Testing database applications with QuickCheck

— Tutorial —

Laura M. Castro

Universidade da Coruña (Spain)

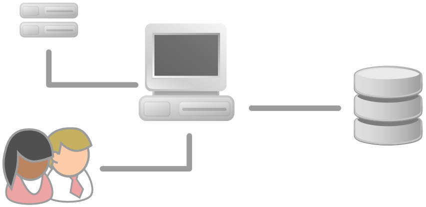
Stockholm, 15th November 2010

Outline

What is a database application?

A **database** or **data-intensive application** is a software system which:

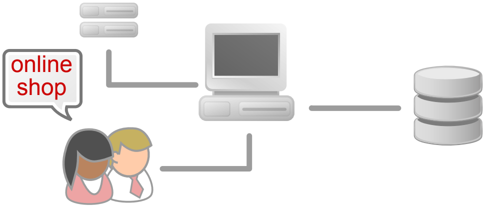
- makes intensive use of great amounts of data,
- relies on external storage sources for persistence (e.g., a database).



What is a database application?

A **database** or **data-intensive application** is a software system which:

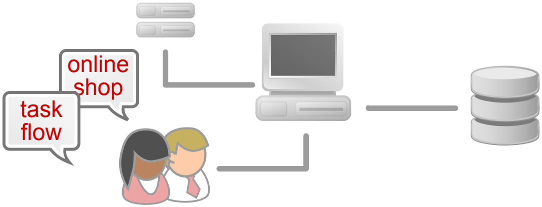
- makes intensive use of great amounts of data,
- relies on external storage sources for persistence (e.g., a database).



What is a database application?

A **database** or **data-intensive application** is a software system which:

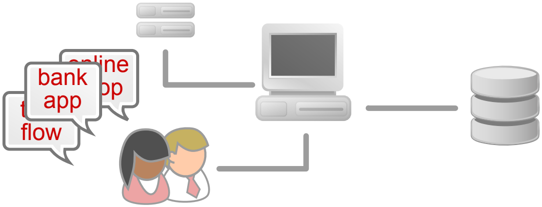
- makes intensive use of great amounts of data,
- relies on external storage sources for persistence (e.g., a database).



What is a database application?

A **database** or **data-intensive application** is a software system which:

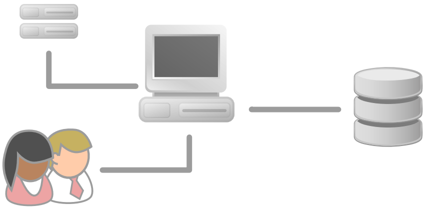
- makes intensive use of great amounts of data,
- relies on external storage sources for persistence (e.g., a database).



Why do DB applications require special testing?

Database or data-intensive applications are software systems which:

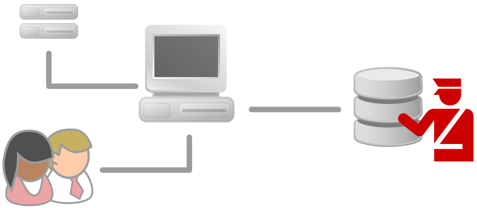
- impose **complex constraints** on the data they handle,
- their **correct operation** depends on their enforcement.



Why do DB applications require special testing?

Database or data-intensive applications are software systems which:

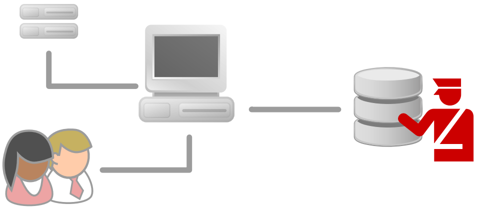
- impose **complex constraints** on the data they handle,
- their **correct operation** depends on their enforcement.



Why do DB applications require special testing?

Database or data-intensive applications are software systems which:

- impose **complex constraints** on the data they handle,
- their **correct operation** depends on their enforcement.



