'(Growing Function-Oriented Software (With Parentheses))

Jivago Alves
@jivagoalves
jalves@EqualExperts

EQUAL EXPERTS

Agenda

The Game

Feedback Loops

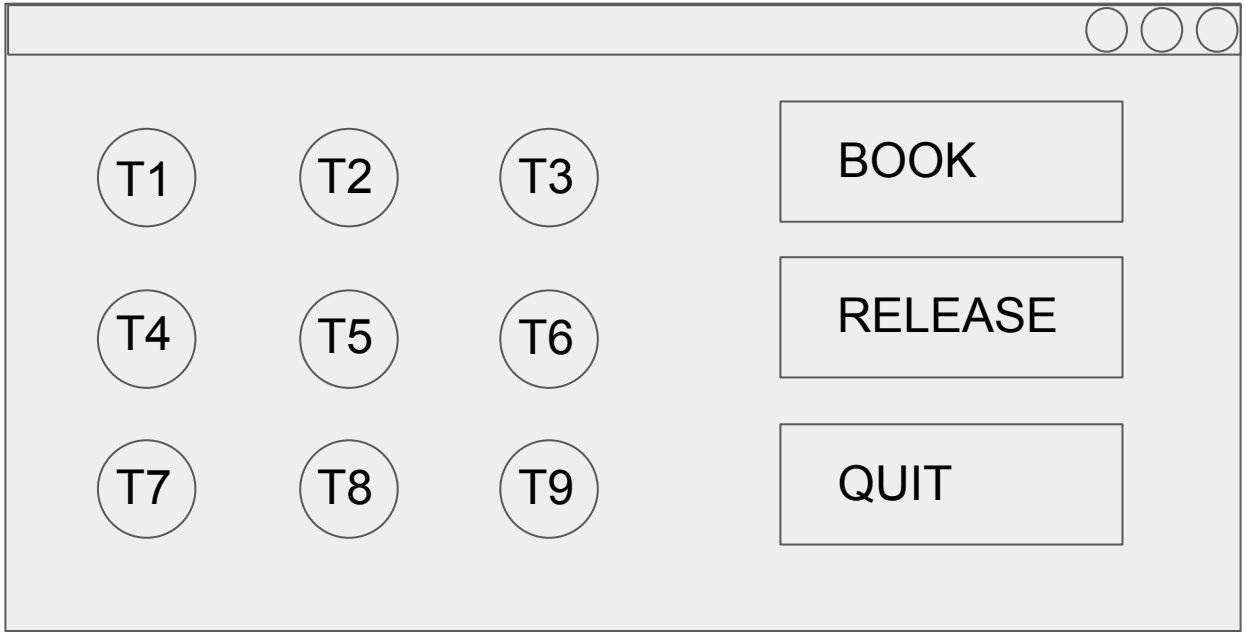
The Architecture

EQUAL EXPERTS

Why should you care?

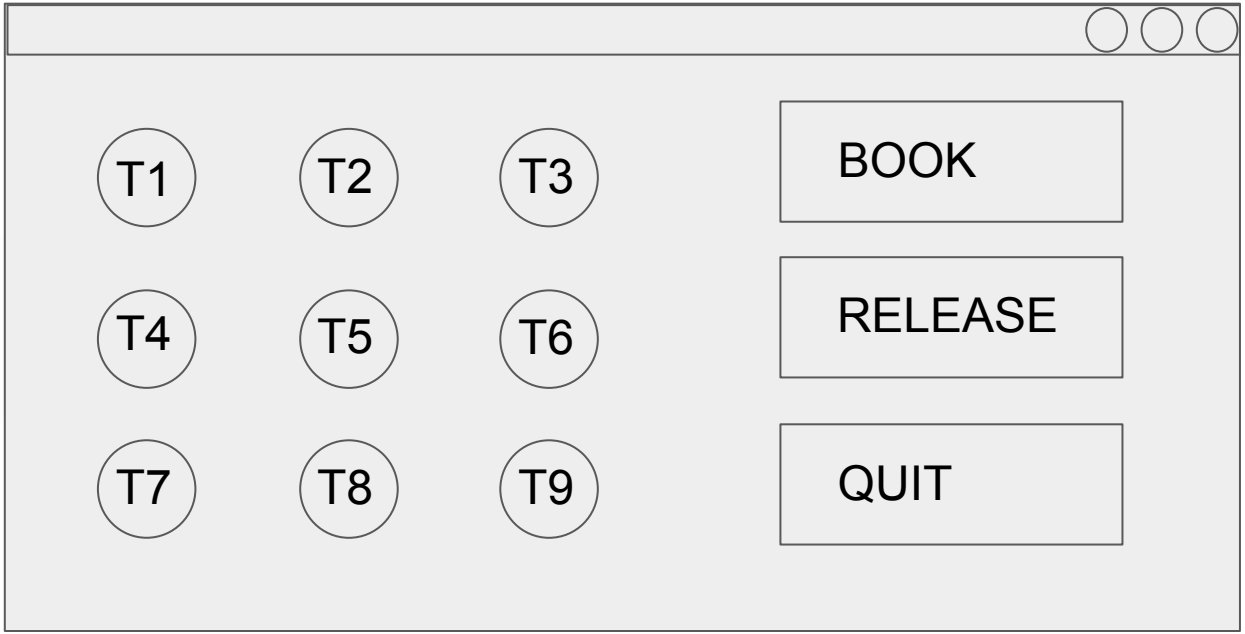
EQUAL EXPERTS

The Table Booking System



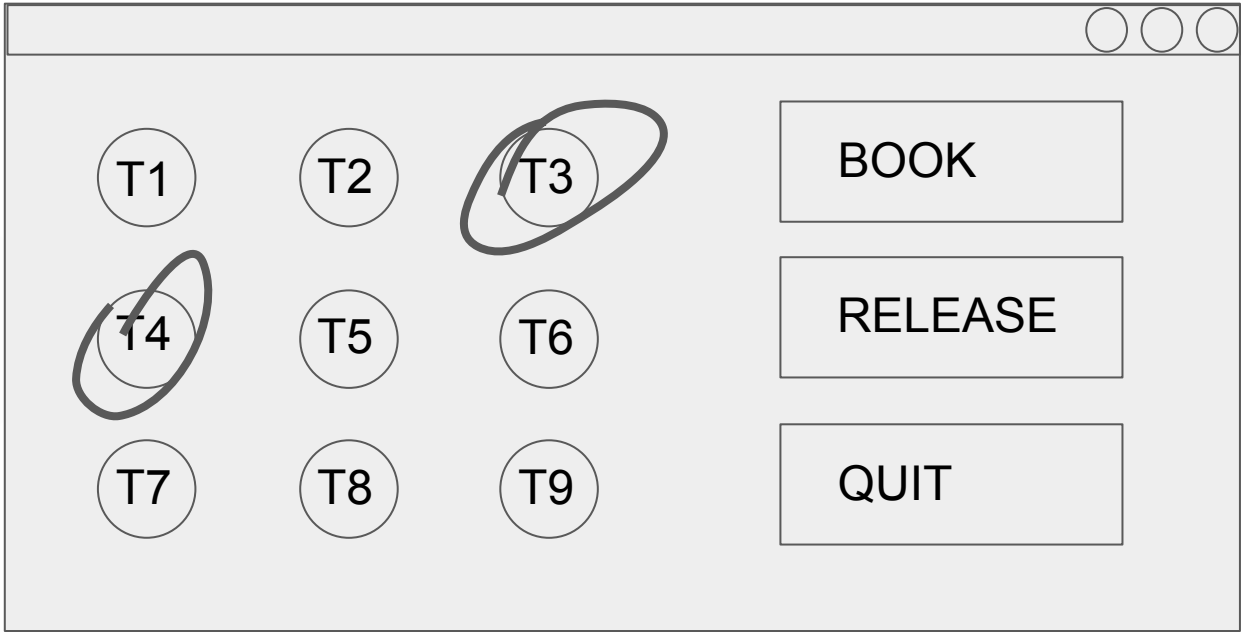
EQUAL EXPERTS

The Table Booking System



EQUAL EXPERTS

The Table Booking System



EQUAL EXPERTS

Essential vs Accidental Complexity

EQUAL EXPERTS

Essential vs Accidental Complexity

Essential Complexity

- Can't be simpler, otherwise it loses value

Examples:

- Booking system needs to differentiate between what's booked and what's not

EQUAL EXPERTS

Essential vs Accidental Complexity

Essential Complexity

- Can't be simpler, otherwise it loses value

Examples:

- Booking system needs to differentiate between what's booked and what's not

Accidental Complexity

- Can be simplified
- Not something we ask for more

Examples:

- Meetings, planning
- Code (e.g. table booking)

EQUAL EXPERTS

Essential vs Accidental Complexity

Essential Complexity

- Can't be simpler, otherwise it loses value

Examples:

- Booking system needs to differentiate between what's booked and what's not

Accidental Complexity

- Can be simplified
- Not something we ask for more

Examples:

- Meetings, planning
- Code (e.g. table booking)

Needs to be reduced to a minimum!

