COM 1001

Introduction to Software Engineering

Professor Phil McMinn

Re-Engineering Smelly Code through

Refactoring

What is “Refactoring”?

Refactoring is the activity of
improving existing code
without changing its external behaviour.

Refactoring is the activity of
improving existing code
without changing its external behaviour.

What do you mean, improve? And why?

Improving its:
Readability
Structure

Why?
Well structured, readable code is easy for humans to understand, maintain and extend. Leading to higher quality software with few bugs.

Code tends to fall into a state of “bad design” or poor structure over a period of time

- Sometimes the way to implement something is unclear, and the initial attempt is not the best way to do it

- New features are added and other incremental changes are made over time, without taking account of the impact on overall structure

- Such code is said to be **smelly**. There are a number of particular **bad smells**…

1. Poor Naming Choices

Bad names for variables, methods, classes, and modules include:

- single letters: e.g. `a`, `b`, `c` (exception: temporary variables, loop iteration variables, or mathematical code)
- Nondescript names like `mickey`, `minnie`, `donald`.

When giving something a name, think:

- What does the thing **do**? What is it **for**? How could my name best describe its **purpose**?
- **Good naming means that another programmer can understand your code quickly and what it does**, without the need to write excessive amounts of comments to document your code.

2. Duplicated Code

These two code segments are effectively the same, using different variables

```ruby
def add_average_width_and_height(widths, heights)
  # get the average of the widths
  sum_widths = 0
  widths.each do |i|
    sum_widths += i
  end
  average_widths = sum_widths / widths.length

  # get the average of the heights
  sum_heights = 0
  heights.each do |i|
    sum_heights += i
  end
  average_heights = sum_heights / heights.length

  # add and return
  average_widths + average_heights
end
```

refactoring/duplicated_code.rb

Sometimes it's tempting to copy and paste code segments around without properly thinking about good code design.

Solution: We could refactor by extracting the code for computing the average into its own method, and calling that method twice

2. Duplicated Code

```ruby
def add_average_width_and_height(widths, heights)
  # get the average of the widths
  sum_widths = 0
  widths.each do |i|
    sum_widths += i
  end
  average_widths = sum_widths / widths.length

  # get the average of the heights
  sum_heights = 0
  heights.each do |i|
    sum_heights += i
  end
  average_heights = sum_heights / heights.length

  # add and return
  average_widths + average_heights
end
```

refactoring/duplicated_code.rb

```ruby
def add_average_width_and_height(widths, heights)
  average(widths) + average(heights)
end

def average(x)
  sum_x = 0
  x.each do |i|
    sum_x += i
  end
  sum_x / x.length
end
```

refactoring/duplicated_code_refactored.rb

The refactored code does the same thing, but with no duplicated code, fewer lines of code.

2. Duplicated Code

```ruby
def add_average_width_and_height(widths, heights)
  average(widths) + average(heights)
end

def average(x)
  sum_x = 0
  x.each do |i|
    sum_x += i
  end
  sum_x / x.length
end
```

refactoring/duplicated_code_refactored.rb

The refactored code does the same thing, but with no duplicated code, fewer lines of code.

It's also much easier to understand.

If we need to fix a bug in the averaging code, we also only need to fix it in one place.

Remember the mantra:
Don't Repeat Yourself! (DRY)

Staying Safe

Of course, since we're not changing behaviour, and if we have tests (and if not, why not…) we can use the tests to check we have made the change correctly.

The same tests should pass whichever version of the code is used.

If there is a failure we have introduced a so-called regression. Testing after changes like this is called **regression testing**.

```ruby
require "rspec"

# uncomment one line or the other to check the two are equivalent
require_relative "duplicated_code"
# require_relative "duplicated_code_refactored"

describe "#add_average_width_and_height" do
  context "when given two arrays" do
    it "returns their averages added together" do
      expect(add_average_width_and_height([2, 3, 4], [20, 30, 40])).to eq(33)
    end
  end
end
```

refactoring/duplicated_code_spec.rb

3. Long Methods

Problem: A method gets very long, making it hard to understand.

This can happen with tests too!

Consider this test I originally wrote for the football players app.

Rubocop considers anything over 10 lines as too long!