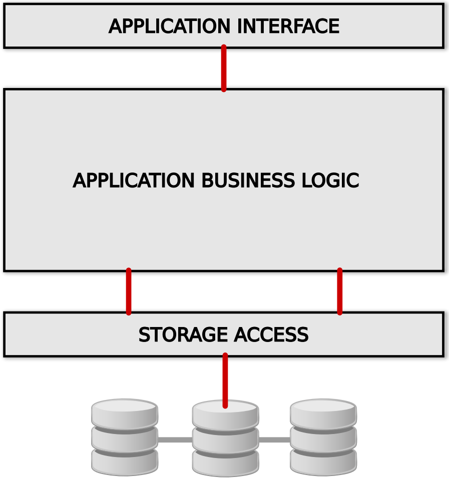
These constraints are usually referred to as **business rules**.

Business Rules
What are they?

- “**Statements** that **define** or constrain some aspect of a business (. . .) intended to assert business **structure or** to **control** or influence behavior”
  (B.R. Group, *Defining Business Rules – What are they really?*)
- “Definitions of **how** the business should be **carried out** and constraints on the business” (I. Sommerville, *Software Engineering*)
- “Software is the **realization** of business rules”
  (R.S. Pressman, *Software Engineering – A practitioner’s approach*)

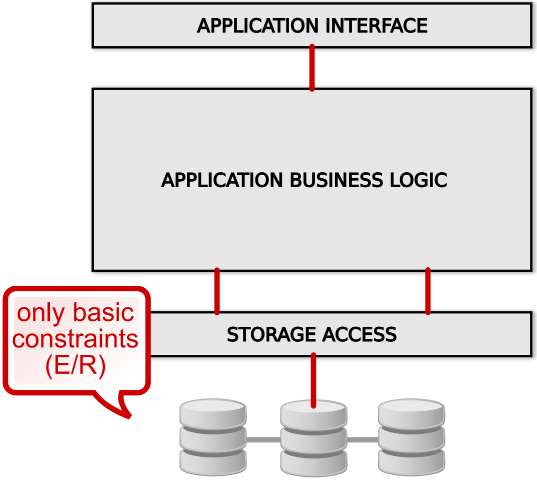
Business Rules

Where are they?



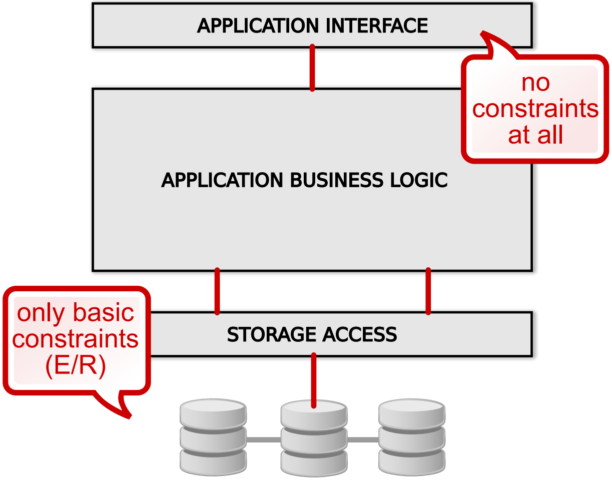
Business Rules

Where are they?



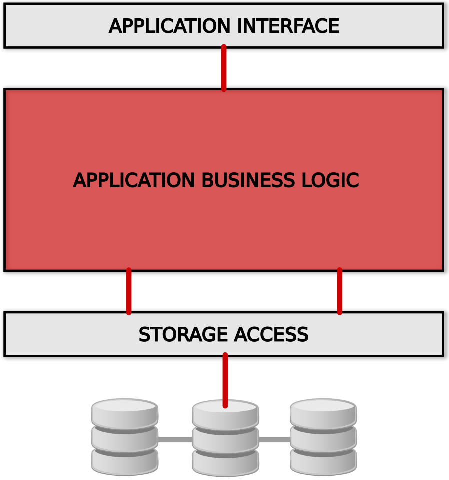
Business Rules

Where are they?



Business Rules

Where are they?



Business Rules

When do we test them?

Since business rules are **not located in a specific unit** or component,

- they are **not covered by unit testing**.