EQUAL EXPERTS

Code is a Liability

EQUAL EXPERTS

Code is a Liability

It is accidental to the problem

EQUAL EXPERTS

Code is a Liability

It is accidental to the problem

We want it to be cheap

EQUAL EXPERTS

High Quality Software is Cheaper

EQUAL EXPERTS

High Quality Software is Cheaper

Ignoring quality leads to complicated code

EQUAL EXPERTS

High Quality Software is Cheaper

Ignoring quality leads to complicated code

Bad design will eventually impact on feature delivery

EQUAL EXPERTS

High Quality Software is Cheaper

Ignoring quality leads to complicated code

Bad design will eventually impact on feature delivery

Keeping high quality code pays off in the long term

EQUAL EXPERTS

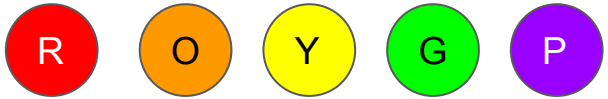
The Game

EQUAL EXPERTS

Code Breaker

8 marbles (colors) available

Code made up of 5 colors

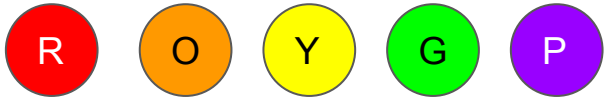


Place the right marbles in the right order

8 attempts to break the code

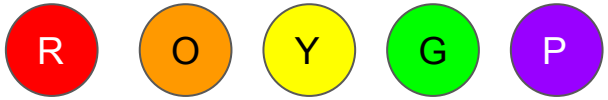
EQUAL EXPERTS

How to Play Code Breaker



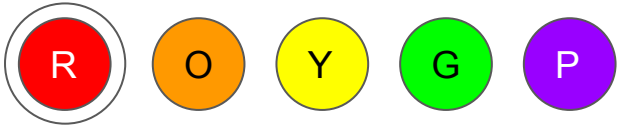
EQUAL EXPERTS

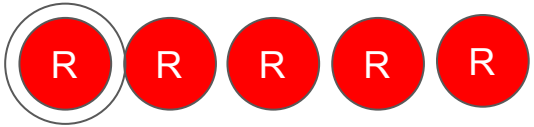
How to Play Code Breaker



EQUAL EXPERTS

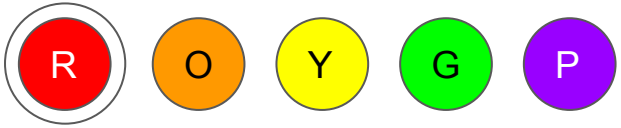
How to Play Code Breaker



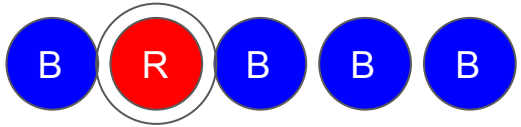


=>O

How to Play Code Breaker



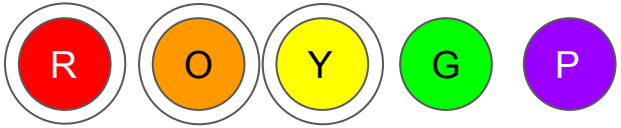
=>O



=>X

EQUAL EXPERTS

How to Play Code Breaker



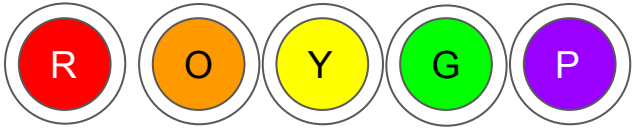
=>O

=>X

=>OXX

EQUAL EXPERTS

How to Play Code Breaker



=>O

=>OOOOO

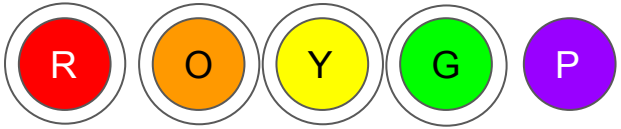
=>X

You won!

=>OXX

EQUAL EXPERTS

How to Play Code Breaker



=>O

=>OXX

=>OOX

=>O

=>OOOX

=>OOOO

=>OO

=>OOX

You lost!

EQUAL EXPERTS

The Specification

EQUAL EXPERTS

The Specification (example)

Feature: Player plays a game

EQUAL EXPERTS

The Specification (example)

Feature: Player plays a game

Scenario: Player starts the game

EQUAL EXPERTS

The Specification (example)

Feature: Player plays a game

Scenario: Player starts the game

Given a new game

When I start the game

Then I should see a welcoming message

EQUAL EXPERTS

Feedback Loops

EQUAL EXPERTS

Feedback Loops

Behaviour-Driven Development (BDD)

REPL-Driven Development (RDD)

Reloaded Workflow

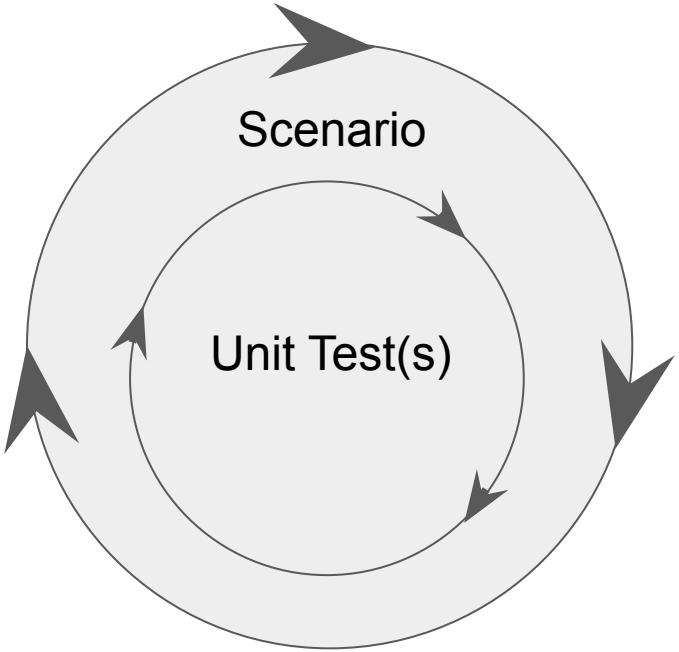
EQUAL EXPERTS

Behaviour-Driven Development

BDD is about describing behaviour from the perspective of the users.