Solution: extract methods to shorten the original method

```
it "allows a player record to be changed" do
  # add a player
  visit "/add"
  fill_in "first_name", with: "George"
  fill_in "surname", with: "Test"
  fill_in "gender", with: "M"
  fill_in "club", with: "Mantester Utd"
  fill_in "country", with: "Northern RSpec"
  fill_in "position", with: "Midfield"
  fill_in "date_of_birth", with: "1946-05-22"
  click_button "Submit"

  # edit a player
  visit "/edit?id=1"
  fill_in "first_name", with: "Zinedine"
  click_button "Submit"
  expect(page).to have_content "Zinedine Test"

  # reset the database
  DB.from("players").delete
end
```

3. Long Methods

```
it "allows a player record to be changed" do
  # add a player
  visit "/add"
  fill_in "first_name", with: "George"
  fill_in "surname", with: "Test"
  fill_in "gender", with: "M"
  fill_in "club", with: "Mantester Utd"
  fill_in "country", with: "Northern RSpec"
  fill_in "position", with: "Midfield"
  fill_in "date_of_birth", with: "1946-05-22"
  click_button "Submit"

  # edit a player
  visit "/edit?id=1"
  fill_in "first_name", with: "Zinedine"
  click_button "Submit"
  expect(page).to have_content "Zinedine Test"

  # reset the database
  DB.from("players").delete
end
```

```
it "allows a player record to be changed" do
  add_test_player

  visit "/edit?id=1"
  fill_in "first_name", with: "Zinedine"
  click_button "Submit"
  expect(page).to have_content "Zinedine Test"

  clear_database
end
```

Recall that I added the `add_test_player` and `clear_database` methods to `spec_helper.rb`.

This code was also duplicated across different tests. So two birds are killed in one stone!

4. Large Classes/Modules/Controllers

Sometimes our classes/modules/controllers just get too big – they have too many methods, routes, etc. meaning a **lot of scrolling** around the file.

This makes them hard to understand.

Usually they are **doing too much**.

Solution: Split into smaller, more contained (cohesive and coupled) files

Think: Which methods/routes should be grouped together? Does this grouping make logical sense? What alternative groupings are possible – do they make more sense? The rationale for the grouping is probably a good place to start for the name for the new class/module/controller and its file.

5. Speculative Generality

Speculative Generality is characterised by the kind of thought that goes…

“Oh, I think we might need the ability to do this one day…”

… and we add all sorts of code to handle special cases that we don’t need right now.

Research suggests that “bets” on future needed functionality rarely pay off.

This leads to code that isn’t required, and makes the code base harder to understand and maintain.

The old saying is **YAGNI: You Ain’t Gonna Need It**.

Speculative Generality is identified by code that isn’t doing much!
Remove it and/or simplify.

6. Code Comments

I'm not telling you not to write comments.

Comments aren't a bad smell, they're a *sweet smell*.

But often they're used as a **deodorant**: when you see lots of comments it's often because the code is bad – it's not self-evident as to what it does and needs lots of explaining.

Solution: Use more descriptive names. Replace code chunks with methods with names that explain what they do – the types of things we've already discussed here. Once you've done that, ask yourself, do you still need those comments?

Note How Some of Our Examples Removed The Comments Because the Code Became Self Explanatory

Code comments removed in the process of de-duplicating code:

```ruby
def add_average_width_and_height(widths, heights)
  # get the average of the widths
  sum_widths = 0
  widths.each do |i|
    sum_widths += i
  end
  average_widths = sum_widths / widths.length

  # get the average of the heights
  sum_heights = 0
  heights.each do |i|
    sum_heights += i
  end
  average_heights = sum_heights / heights.length

  # add and return
  average_widths + average_heights
end
```

```ruby
def add_average_width_and_height(widths, heights)
  average(widths) + average(heights)
end

def average(x)
  sum_x = 0
  x.each do |i|
    sum_x += i
  end
  sum_x / x.length
end
```

No need for code comments – the code is now pretty self-explanatory.

Note How Some of Our Examples Removed The Comments Because the Code Became Self Explanatory

Code comments removed in the process of reducing a long code block:

```ruby
it "allows a player record to be changed" do
  # add a player
  visit "/add"
  fill_in "first_name", with: "George"
  fill_in "surname", with: "Test"
  fill_in "gender", with: "M"
  fill_in "club", with: "Mantester Utd"
  fill_in "country", with: "Northern RSpec"
  fill_in "position", with: "Midfield"
  fill_in "date_of_birth", with: "1946-05-22"
  click_button "Submit"

  # edit a player
  visit "/edit?id=1"
  fill_in "first_name", with: "Zinedine"
  click_button "Submit"
  expect(page).to have_content "Zinedine Test"

  # reset the database
  DB.from("players").delete
end
```

```ruby
it "allows a player record to be changed" do
  add_test_player

  visit "/edit?id=1"
  fill_in "first_name", with: "Zinedine"
  click_button "Submit"
  expect(page).to have_content "Zinedine Test"

  clear_database
end
```

The names of the methods called in the code self-describe what's happening.