Business Rules
When do we test them?

Since business rules are **not located in a specific unit** or component,

- they are **not covered by unit testing**.

Since business rules dictate **data-related** constraints,

- they are **not the scope of integration testing**.

Business Rules

When do we test them?

Since business rules are **not located in a specific unit** or component,

- they are **not covered by unit testing**.

Since business rules dictate **data-related** constraints,

- they are **not the scope of integration testing**.

Since business rules need to be **respected at all times**,

- they are **not considered when testing the GUI**.

Business Rules

When do we test them?

Since business rules are **not located in a specific unit** or component,

- they are **not covered by unit testing**.

Since business rules dictate **data-related** constraints,

- they are **not the scope of integration testing**.

Since business rules need to be **respected at all times**,

- they are **not considered when testing the GUI**.

Business rules must be tested as part of **system testing**.

Why do DB applications require special testing?

Because of Business Rules

Therefore, **database** or **data-intensive** applications:

- include **business rules** that put constraints on the **data** they handle,
- business rules must be enforced by the system **at all times**,
- **location** of the business rules is **unclear**.

Why do DB applications require special testing?

Because of Business Rules

Therefore, **database** or **data-intensive** applications:

- include **business rules** that put constraints on the **data** they handle,
- business rules must be enforced by the system **at all times**,
- **location** of the business rules is **unclear**.

In this tutorial, we will present a methodology to

test business rules at system testing level.

The theory

How to test business rules with QuickCheck

To test that a data-intensive application complies with the data constraints imposed by its business rules at all times, we use **QuickCheck**:

- an **automatic testing tool**,
- **generates and runs random sequences** of test cases,
- when an error is found, test sequence is shrunk to return a **minimal test case**.

The theory

How to test business rules with QuickCheck

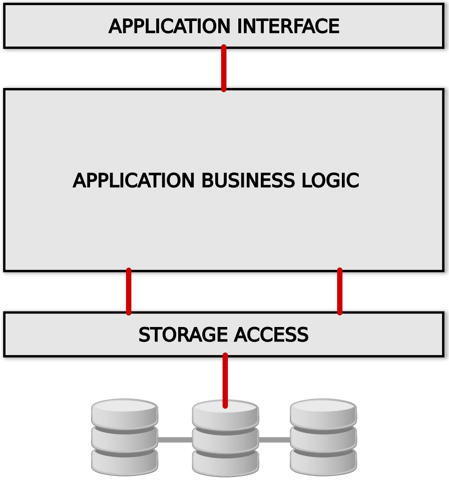
To test that a data-intensive application complies with the data constraints imposed by its business rules at all times, we use **QuickCheck**:

- an **automatic testing tool**,
- **generates and runs random sequences** of test cases,
- when an error is found, test sequence is shrunk to return a **minimal test case**.

In the rest of the tutorial we assume familiarity with Quviq QuickCheck testing tool. We will present the basics of how QuickCheck state machine library works, but explaining these concepts is not the purpose of this specific tutorial.