EQUAL EXPERTS

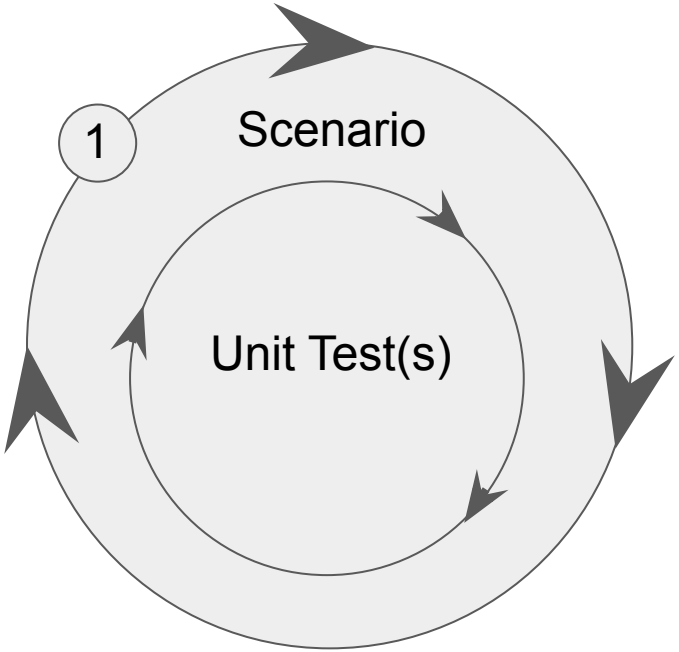
BDD Cycle



EQUAL EXPERTS

BDD Cycle

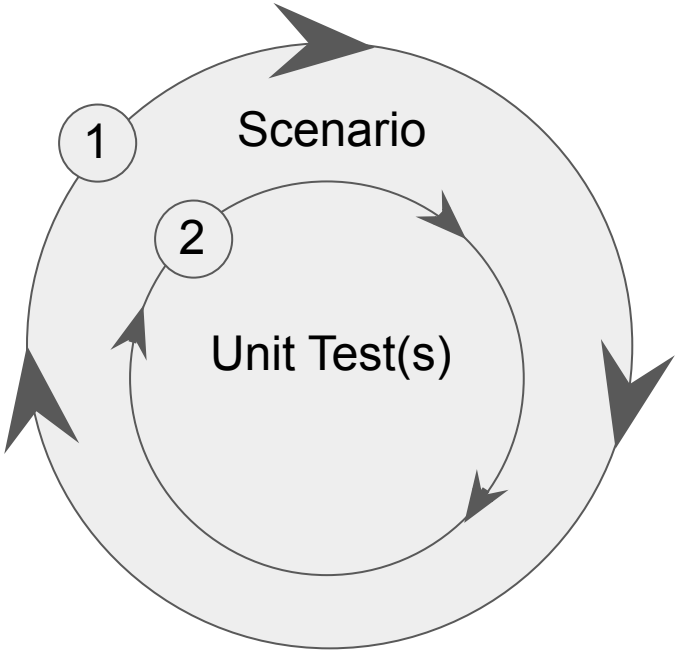
1. Write failing step (Given, When, Then)



EQUAL EXPERTS

BDD Cycle

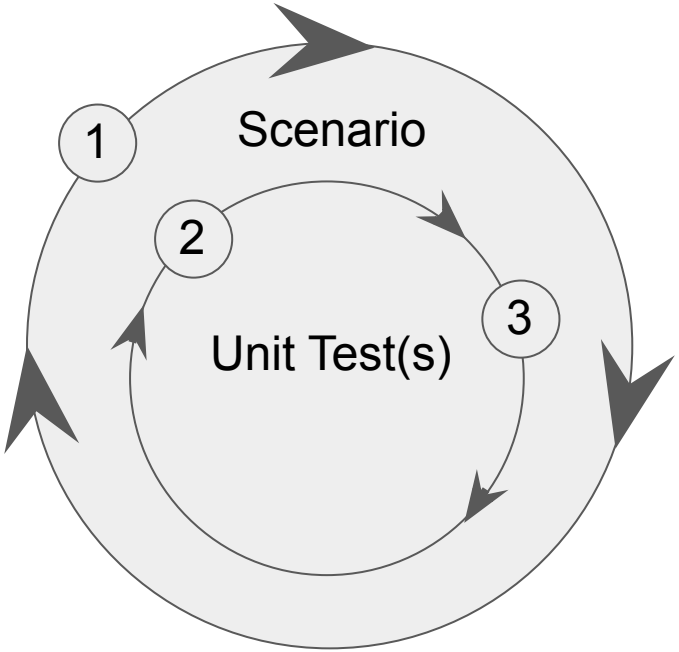
1. Write failing step (Given, When, Then)
2. Write failing unit test



EQUAL EXPERTS

BDD Cycle

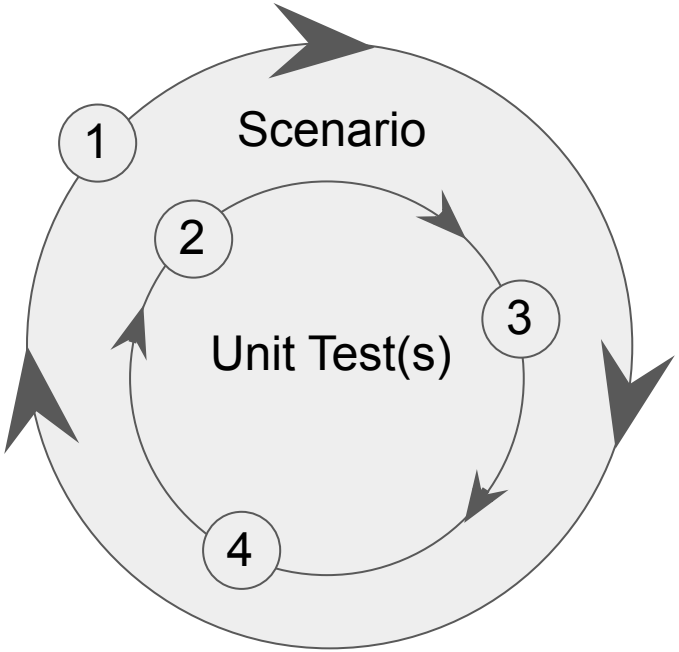
1. Write failing step (Given, When, Then)
    2. Write failing unit test
    3. Make it pass



EQUAL EXPERTS

BDD Cycle

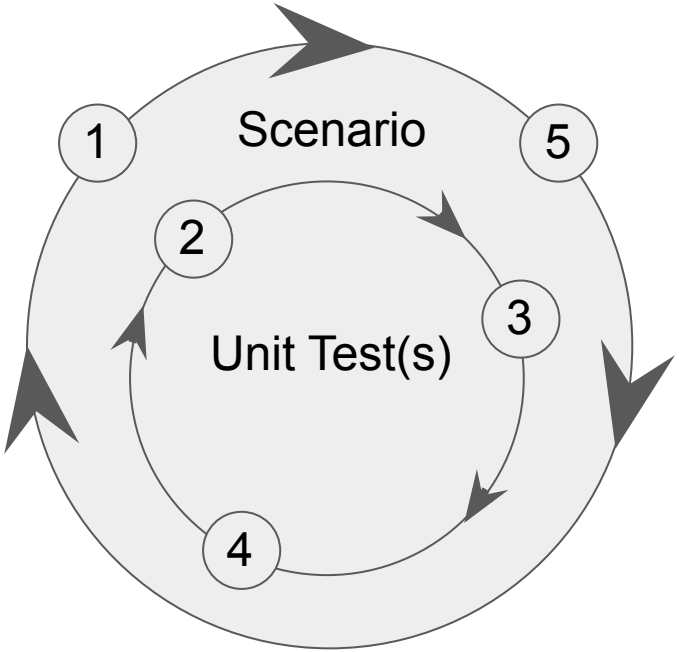
1. Write failing step (Given, When, Then)
2. Write failing unit test
3. Make it pass
4. Refactor



EQUAL EXPERTS

BDD Cycle

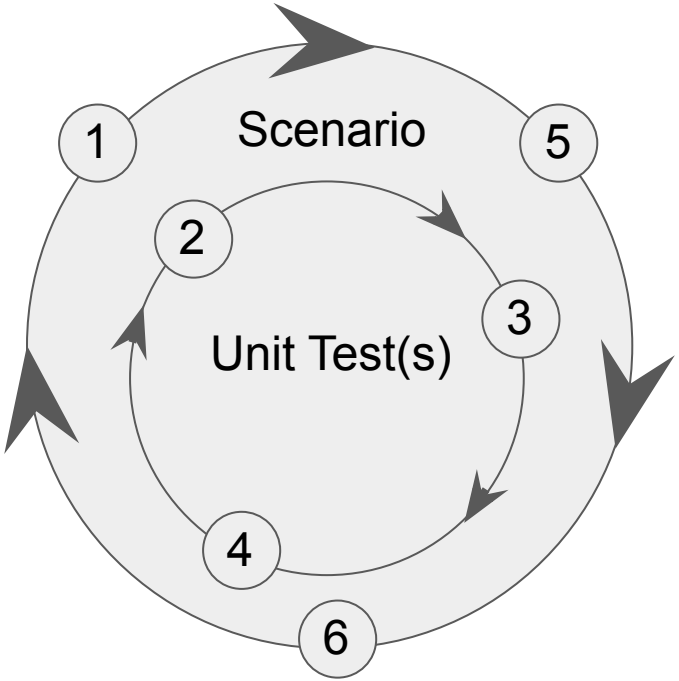
1. Write failing step (Given, When, Then)
   2. Write failing unit test
   3. Make it pass
   4. Refactor
5. Make step pass



