indication that usually corresponds to a deeper problem in the system (implied that smell is bad).

- Find areas of the code that can be improved
- Identify where refactoring is needed
- Address problems **early** before they have larger effects

Refactoring: Improving the Design of Existing Code
Martin Fowler, Kent Beck

Code Smells

"Code smell" is a term for a surface indication that usually corresponds to a deeper problem in the system (implied that smell is bad).

- Find areas of the code that can be improved
- Identify where refactoring is needed
- Address problems **early** before they have larger effects

Presence of a "smell" does not always mean there is a problem!

Refactoring: Improving the Design of Existing Code
Martin Fowler, Kent Beck

Code Smells

- Mysterious Name
- Duplicated Code
- Long Function
- Long Parameter List
- Global Data
- Mutable Data
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Repeated Switches
- Loops
- Lazy Element
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Insider Trading
- Large Class
- Alternative Classes with Different Interfaces
- Data Class
- Refused Bequest
- Comments

Code Smells

Martin Fowler, Kent Beck

Global Data

- When data can be modified from anywhere
  - Dangerous, hard to find which modification is causing a bug
- Encapsulate variables (in a function / class)
- Global **constants** are generally okay

Code Smells

Martin Fowler, Kent Beck

Mutable Data

- Changes to data can lead to unexpected consequences (bugs)
- Encapsulate data such that updates occur through functions
- Assign modified data to new variables rather than reassigning to the same variable

Code Smells

Martin Fowler, Kent Beck

Data Clumps

- If bunches of data tend to always show up together, consider making a class

Code Smells

Martin Fowler, Kent Beck

Insider Trading

- Passing data around increases coupling
- Minimize passing data, and make it obvious when needed
  - Move functions/fields to reduce passing data, or create an intermediary module
  - Replace a subclass with a delegate

Code Smells

Martin Fowler, Kent Beck

Data Class

- When classes have fields, getting and setting methods, and nothing else
- Depending on where the getting/setting methods are being used by other classes, move more behavior from those classes into the data class

Code Smells

Martin Fowler, Kent Beck

Divergent Change

- When one module is often changed in different ways for different reasons (e.g., update for interaction with a new database and also update for a new display)
- Fix by separating contexts
  - Get data from a database, store in data structure to transfer to another context that handles formatting for the display

Code Smells

Martin Fowler, Kent Beck

Shotgun Surgery

- When every time you make a change you have to make a lot of small edits to different classes
  - Opposite of divergent change
  - Hard to find, easy to miss a necessary change
- Move code such that changes occur in a single module (function, class)
- May end up with “Long Function/Large Class” temporarily during refactoring

Code Smells

Martin Fowler, Kent Beck

Speculative Granularity

- Anticipating features can lead to unnecessary complexity (unnecessary delegation, unused parameters)
- Make changes as needed, not in anticipation of need

Code Smells

Martin Fowler, Kent Beck

Alternative Classes with Different Interfaces

- Use interfaces for consistent interaction with a group of objects

Code Smells

Martin Fowler, Kent Beck

Comments

- Comments are generally good!
- But, don’t use comments to compensate for unreadable code
- Increase analyzability, so that fewer comments are needed
- Use comments to communicate ”why?” (can include if you are unsure if this is the best implementation, or if more could be done!)

Cohesion and Coupling

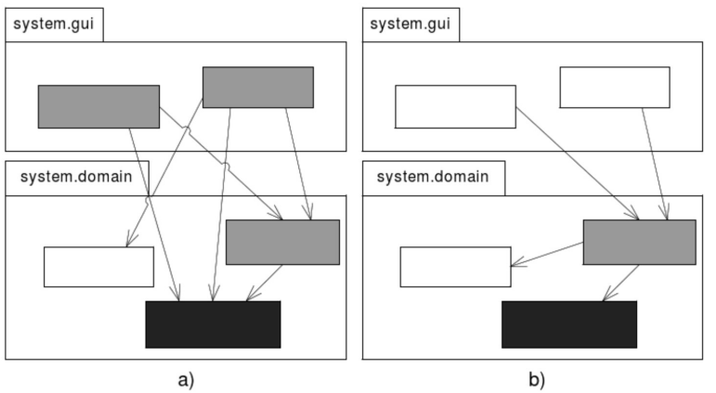


Figure 10.1: Tight (a) and low (b) coupling.