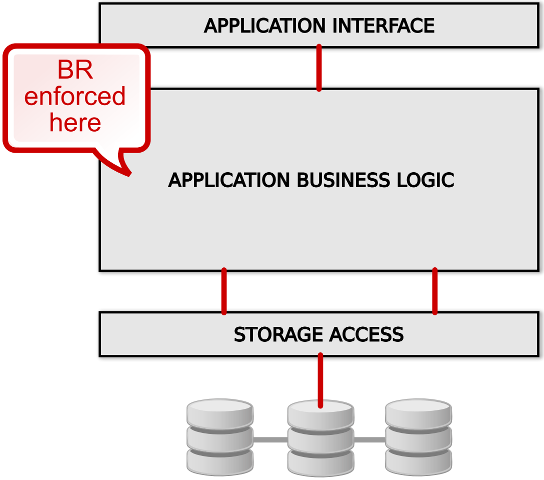
The theory

How to test business rules with QuickCheck



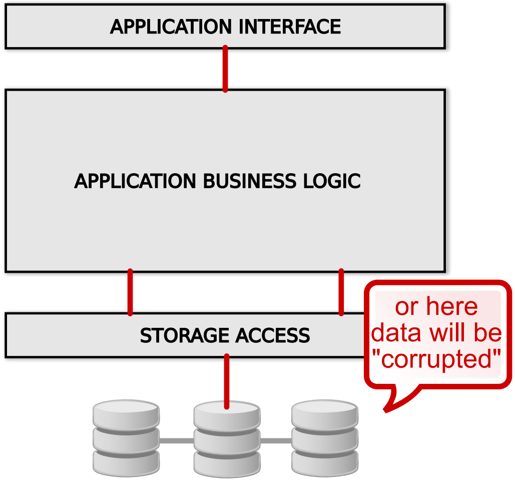
The theory

How to test business rules with QuickCheck



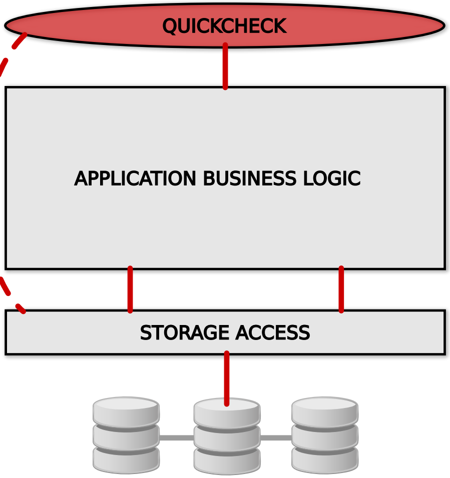
The theory

How to test business rules with QuickCheck



The theory

How to test business rules with QuickCheck



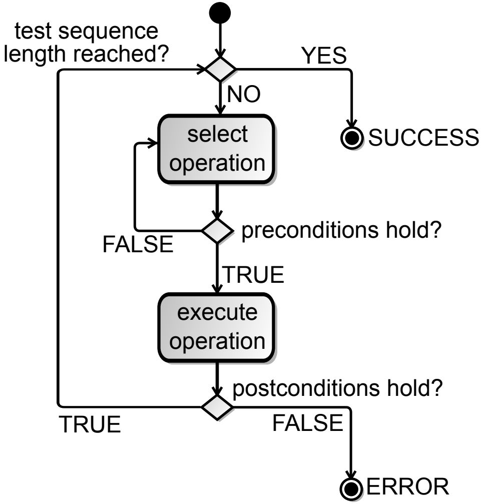
The theory
QuickCheck state machine library

In particular, we use **QuickCheck state machine library**:

- mechanism to easily implement a testing state machine (library callbacks),
- the testing state machine generates and runs test sequences,
- tests are sequences of calls to the functionalities under test.

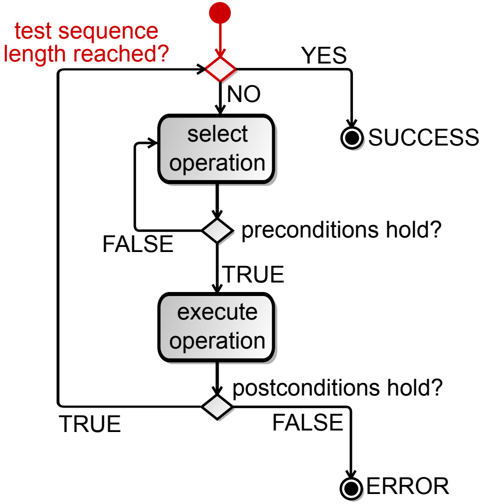
The theory

How are test sequences generated?