There's no need for the code comments anymore.

Refactoring Databases

… because software is more than just code!

How Can Databases Be Smelly?

1. Repeated Data

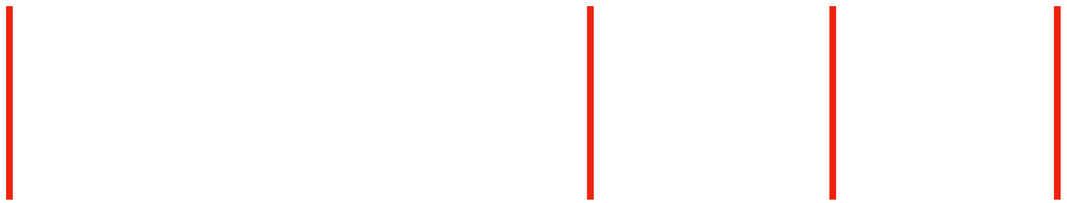
2. Multiple values for a column in one Row

3. Multiple “Entities” in a Table

1. Repeated Data

players

id	first_name	surname	gender	date_of_birth	country	position	club
1	Dominic	Calvert-Lewin	M	1997-03-16	England	Forward	Everton
2	Sam	Kerr	F	1993-09-10	Australia	Forward	Chelsea
3	Harry	Kane	M	1993-07-28	England	Forward	Tottenham
4	Rose	Lavelle	F	1995-05-14	USA	Midfielder	Manchester City
5	Son	Heung-min	M	1992-07-08	South Korea	Forward	Tottenham
6	Pernille	Harder	F	1992-11-15	Denmark	Forward	Chelsea
7	Bruno	Fernandes	M	1994-09-08	Portugal	Midfielder	Manchester
8	Hayley	Raso	F	1994-09-05	Austrailia	Midfielder	Everton
9	Kevin	De Bruyne	M	1991-06-28	Belgium	Midfielder	Manchester City
10	Vivianne	Miedema	F	1996-07-15	Netherlands	Forward	Arsenal
11	Michael	Keane	M	1993-01-11	England	Defender	Everton
12	Ellie	Roebuck	F	1999-09-23	England	Goalkeeper	Manchester City



These columns contains repeated values

1. Repeated Data

Why is this bad?