EQUAL EXPERTS

BDD Cycle

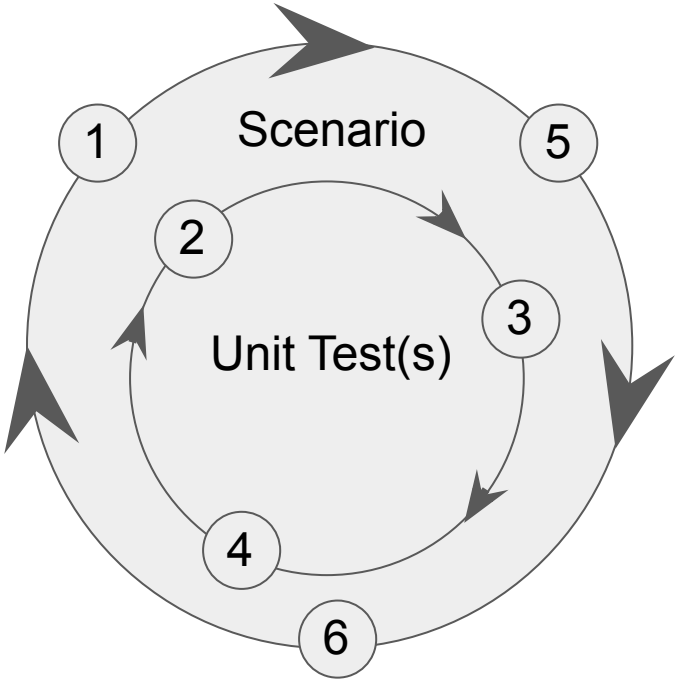
1. Write failing step (Given, When, Then)
   2. Write failing unit test
   3. Make it pass
   4. Refactor
5. Make step pass
6. Refactor



EQUAL EXPERTS

BDD Cycle

1. Write failing step (Given, When, Then)
   2. Write failing unit test
   3. Make it pass
   4. Refactor
5. Make step pass
6. Refactor



EQUAL EXPERTS

REPL-Driven Development

EQUAL EXPERTS

REPL-Driven Development

1. Write code

```
1 (defn plus
2   [a b]
3   (+ a b))
```

EQUAL EXPERTS

REPL-Driven Development

1. Write code
2. Send it to the REPL

```
1 (defn plus
2   [a b]
3   (+ a b)) ; => #'user/plus
```

EQUAL EXPERTS

REPL-Driven Development

1. Write code
2. Send it to the REPL
3. Inspect the result

```
1 (defn plus
2   [a b]
3   (+ a b)) ; => #'user/plus
4
5 (plus 2 2) ; => 4
```

EQUAL EXPERTS

REPL-Driven Development

1. Write code
2. Send it to the REPL
3. Inspect the result

```
1 (defn plus
2   [a b]
3   (+ a b)) ; => #'user/plus
4
5 (plus 2 2) ; => 4
```

EQUAL EXPERTS

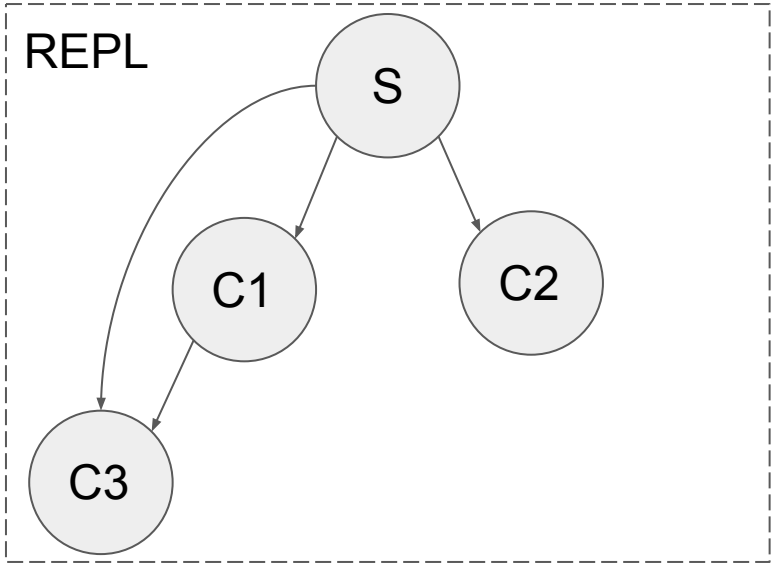
Reloaded Workflow

EQUAL EXPERTS

Reloaded Workflow

0 Given a system running

```
1 (defn plus
2   [a b]
3   (+ a b))
```

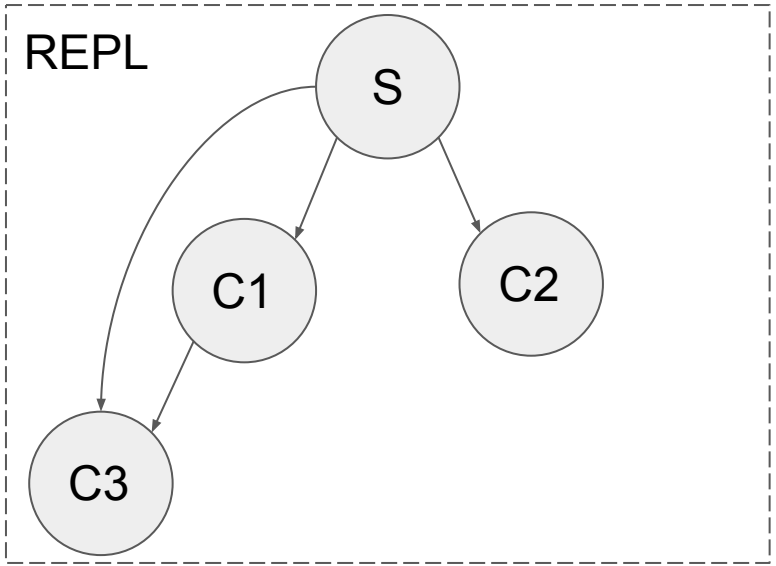


EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change

```
1 (defn plus
2   [& args]
3   (apply + args))
```

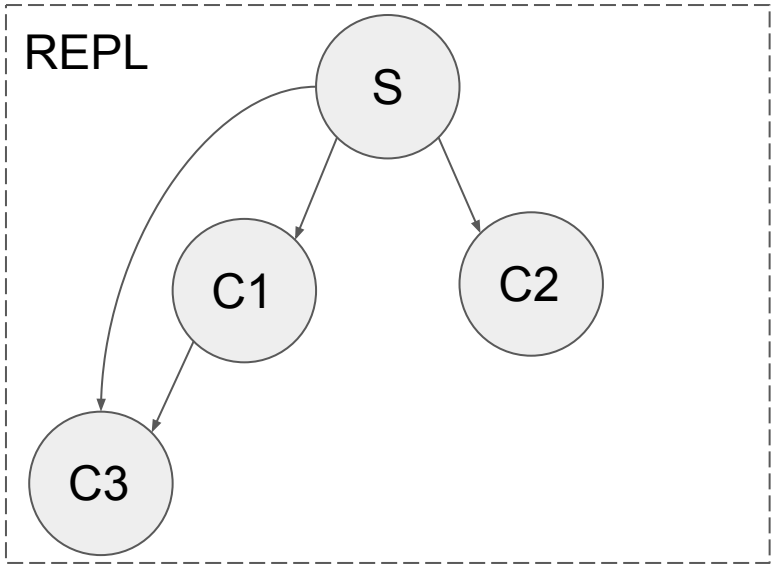


EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change
2. Reset the system

```
1 (defn plus
2   [& args]
3   (apply + args))
```