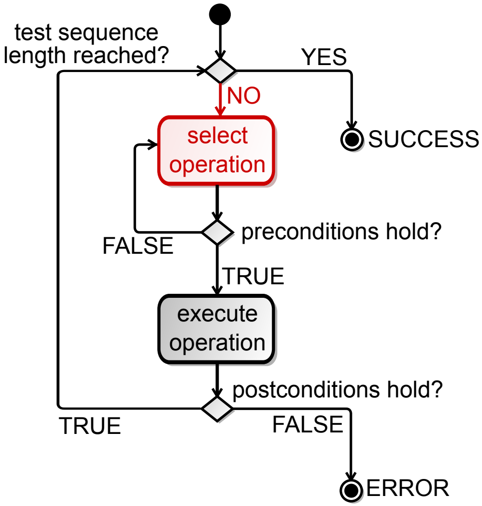
The theory

How are test sequences generated?



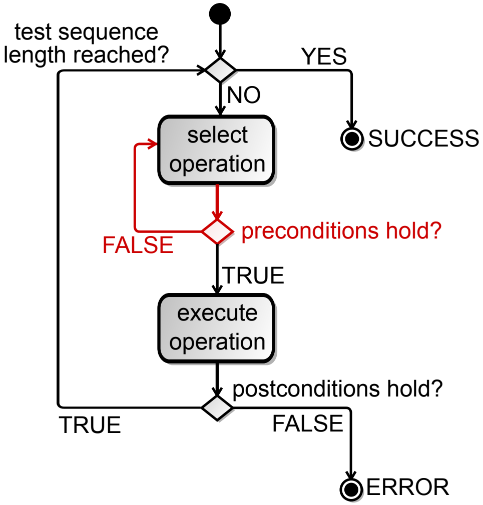
The theory

How are test sequences generated?



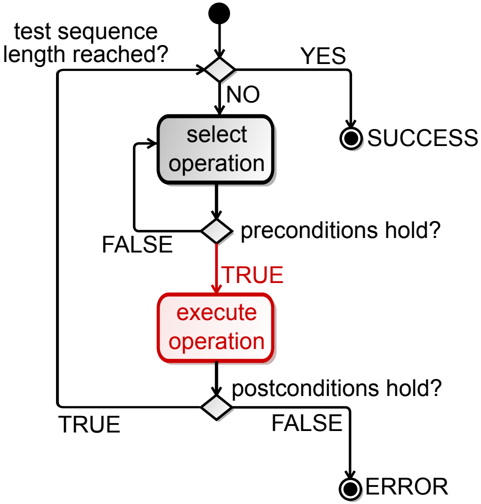
The theory

How are test sequences generated?



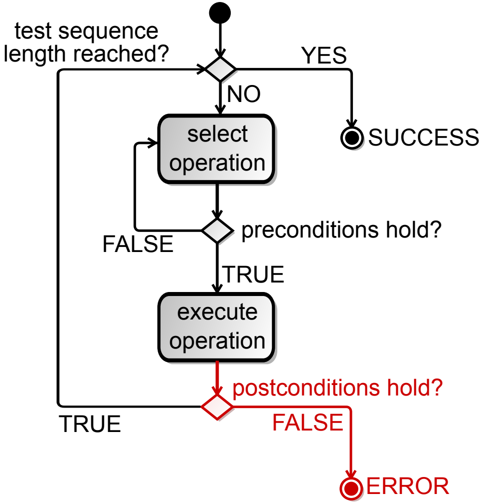
The theory

How are test sequences generated?



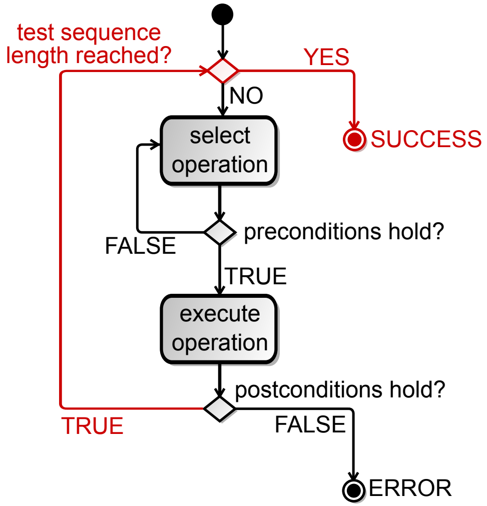
The theory

How are test sequences generated?



