100 baby steps away from spaghetti code

An emerging design workshop

Commandline Videostore

- Original Java version by Rick Janda: https://github.com/rickjanda/commandline-videostore
- Ported to C++ https://github.com/arnemertz/commandline-videostore-cpp

What is spaghetti code?

- **Low cohesion**: Multiple concerns or responsibilities are entangled with each other in single methods, classes or packages
- **High coupling**: Classes and methods depend on many other classes, state and logic are separated
- **Missing abstractions**: No classes representing domain concepts or technical responsibilities

Emerging design - basic idea

Use refactoring to bring state and logic together in meaningful new classes

1. Extract some logic into methods
2. Wrap state with a new class
3. Move extracted methods into the new class (steps 1 and 2 are interchangeable)

Emerging design process

1. *Separate responsibilities within a function*
2. **Extract private functions**
3. *Remove direct access to fields other than those of the new class*
4. **Wrap target state with domain specific class**
5. **Move methods to the target class as public**
6. *Simplify parameters to remove unwanted dependencies*

What does baby step refactoring mean?

- Split large refactorings into many small steps
- The code is compilable and the tests are green after every step
- If you can not make the code compilable and the tests green within 30 seconds, roll back to last green state

Why?

- Avoids merging nightmare
- Avoids debugging hell
- Avoids analysis paralysis
- Allows us to fix bugs or start other work at any time
- Easier to deal with interruptions
- Allows us to tackle technical debt step-wise

Taking small steps makes you fall over less often!

Preparation

Clone the repository

https://github.com/arnemertz/commandline-videostore-cpp

Import the project to CLion

Open CMakeLists.txt as project

Compile and run the Test

Preparation

Demo!

Find the smells

Identify domain concepts

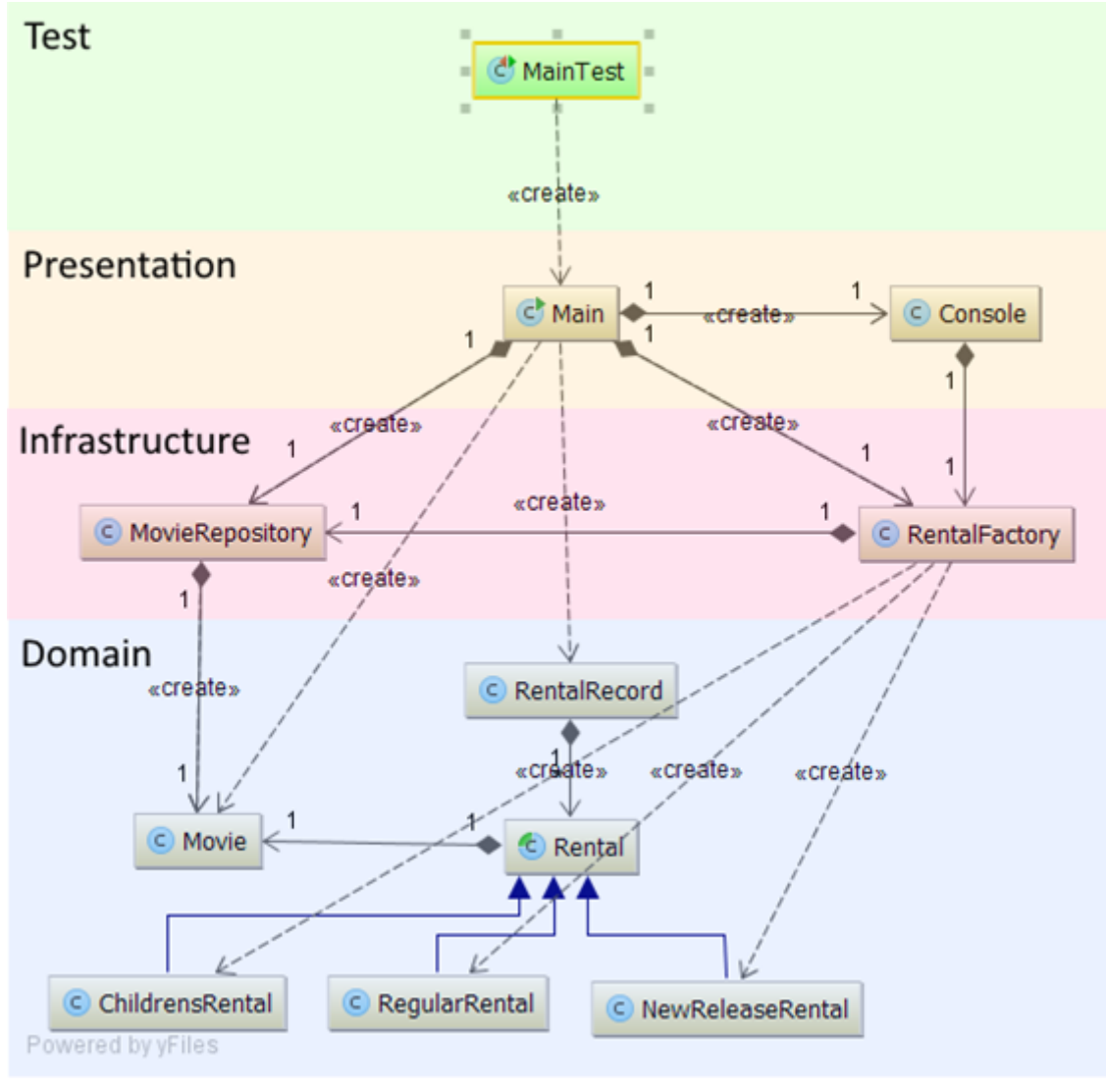
What we are going to do:

In (up to) 10 rounds:

- Extract domain concepts and logic into domain classes
- Separate infrastructure logic into infrastructure classes
- Separate presentation logic with a kind of MVC pattern

Focus on baby step refactoring and emerging design techniques

For timing reasons we skip low level code polishing and long discussions of other design options.