Recall:

Coupling is a measure of how strongly dependent one software unit is on other units

Cohesion is ta measure of how strongly related and focused the responsibilities and behaviors of a software unit are

Assign responsibility so coupling is low, and cohesion is high

→ **Lower the cost of change**

Cohesion and Coupling

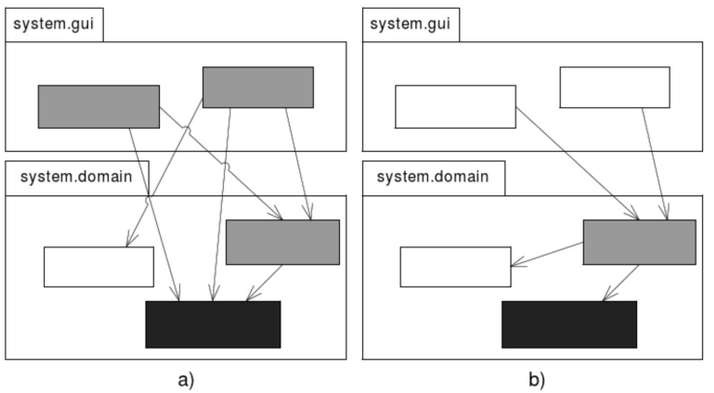


Figure 10.1: Tight (a) and low (b) coupling.

How do we design for low coupling?

Law of Demeter: Do not collaborate with indirect objects

"Don't talk to strangers"

→ Units should have only **limited knowledge** of other units

→ Limit assumptions about structure of other units

Example:

```
order.getItem(3).getPrice().addTax()
```

should be replaced by

```
order.addTaxToItem(3);
```

Cohesion and Coupling

How do we design for high cohesion?

→ More about roles and responsibility in CH 15

Quality of Test Code

TDD produces two code bases

- The **production** code
- The **test** code

Code quality is important in both!

- You want to be able to read and understand test code
- **Test code exception to "clean code": Arguments in Argument Lists**
  - "moveRedArcherToPosition45()"

Quality of Test Code

Recall: Avoid tying test code to implementation details

Exercise: Review this test code:

```
@Test
public void blueShouldHaveLegionUnitAtPositionThreeTwo(){
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getOwner(), is(Player.BLUE));
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getTypeString(), is(GameConstants.LEGION));
}
```

Quality of Test Code

Recall: Avoid tying test code to implementation details

Exercise: Review this test code:

```
@Test
public void blueShouldHaveLegionUnitAtPositionThreeTwo(){
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getOwner(), is(Player.BLUE));
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getTypeString(), is(GameConstants.LEGION));
}
```

What happens if I need to change the Unit data structure???

How should we rewrite this test?

Quality of Test Code

Recall: Avoid tying test code to implementation details

Exercise: Review this test code:

```
@Test
public void blueShouldHaveLegionUnitAtPositionThreeTwo(){
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getOwner(), is(Player.BLUE));
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getTypeString(), is(GameConstants.LEGION));
}
```

What happens if I need to change the Unit data structure???

How should we rewrite this test?

→ Use Game and Unit interface methods: getUnitAt(…).getOwner()

Quality of Test Code

Only test implementation details if:

- They cannot just as easily be test-driven by using outside interfaces
- They are highly complex by themselves and warrant a TDD approach because of this complexity
- There is low probability of changing the data structure in the future

Quality of Test Code

Test case: We cannot stack two friendly units

Problem: How to make the **fixture -** the environment/setup of objects in preparation for the actual test?

- Need to have two RED archers on adjacent tiles

Solutions?

Quality of Test Code

Test case: We cannot stack two friendly units

Problem: How to make the **fixture -** the environment/setup of objects in preparation for the actual test?