The theory

How are test sequences generated?



The theory

QuickCheck statem machine skeleton

```
-module(test_eqc).

-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

-compile(export_all).

-record(state,{useful_info}).

%% Initialize the state
initial_state() ->
    #state{useful_info = []}.

%% Command generator, S is the state
command(S) ->
    oneof([ PUBLIC API OPERATIONS ]).

%% Next state transformation, S is the current state
next_state(S,_V,{call,_,_,_}) ->
    S.

%% Precondition, checked before command is added to the command sequence
precondition(_S,{call,_,_,_}) ->
    true.

%% Postcondition, checked after command has been evaluated
%% OBS: S is the state before next_state(S,_,<command>)
postcondition(_S,{call,_,_,_},_Res) ->
    true.

prop_statem() ->
    ?FORALL(Cmds,commands(?MODULE),
            begin
                {H,S,Res} = run_commands(?MODULE,Cmds),
                ?WHENFAIL(
                   io:format("History: ~p~nState: ~p~nRes: ~p~n",[H,S,Res]),
                   Res == ok)
            end).
```

The practise

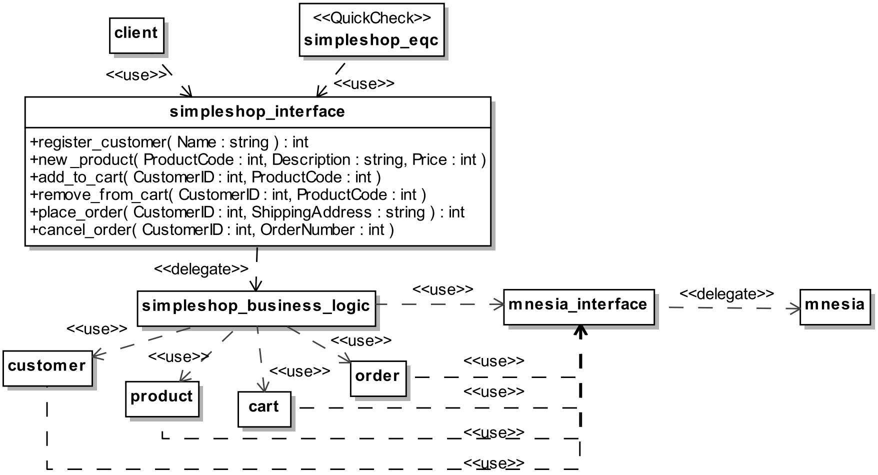
Testing a simple e-shop

Very simple online shop application:

- Register new customer
- Add new product to shop
- Add product to cart
- Remove product from cart
- Place order
- Cancel order

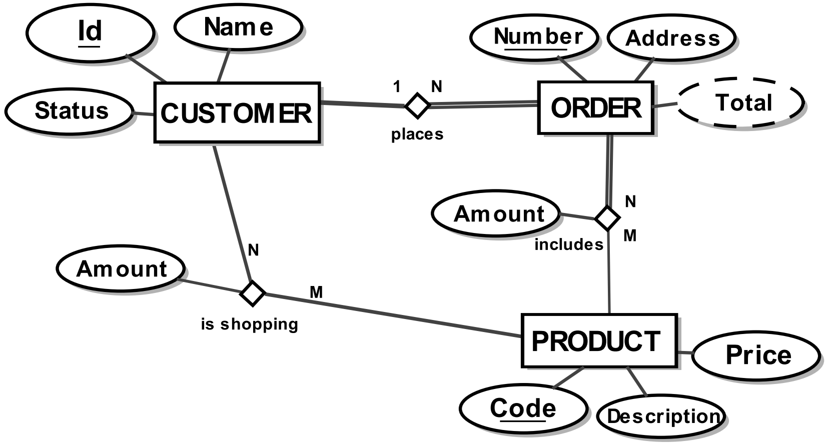
The practise

UML model: main components


client
<<QuickCheck>>
simpleshop_eqc
<<use>>
<<use>>
simpleshop_interface
+register_customer(Name : string) : int
+new_product(ProductCode : int, Description : string, Price : int)
+add_to_cart(CustomerID : int, ProductCode : int)
+remove_from_cart(CustomerID : int, ProductCode : int)
+place_order(CustomerID : int, ShippingAddress : string) : int
+cancel_order(CustomerID : int, OrderNumber : int)
<<delegate>>
simpleshop_business_logic
<<use>>
mnesia_interface
<<delegate>>
mnesia
<<use>>
customer
<<use>>
product
<<use>>
cart
<<use>>
order
<<use>>
<<use>>
<<use>>
<<use>>