Increased storage requirements

Inefficient data retrieval

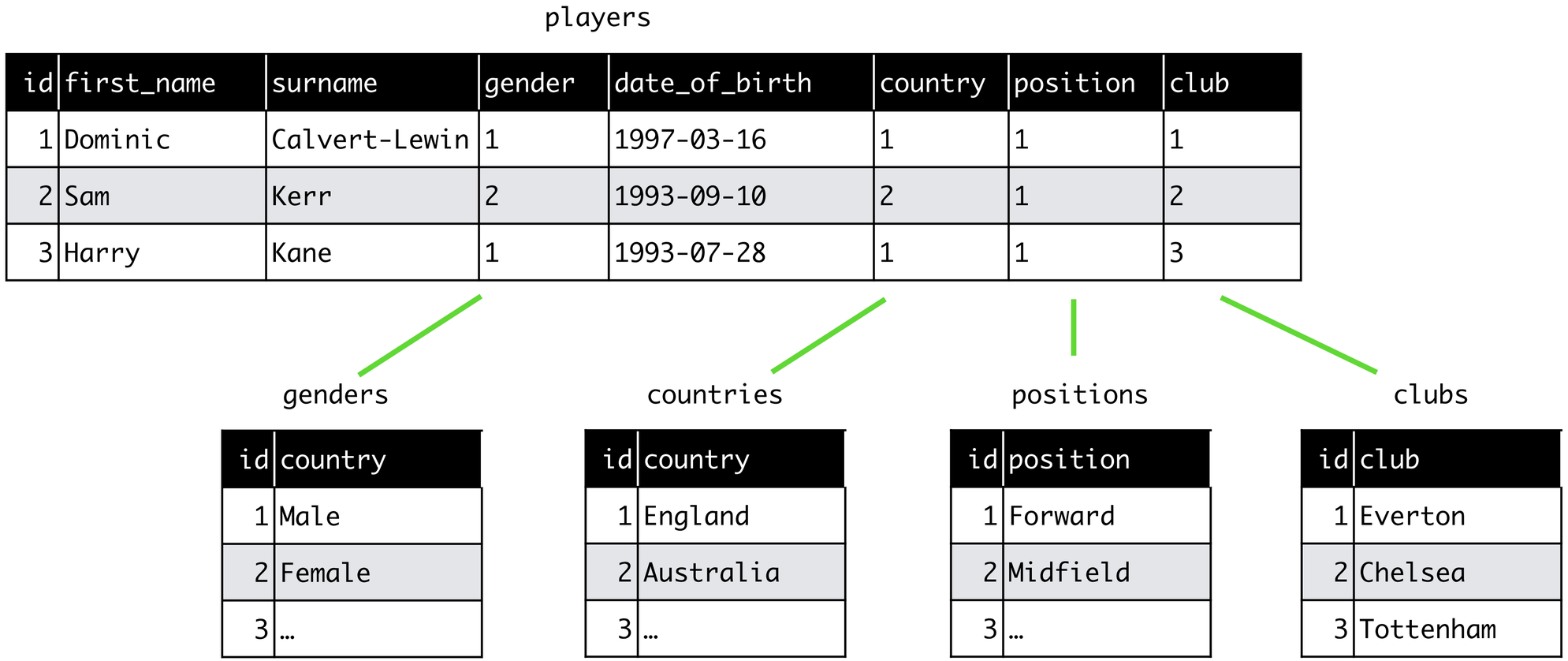
Typos and inconsistencies lead to problems:

- “`Evrton`” wouldn’t be picked up in a search for “`Everton`”
- “`Man. City`” / “`Man City`” / “`Manchester City`” all different ways to refer the same team. But without this knowledge, how do we know they’re the same?
- Also is it “`Tottenham`” or “`Tottenham Hotspur`”? Or just “`Spurs`”?

1. Repeated Data

Solution

Split the repeated data out into another table, with a canonical value, and an index that can be referenced by other tables.



We can make this formal in the database through the use of a **foreign key**

Relational databases can be used to enforce this relationship – that is, if a foreign key value does not match a value in the referenced table's column, an error is thrown when an `INSERT` is attempted.

However, SQLite does not enforce it. See https://sqlite.org/foreignkeys.html for more information.

```
CREATE TABLE players (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    surname TEXT,
    gender INTEGER,
    date_of_birth TEXT,
    country INTEGER,
    position INTEGER,
    club INTEGER,
    FOREIGN KEY(gender) REFERENCES genders(id),
    FOREIGN KEY(country) REFERENCES countries(id),
    FOREIGN KEY(position) REFERENCES positions(id),
    FOREIGN KEY(club) REFERENCES clubs(id)
);

CREATE TABLE genders (
    id INTEGER PRIMARY KEY,
    gender TEXT
);

CREATE TABLE countries (
    id INTEGER PRIMARY KEY,
    country TEXT
);

CREATE TABLE positions (
    id INTEGER PRIMARY KEY,
    country TEXT
);

CREATE TABLE clubs (
    id INTEGER PRIMARY KEY,
    club TEXT
);
```

Note the type change from `TEXT` to `INTEGER`. This is to match the respective `id` fields in the new tables

2. Multiple Values in a Column for a Row

What about players that play in more than one position?

players

id	first_name	surname	gender	date_of_birth	country	position	club
100	George	Test	…	1946-05-22	…	1, 2	…

… or worse:

id	first_name	surname	gender	date_of_birth	country	position	club
100	George	Test	…	1946-05-22	…	Winger,Striker	…