- Need to have two RED archers on adjacent tiles

Solutions?

1) Age the world until two red units exist, move one next to the other
2) gameImpl.setInternalMatrix(4, 4, new UnitImpl(RED, ARCHER))

Exercise: Which is the solution with most **stability**?

Definition: **Stability (ISO 9126)**

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

Quality of Test Code

1) Age the world until two red units exist, move one next to the other
   - More cumbersome, requires a lot of steps taken in the proper order
     → **Less evident test**
   - But is a more **stable** test case
     → Game's internal data structures are **encapsulated**

2) gameImpl.setInternalMatrix(4, 4, new UnitImpl(RED, ARCHER))
   - **Less stable** because if I change implementations inside Game then **test cases will break**
     → Defeats the purpose of encapsulation

Quality of Test Code

A third option? Test stubs

→ Inject **delegates** that configure the world to a particular setup

- Use a **world setup strategy** that configures the world to have two red archers next to each other
- “Dependency injection”

More later…

Quality of Test Code

TDD means **covering the requirements!**

Suppose that this single test case:

```
assertThat(game.moveUnit((4,3),(5,5)),is(true));
```

was used to cover the full moveUnit

But, moveUnit is pretty complex:

- Non passable terrain
- No stacking
- Attacking enemies
- City conquest

Quality of Test Code

Write the test to ensure:

- A unit can move from one plain (5,5) to another empty plain (6,6)
  - Write production code
  - Test should confirm that the unit is no longer on (5,5) and is on (6,6)
- A unit cannot move into the ocean
  - Write production code
  - Test should confirm that the unit is still on ‘from’ and not on ‘to’
- A unit cannot stack onto a friendly unit
  - Write production code
  - Test should confirm that the unit is still on ‘from’ and not on ‘to’

Result: The test code tests **all requirements**

→ And can keep testing them throughout the system’s lifetime!

When to subclass?

Avoid inheritance for **non-behavioral variability**

→ Class MountainTile implements Tile ?

→ Only use subclasses if you get a distinct **behavioral** advantage

Parameterization can be changed at run-time

Regarding Units

Unit types are **behaviorally distinct**!

So: Polymorphic design?

Regarding Units

Unit types are **behaviorally distinct**!

So: Polymorphic design?

→ Will work with some hacks

- e.g., How does a settler unit remove itself as part of its action?

Regarding Units

Unit types are **behaviorally distinct**!

So: Polymorphic design?

→ Will work with some hacks

- e.g., How does a settler unit remove itself as part of its action?

How about compositional design?

- Make a **StandardUnit/UnitImpl**, and delegate actions to a **Strategy**
- Suggestion: StandardUnit/UnitImpl has **general** behavior parameters
  - 'isMoveable' boolean, shared by all, not just by archers

  → Always true for Settlers and Legions, more flexible for adding future behaviors
- The 'type switch' usually pops up anyway

```
if (theUnit instanceof ArcherUnit) {
        ArcherUnit su = (ArcherUnit) theUnit;
        su.setMoveable(false);
}
```

Summary

Code quality guidelines are “rules of thumb” for analyzability

Remember: Actual “rules” vary, depend on:

- Company culture
- Preferences
- Most important: Agreement among developers

Livable Code

Sarah Mei

https://twitter.com/i/events/843392359903649792
https://www.youtube.com/watch?app=desktop&v=UDp6eDCi1Vo

"Staged houses are beautiful… but you can't actually live in a staged house. They leave out lots of things that make a space more cluttered, but also make it work…

On the other side, we can't live easily in a house that is so overrun with clutter and stuff that it's mostly unusable. This is hoarding.

Most of our codebases are hoarded. But most of our object-oriented advice is a staged house. What we need is something in the middle…

enough clutter to be comfortable, with enough space to be flexible.

Instead of 'does this bring me joy?' as in konmari, I think the right question for code is 'do I understand this?'

…understanding is personal. It depends on who's on your team."