The practise

E/R model: basic data constraints



The practise

Golden business rule

Example of business rule (complex data constraint).

The practise

Golden business rule

Example of business rule (complex data constraint).

Business rule

Only golden customers may purchase golden products.

The practise
Golden business rule

Example of business rule (complex data constraint).

Business rule

Only golden customers may purchase golden products.

Business rules may be implemented in different ways…

The practise
Golden business rule

Example of business rule (complex data constraint).

Business rule

Only golden customers may purchase golden products.

Business rules may be implemented in different ways. . .

. . . but we only care they actually **are**.

The practise

Hands-on time!

1. Explore the simple e-shop implementation given,
2. inspect the `simpleshop_eqc` module stub,
3. find out if business rule is respected!

The practise

Hands-on time!

1. Explore the simple e-shop implementation given,
2. inspect the `simpleshop_eqc` module stub,
3. find out if business rule is respected! (and if not, fix it!!)

The practise

Outcome: QuickCheck statem machine skeleton for BR testing (I)

```
-module(testbr_eqc).

-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

-compile(export_all).

-record(state, {useful_info}).

initial_state() ->
    #state{useful_info = []}.

command(S) ->
    oneof([ PUBLIC API OPERATIONS (LOCAL WRAPPERS) ]).

next_state(S,_V,{call,_,_,_}) ->
    S.

precondition(_S,{call,_,_,_}) ->
    true.

postcondition(_S,{call,_,_,_},_Res) ->
    true.

prop_brstatem() ->
    ?FORALL(Cmds, commands(?MODULE),
            begin
                true = check_data_invariant(),
                {H, S, Res} = run_commands(?MODULE, Cmds),
                Invariant = check_data_invariant(),
                clean_up(S),
                ?WHENFAIL(io:format("H ~p~nS ~p~nRes ~p~n", [H, S, Res]),
                          conjunction([{test_execution, Res == ok},
                                       {business_rules, Invariant}]))
             end).
```

The practise

Outcome: QuickCheck statem machine skeleton for BR testing (& II)

```
<command>_local(Args) ->
    Expected = expected_result(<command>, Args),
    Obtained = <command>(Args),
    match(Expected, Obtained).

check_data_invariant() ->
    IMPLEMENTATION OF BUSINESS RULES AS STORAGE QUERIES.

expected_result(<command>, Args) ->
    QUERY STORAGE TO GUESS RESULT.

clean_up(S) ->
    EMPTY STATE BETWEEN TEST SEQUENCES.
```

Summing up
Testing of data-intensive applications

When **testing database or data-intensive applications**,

- special attention must be paid to **data-consistency business rules**,
- data-consistency constraints **cannot** always **be trusted** to the data **storage** and can never be trusted to the **user interface**,
- business rules implementation **may be spread** over the system,
- **system testing** is the most adequate level to test for business rules compliance.

Summing up

Methodology to test BR using QuickCheck

1. Use a **QuickCheck state machine**,
2. keep **state minimum**,
3. add **public API operations as commands**/transitions,
   - use **local wrappers** to **predict** the result according to existing data,
   - and then **match with the result** actually obtained
4. specify **pre- and postconditions** as *true*,
5. formulate **business rules** (invariants) as **queries** to data storage,
6. write property **checking invariants** after each test sequence.

Summing up

I hope this tutorial has been useful!

`Attendants ! thanks`

Get help subscribing to: `quickcheck-questions@quviq.com`
Material for images came from: `openclipart.org`, `kde-look.org`