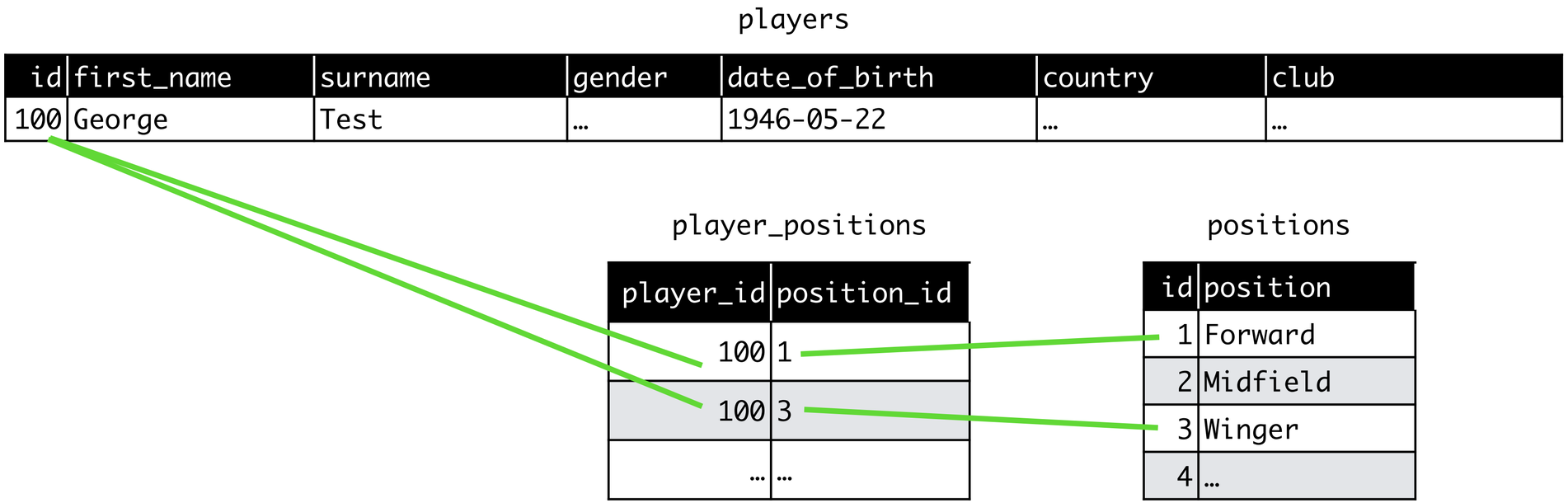
Why is this bad?

Breaks relational design – very hard to query on the multiple values without extremely complicated `SELECT` statements

2. Multiple Values in a Column for a Row

Solution

Use a linker table to create a one-to-many relationship.



Note the “compound” primary key. Neither value is unique, only the pair.

There are now two foreign keys, both of which reside in the linker table.

```
CREATE TABLE players (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    surname TEXT,
    gender INTEGER,
    date_of_birth TEXT,
    country INTEGER,
    club INTEGER,
    FOREIGN KEY(gender) REFERENCES genders(id),
    FOREIGN KEY(country) REFERENCES countries(id),
    FOREIGN KEY(club) REFERENCES clubs(id)
);

CREATE TABLE player_positions (
    player_id INTEGER,
    position_id INTEGER,
    PRIMARY KEY(player_id, position_id),
    FOREIGN KEY(player_id) REFERENCES players(id)
    FOREIGN KEY(position_id) REFERENCES positions(id)
);

CREATE TABLE positions (
    id INTEGER PRIMARY KEY,
    country TEXT
);
```

3. Multiple “Entities” in a Table

This table is about a person, but it also contains details about the company (that they work for).

But company information is not related to a person – it should be in a separate table. Person can then reference the company information via a foreign key.

Solution: split into two tables.

```
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    surname TEXT,
    email_address TEXT,
    company_name TEXT,
    company_address TEXT,
    company_tel_no TEXT
);
```

refactoring/person.sql

3. Multiple “Entities” in a Table

```sql
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    surname TEXT,
    email_address TEXT,
    company_name TEXT,
    company_address TEXT,
    company_tel_no TEXT
);
```

refactoring/person.sql

```sql
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    surname TEXT,
    email_address TEXT,
    company INTEGER,
    FOREIGN KEY(company) REFERENCES company(id)
);

CREATE TABLE company (
    id INTEGER PRIMARY KEY,
    company_name TEXT,
    company_address TEXT,
    company_tel_no TEXT
);
```

refactoring/person_refactored.sql

Normalisation

Relational Database Normalisation is a set of rules for database design that helps avoid bad smells in database design.

Typically, however, a developer will be aware of good/bad database design issues without referencing these formal rules all of the time.

You'll encounter normal forms in *COM2008 Systems Design and Security*.