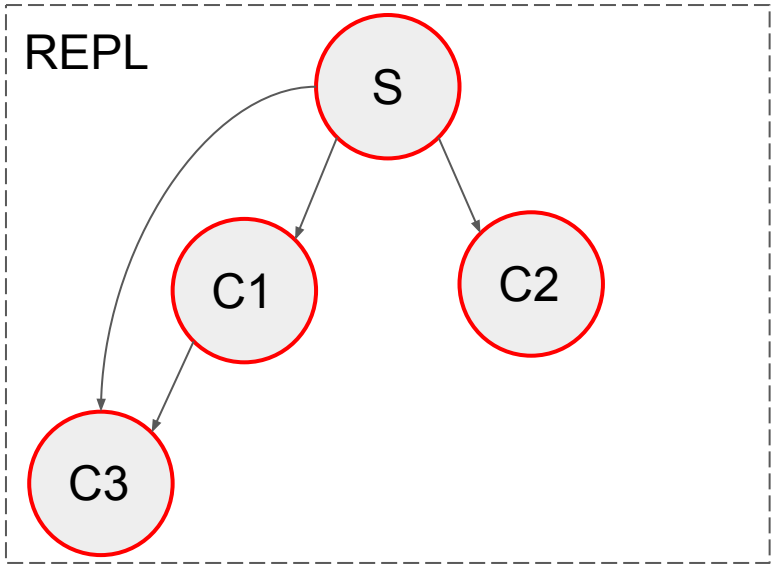
EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change
2. Reset the system

   Stop all components

```
1 (defn plus
2   [& args]
3   (apply + args))
```

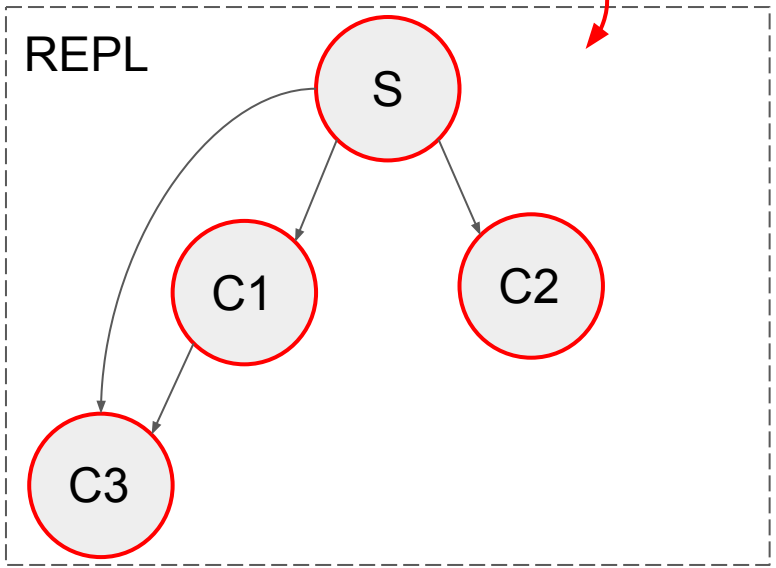


EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change
2. Reset the system
   - Stop all components
   - Refresh code

```
1 (defn plus
2   [& args]
3   (apply + args))
```

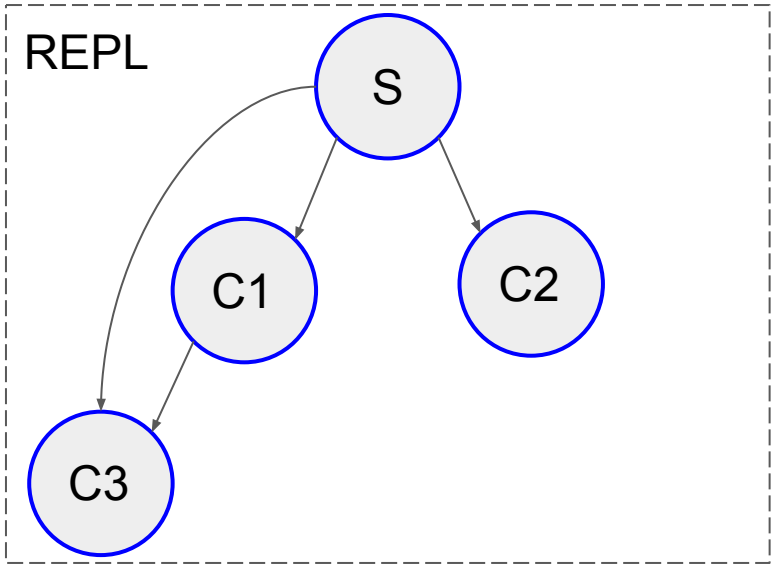


EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change
2. Reset the system
   - Stop all components
   - Refresh code
   - Start all components

```
1 (defn plus
2   [& args]
3   (apply + args))
```

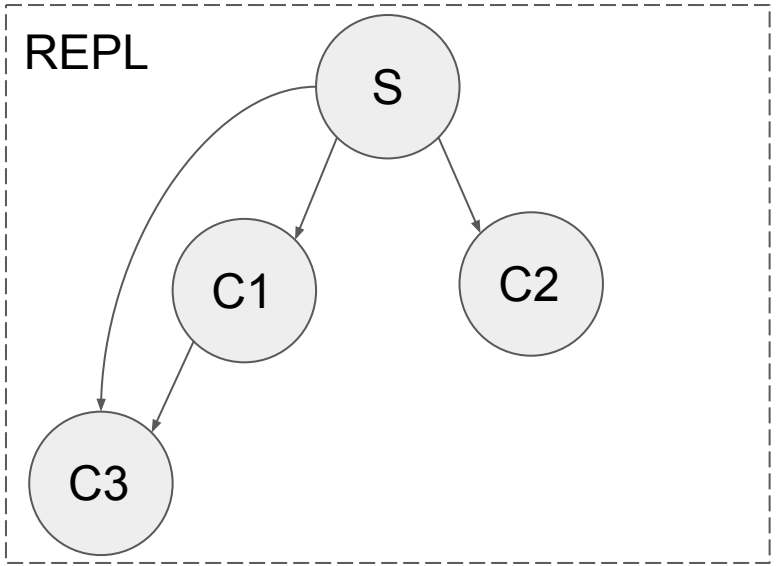


EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change
2. Reset the system
   - Stop all components
   - Refresh code
   - Start all components
3. Inspect it

```clojure
1 (defn plus
2   [& args]
3   (apply + args))
```



EQUAL EXPERTS

Reloaded Workflow

0. Given a system running
1. Make a change
2. Reset the system
   - Stop all components
   - Refresh code
   - Start all components
3. Inspect it

Libraries

Component + reloaded.repl

Mount

Integrant

EQUAL EXPERTS

Feedback Loop (example)

Playing from the Command-Line Interface (CLI) ...

EQUAL EXPERTS

Feedback Loop (example)

```
1 (Feature "Player plays a game"
2
3   (Scenario "Player starts a game"
4     (Given "a new game"
5       (When "the player starts the game"
6         (Then "the player sees a welcoming message")))))
```

EQUAL EXPERTS

Writing failing step

```
1  (Feature "Player plays a game"
2
3    (Scenario "Player starts a game"
4      (Given "a new game"
5
6        (When "the player starts the game"
7          (let [out (with-out-str
8                      (cli/start))]
9
10           (Then "the player sees a welcoming message"))))))
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)
:error-while-loading features.player-plays-a-game
#error {
 :cause "No such var: cli/start"
 :via
 [{:type clojure.lang.Compiler$CompilerException
   :message "Syntax error compiling at (features/player_plays_a_game.clj:19:23)."
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)
:error-while-loading features.player-plays-a-game
#error {
 :cause "No such var: cli/start"
 :via
 [{:type clojure.lang.Compiler$CompilerException
   :message "Syntax error compiling at (features/player_plays_a_game.clj:19:23)."
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)
:error-while-loading features.player-plays-a-game
#error {
 :cause "No such var: cli/start"
 :via
 [{:type clojure.lang.Compiler$CompilerException
   :message "Syntax error compiling at (features/player_plays_a_game.clj:19:23)."
```

EQUAL EXPERTS

Making it pass

```
1 (ns codebreaker.cli)
2
3 (defn start [])
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

Ran 1 tests containing 0 assertions.
0 failures, 0 errors.
{:error 0 :fail 0 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

Ran 1 tests containing 0 assertions.
0 failures, 0 errors.
{:error 0 :fail 0 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

Ran 1 tests containing 0 assertions.
0 failures, 0 errors.
{:error 0 :fail 0 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Writing next (failing) step

```
1 (Feature "Player plays a game"
2
3   (Scenario "Player starts a game"
4     (Given "a new game"
5
6       (When "the player starts the game"
7         (let [out (with-out-str
8                     (cli/start))]
9
10          (Then "the player sees a welcoming message"
11            (is (= "Welcome!" (first (lines out))))))))))
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

FAIL in (player-plays-a-game) (player_plays_a_game.clj:22)

Feature: Player plays a game
 Scenario: Player starts a game
  Given a new game
  When the player starts the game
  Then the player sees a welcoming message

expected: "Welcome!"
  actual: ""

    diff: - "Welcome!"