Test
MainTest
«create»
Presentation
Main
Console
Infrastructure
MovieRepository
RentalFactory
Domain
RentalRecord
Movie
Rental
ChildrensRental
RegularRental
NewReleaseRental
1
«create»
Powered by yFiles

Good to know:

If you get "lost" or mess up your code:
For each exercise there is a branch you can check out
(E0..E8b)

CLion keyboard shortcuts	**PC**	**Mac**
Refactor submenu	Ctrl+Alt+Shift+T	^T
Rename	Shift+F6	⇧+F6
Extract **M**ethod/function	Ctrl+Alt+**M**	⌘+⌥+**M**
Extract **V**ariable	Ctrl+Alt+**V**	⌘+⌥+**V**
Change signature	Ctrl+F6	⌘+F6
I**n**line method/variable	Ctrl+Alt+**N**	⌘+⌥+**N**

E0: Extract helper function to split strings

- In order to remove code duplication, create the function

```
std::vector<std::string> split(std::string const& str, char delim
```

- Extract the two places where strings are split into functions
- Refactor until they are identical
- Reduce to a single function

Technique: Parallel Change

1. Create new structure in parallel to existing one
2. Step-wise add write accesses to new structure next to each write access to existing structure
3. Step-wise switch read accesses from old to new structure
4. Remove old structure
5. *Clean up*

Run tests after each baby step & commit frequently

Technique: Narrow Change

Reduce amount of similar manual changes by means of the duplication detection in some automated refactorings:

- Introduce indirection by automated refactoring
- Extract Method; or
- Extract local variable; or
- Introduce factory method (for constructors)
- Manual change in one place → **Run your tests**
- Inline indirection again

E1: Extract domain concept - introduce `Movie` struct

- In order to remove the Primitive Obsession smell

Primitive Obsession: using primitive data types instead of classes to represent domain ideas.

- `Movie` is just a bunch of data, i.e. a struct suffices.
- Use the parallel change technique
- No logic to move into `Movie`

E2: Extract domain concept - introduce `Rental` class

- In order to remove the Primitive Obsession smell
- `Rental` should get the following constructor:

```
Rental(Movie const& movie, int daysRented);
```

- Extract and move the following logic into `Rental`:

```
double getAmount() const;
int getFrequentRenterPoints() const;
string const& getMovieName() const;
```

- Parsing logic should be kept in the `run` function

Technique: Loop Splitting

If the body of a loop contains several responsibilities that we want to separate we have to duplicate the loop and keep only one responsibility in every loop body.

- Create a container in the first loop and iterate over it in the following loops
- This approach of duplicating the control structure is also feasible for other control structures with several responsibilities in the body.

E3: Extract infrastructure logic - introduce `MovieRepository`

- Extract file reading and parsing logic for `Movie` into a new `MovieRepository` class
- The printout of the movie list must *not* be moved into the `MovieRepository`
- The `MovieRepository` should get two methods:

```
std::vector<Movie> getAll() const;
const Movie& getByKey(int key) const;
```

- The `MovieRepository` should read from the file in its constructor

E4: Extract infrastructure logic - introduce `RentalFactory`

E5: Split the while loop

- Into 4 loops with a single responsibility each
- Extract the following methods:

```
std::vector<Rental> inputRentals(std::istream& in, RentalFactory& r
int getFrequentRenterPoints(std::vector<Rental> const& rentals);
double getTotalAmount(std::vector<Rental> const& rentals);
```

E6: Extract Domain concept - Create `RentalRecord` class

E7: Extract Presentation Logic - Introduce Console class

- That wraps `in` and `out`
- The `run` function should remain as a controller with a single level of abstraction (SLoA) and all details moved out to other functions and classes.

Technique: Duplicate and Reduce

A technique to split methods containing multiple responsibilities needed by different callers:

1. Duplicate the method by:
   - Copy and paste (plus rename); or
   - Pull member down to all subclasses; or
   - Inline method implementation at caller side
2. Reduce both copies to what is needed for the specific caller or in the given scope

Technique: Introduce Polymorphism

1. Encapsulate constructor with static factory method
2. Create empty subclasses with the same constructor signatures like the parent class, e.g. `using Base::Base`.
3. Instantiate subclasses instead of parent class in static factory method
   - with switch case or if/else blocks switching on the according type parameter

->

Technique: Introduce Polymorphism (contd.)

4. Declare parent class abstract
5. Use duplicate and reduce to move logic into subclasses and simplify them (push members down or copy/paste for duplicate)

E8: Introduce polymorphism for Rental

Based on `movie.category`

E8a: `RentalFactory::createRental(const std::string& input)` has to be changed to return a `unique_ptr<Rental>`

E8b:

- Create `RegularRental`, `ChildrensRental` and `NewReleaseRental` classes
- Remove the simulated polymorphic behavior (if/else over category)
- First: create and instantiate skeletons for subclasses
- Second: Use Duplicate&Reduce (use refactoring "push down")

What techniques we have learned:

- Parallel change
- Narrow change
- Loop splitting
- Duplicate and reduce
- Separate responsibilities and concerns
- Emerge compositions and inheritance
- Code generation, refactorings, IDE suggestions to improve productivity.

Hint: Learn your IDE shortcuts
(The mouse is your enemy ;-)