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
{:error 0 :fail 1 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

FAIL in (player-plays-a-game) (player_plays_a_game.clj:22)

Feature: Player plays a game
 Scenario: Player starts a game
  Given a new game
  When the player starts the game
  Then the player sees a welcoming message

expected: "Welcome!"
  actual: ""

    diff: - "Welcome!"

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
{:error 0 :fail 1 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

FAIL in (player-plays-a-game) (player_plays_a_game.clj:22)

Feature: Player plays a game
 Scenario: Player starts a game
  Given a new game
  When the player starts the game
  Then the player sees a welcoming message

expected: "Welcome!"
  actual: ""

    diff: - "Welcome!"

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
{:error 0 :fail 1 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Making it pass

```
1 (ns codebreaker.cli)
2
3 (defn start []
4   (println "Welcome!"))
```

EQUAL EXPERTS

Running tests

```
user=> (run-tests)
:reloading (support codebreaker.cli user features.player-plays-a-game)

Testing features.player-plays-a-game

Ran 1 tests containing 0 assertions.
0 failures, 0 errors.
{:error 0 :fail 0 :pass 0 :test 1 :type :summary}
```

EQUAL EXPERTS

Inspecting in the REPL

```
1 (ns codebreaker.cli)
2
3 (defn start []
4   (println "Welcome!"))
```

```
user=> (cli/start)
Welcome!
nil
```

EQUAL EXPERTS

The Architecture

EQUAL EXPERTS

Player plays a game

EQUAL EXPERTS

Player plays a game

Scenario: Player starts a game 

EQUAL EXPERTS

Player plays a game

Scenario: Player starts a game ✓

Scenario: Player wins the game ✓

EQUAL EXPERTS

Player plays a game

Scenario: Player starts a game ✓

Scenario: Player wins the game ✓

Scenario: Player loses the game ✓

EQUAL EXPERTS

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Player plays a game

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

EQUAL EXPERTS

Side Effects

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

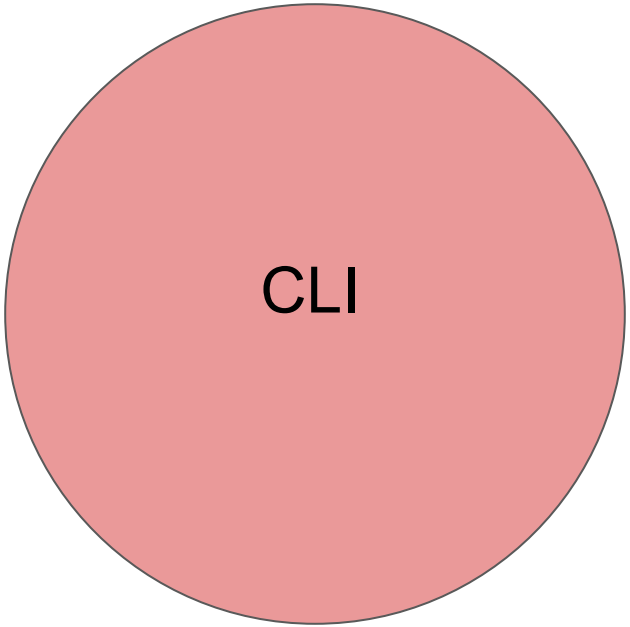
EQUAL EXPERTS

Business Rules

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

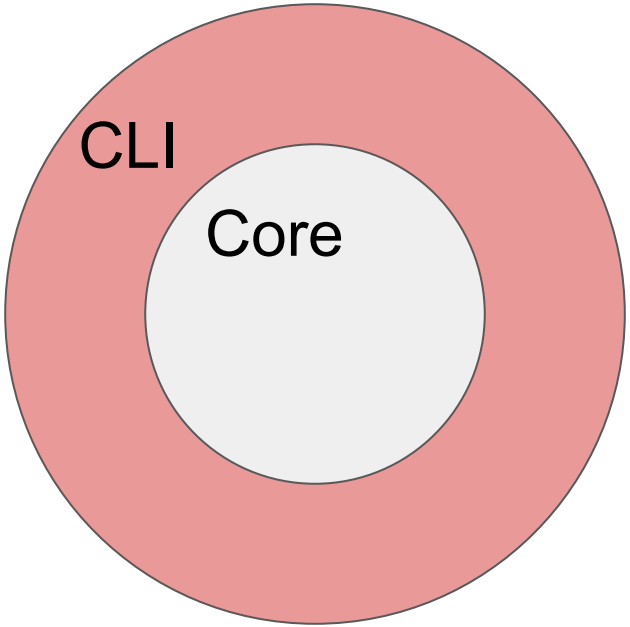
EQUAL EXPERTS

The "Architecture"



EQUAL EXPERTS

What if we have...



EQUAL EXPERTS

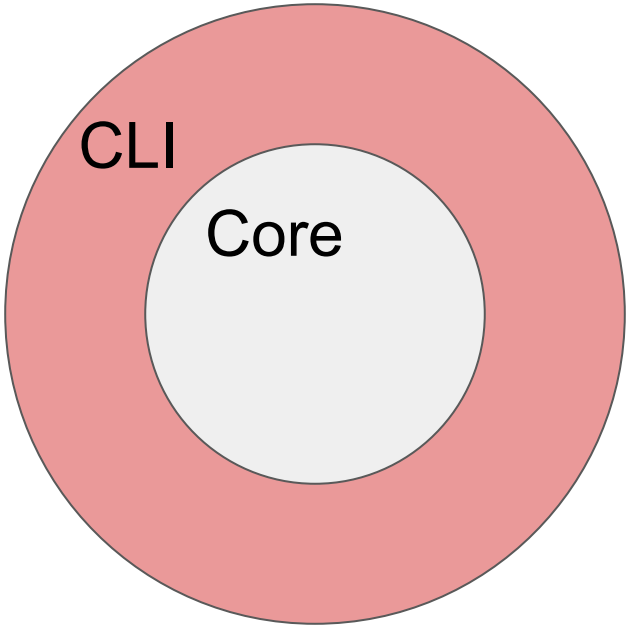
The Architecture

CLI

Side Effects

Presentation

Depends on Core



EQUAL EXPERTS

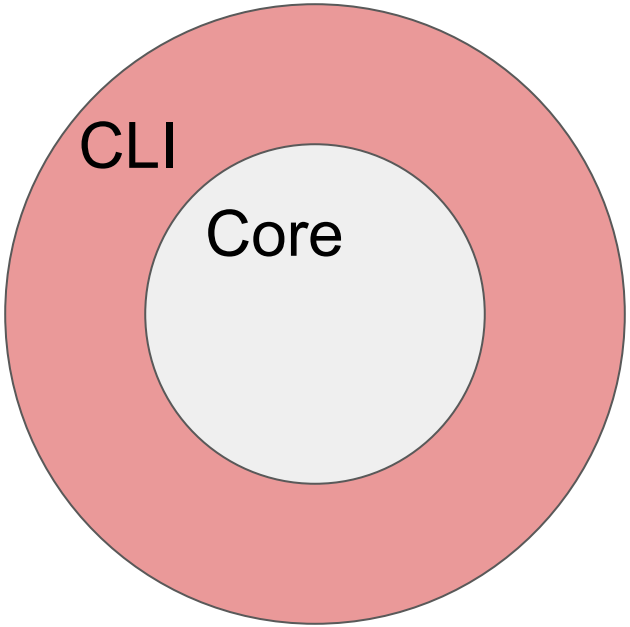
The Architecture

CLI

Side Effects

Presentation

Depends on Core



Core

Pure

Business Rules

No External Dependencies

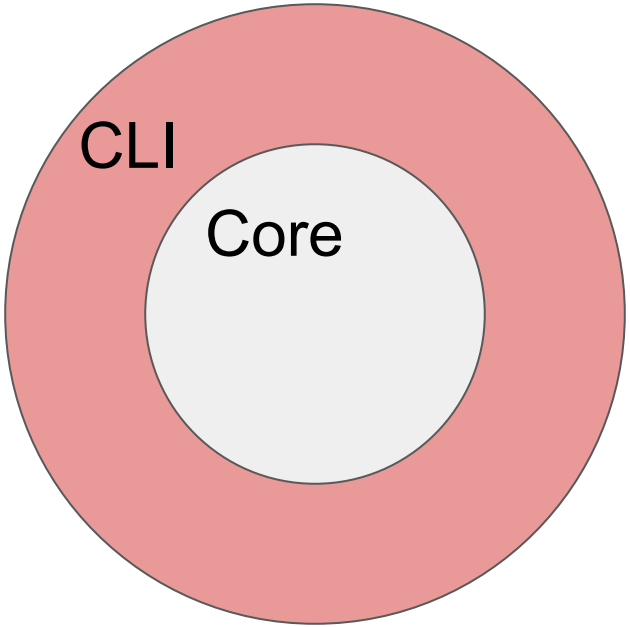
EQUAL EXPERTS

The Architecture

Hexagonal Architecture

Onion Architecture

Functional Core / Imperative Shell



EQUAL EXPERTS

Functional Core / Imperative Shell

EQUAL EXPERTS

EQUAL EXPERTS

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [attempt 1]
    (if-not (exceeded-attempts? attempt)
      (do
        (print "> ")
        (flush)
        (when-let [guess (read-line)]
          (if (= code guess)
            (println "You won!")
            (recur (inc attempt)))))
      (println "You lost!"))))
```

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (print "> ")
14         (flush)
15         (when-let [guess (read-line)]
16           (if (= code guess)
17             (println "You won!")
18             (recur (inc attempt)))))
19       (println "You lost!"))))
```

```
 7 (defn start
 8   [{:as game :keys [code]}]
 9   (println "Welcome!")
10   (loop [attempt 1]
11     (if-not (exceeded-attempts? attempt)
12       (do
13         (prompt)
14         (when-let [guess (read-line)]
15           (if (= code guess)
16             (println "You won!")
17             (recur (inc attempt)))))
18       (println "You lost!"))))
```


(defn prompt []
(print "> ")
(flush))
(defn start
[{:as game :keys [code]}]
(println "Welcome!")
(loop [attempt 1]
(if-not (exceeded-attempts? attempt)
(do
(prompt)
(when-let [guess (read-line)]
(if (= code guess)
(println "You won!")

```
 7 (defn prompt []
 8   (print "> ")
 9   (flush))
10
11 (defn start
12   [{:as game :keys [code]}]
13   (println "Welcome!")
14   (loop [attempt 1]
15     (if-not (exceeded-attempts? attempt)
16       (do
17         (prompt)
18         (when-let [guess (read-line)]
19           (if (= code guess)
20             (println "You won!")
```

```
 7 (defn prompt []
 8   (print "> ")
 9   (flush))
10
11 (defn start
12   [{:as game :keys [code]}]
13   (println "Welcome!")
14   (loop [attempt 1]
15     (if-not (exceeded-attempts? attempt)
16       (do
17         (prompt)
18         (when-let [guess (ask-for-guess)]
19           (if (= code guess)
20             (println "You won!")
```

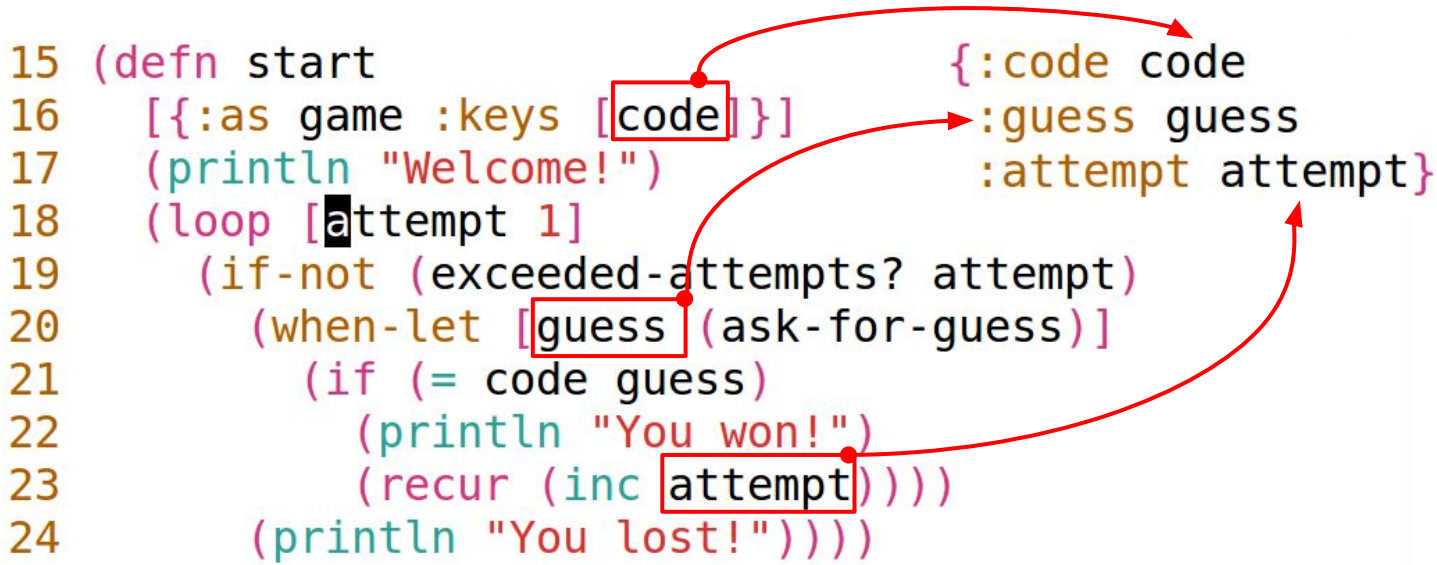

(defn ask-for-guess []
(read-line))
(defn start
[{:as game :keys [code]}]
(println "Welcome!")
(loop [attempt 1]
(if-not (exceeded-attempts? attempt)
(do
(prompt)
(when-let [guess (ask-for-guess)]
(if (= code guess)
(println "You won!")
(recur (inc attempt))))))


11 (defn ask-for-guess []
12 (read-line))
13
14 (defn start
15 [{:as game :keys [code]}]
16 (println "Welcome!")
17 (loop [attempt 1]
18 (if-not (exceeded-attempts? attempt)
19 (do
20 (prompt)
21 (when-let [guess (ask-for-guess)]
22 (if (= code guess)
23 (println "You won!")
24 (recur (inc attempt)))))

```
11 (defn ask-for-guess []
12   (prompt)
13   (read-line))
14
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [attempt 1]
19     (if-not (exceeded-attempts? attempt)
20       (do
21         (when-let [guess (ask-for-guess)]
22           (if (= code guess)
23             (println "You won!")
24             (recur (inc attempt)))))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [attempt 1]
19     (if-not (exceeded-attempts? attempt)
20       (do
21         (when-let [guess (ask-for-guess)]
22           (if (= code guess)
23             (println "You won!")
24             (recur (inc attempt)))))
25       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [attempt 1]
19     (if-not (exceeded-attempts? attempt)
20       (when-let [guess (ask-for-guess)]
21         (if (= code guess)
22           (println "You won!")
23           (recur (inc attempt))))
24       (println "You lost!"))))
```


15 (defn start
16 [{:as game :keys [code]}]
17 (println "Welcome!")
18 (loop [attempt 1]
19 (if-not (exceeded-attempts? attempt)
20 (when-let [guess (ask-for-guess)]
21 (if (= code guess)
22 (println "You won!")
23 (recur (inc attempt))))
24 (println "You lost!"))))
{:code code
:guess guess
:attempt attempt}

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [attempt 1]
19     (if-not (exceeded-attempts? attempt)
20       (when-let [guess (ask-for-guess)]
21         (if (= code guess)
22           (println "You won!")
23           (recur (inc attempt))))
24       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts? attempt)
20       (when-let [guess (ask-for-guess)]
21         (if (= code guess)
22           (println "You won!")
23           (recur (inc attempt))))
24       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts? attempt)
20       (when-let [guess (ask-for-guess)]
21         (if (= code guess)
22           (println "You won!")
23           (recur (inc attempt))))
24       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts?
20               (:attempt current-game))
21       (when-let [guess (ask-for-guess)]
22         (if (= code guess)
23           (println "You won!")
24           (recur (inc attempt))))
25       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts?
20               (:attempt current-game))
21       (when-let [guess (ask-for-guess)]
22         (if (= code guess)
23           (println "You won!")
24           (recur (inc attempt))))
25       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts?
20                (:attempt current-game))
21       (when-let [guess (ask-for-guess)]
22         (if (= code guess)
23           (println "You won!")
24           (recur (update current-game
25                          :attempt inc))))
26       (println "You lost!"))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (if-not (exceeded-attempts?
                (:attempt current-game))
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (-> current-game
                     (update :attempt inc)))))
      (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts?
20                (:attempt current-game))
21       (when-let [guess (ask-for-guess)]
22         (if (= code guess)
23           (println "You won!")
24           (recur (-> current-game
25                      (update :attempt inc)
26                      (assoc :guess guess)))))
27       (println "You lost!"))))
```

```
15 (defn start
16   [{:as game :keys [code]}]
17   (println "Welcome!")
18   (loop [current-game (assoc game :attempt 1)]
19     (if-not (exceeded-attempts?
20                (:attempt current-game))
21       (when-let [guess (ask-for-guess)]
22         (if (= code guess)
23           (println "You won!")
24           (recur (-> current-game
25                      (update :attempt inc)
26                      (assoc :guess guess)))))
27       (println "You lost!"))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (if-not (exceeded-attempts?
                (:attempt current-game))
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (play-game current-game guess))))
      (println "You lost!"))))
```


15 (defn play-game
16 [game guess]
17 (-> game
18 (update :attempt inc)
19 (assoc :guess guess))
20
21 (defn start
22 [{:as game :keys [code]}]
23 (println "Welcome!")
24 (loop [current-game (assoc game :attempt 1)]
25 (if-not (exceeded-attempts?
26 (:attempt current-game))
27 (when-let [guess (ask-for-guess)]

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (if-not (exceeded-attempts?
              (:attempt current-game))
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (play-game current-game guess))))
      (println "You lost!"))))
```

```
21 (defn start
22   [{:as game :keys [code]}]
23   (println "Welcome!")
24   (loop [current-game (assoc game :attempt 1)]
25     (if-not (exceeded-attempts?
26               (:attempt current-game))
27       (when-let [guess (ask-for-guess)]
28         (if (= code guess)
29           (println "You won!")
30           (recur (play-game current-game guess))))
31       (println "You lost!"))))
```

```
21 (defn start
22   [{:as game :keys [code]}]
23   (println "Welcome!")
24   (loop [current-game (assoc game :attempt 1)]
25     (if-not (game-over? current-game)
26       (when-let [guess (ask-for-guess)]
27         (if (= code guess)
28           (println "You won!")
29           (recur (play-game current-game guess))))
30       (println "You lost!"))))
```

```clojure
(defn game-over?
  [game]
  (exceeded-attempts? (:attempt game)))

(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (if-not (game-over? current-game)
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (play-game current-game guess))))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (if-not (game-over? current-game)
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (play-game current-game guess))))
      (println "You lost!"))))
```

```
25 (defn start
26   [{:as game :keys [code]}]
27   (println "Welcome!")
28   (loop [current-game (assoc game :attempt 1)]
29     (cond
30       (game-over? current-game)
31       (println "You lost!"))
32     (if-not (game-over? current-game)
33       (when-let [guess (ask-for-guess)]
34         (if (= code guess)
35           (println "You won!")
36           (recur (play-game current-game guess)))))))
```

```
25 (defn start
26   [{:as game :keys [code]}]
27   (println "Welcome!")
28   (loop [current-game (assoc game :attempt 1)]
29     (cond
30       (game-over? current-game)
31       (println "You lost!"))
32     (if-not (game-over? current-game)
33       (when-let [guess (ask-for-guess)]
34         (if (= code guess)
35           (println "You won!")
36           (recur (play-game current-game guess)))))))
```

```
25 (defn start
26   [{:as game :keys [code]}]
27   (println "Welcome!")
28   (loop [current-game (assoc game :attempt 1)]
29     (cond
30       (game-over? current-game)
31       (println "You lost!")
32
33       :else
34       (if-not (game-over? current-game)
35         (when-let [guess (ask-for-guess)]
36           (if (= code guess)
37             (println "You won!")
```

```
25 (defn start
26   [{:as game :keys [code]}]
27   (println "Welcome!")
28   (loop [current-game (assoc game :attempt 1)]
29     (cond
30       (game-over? current-game)
31       (println "You lost!")
32
33       :else
34       (if-not (game-over? current-game)
35         (when-let [guess (ask-for-guess)]
36           (if (= code guess)
37             (println "You won!")
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      :else
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (play-game current-game guess)))))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      :else
      (when-let [guess (ask-for-guess)]
        (if (= code guess)
          (println "You won!")
          (recur (play-game current-game guess)))))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      :else
      (when-let [guess (ask-for-guess)]
        (if (won? current-game guess)
          (println "You won!")
          (recur (play-game current-game guess)))))))
```

```
25 (defn won?
26   [game guess]
27   (= (:code game) guess))
28
29 (defn start
30   [{:as game :keys [code]}]
31   (println "Welcome!")
32   (loop [current-game (assoc game :attempt 1)]
33     (cond
34       (game-over? current-game)
35       (println "You lost!")
36
37       :else
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      :else
      (when-let [guess (ask-for-guess)]
        (if (won? current-game guess)
          (println "You won!")
          (recur (play-game current-game guess)))))))
```

```
29 (defn start
30   [{:as game :keys [code]}]
31   (println "Welcome!")
32   (loop [current-game (assoc game :attempt 1)]
33     (cond
34       (game-over? current-game)
35       (println "You lost!")
36
37       (won? current-game guess)
38       (println "You won!")
39
40       :else
41       (when-let [guess (ask-for-guess)]
```

```
29 (defn start
30   [{:as game :keys [code]}]
31   (println "Welcome!")
32   (loop [current-game (assoc game :attempt 1)]
33     (cond
34       (game-over? current-game)
35       (println "You lost!")
36
37       (won? current-game guess)
38       (println "You won!")
39
40       :else
41       (when-let [guess (ask-for-guess)]
```

```
29 (defn start
30   [{:as game :keys [code]}]
31   (println "Welcome!")
32   (loop [current-game (assoc game :attempt 1)]
33     (cond
34       (game-over? current-game)
35       (println "You lost!")
36
37       (won? current-game)
38       (println "You won!")
39
40       :else
41       (when-let [guess (ask-for-guess)]
```

```
25 (defn won?
26   [game guess]
27   (= (:code game) guess))
28
29 (defn start
30   [{:as game :keys [code]}]
31   (println "Welcome!")
32   (loop [current-game (assoc game :attempt 1)]
33     (cond
34       (game-over? current-game)
35       (println "You lost!")
36
37       (won? current-game)
```

```clojure
(defn won?
  [game]
  (= (:code game) (:guess game)))

(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      (won? current-game)
```

```
32  (loop [current-game (assoc game :attempt 1)]
33    (cond
34      (game-over? current-game)
35      (println "You lost!")
36
37      (won? current-game)
38      (println "You won!")
39
40      :else
41      (when-let [guess (ask-for-guess)]
42        (if (won? current-game guess)
43          (println "You won!")
44          (recur (play-game current-game guess))))))
```

```
32  (loop [current-game (assoc game :attempt 1)]
33    (cond
34      (game-over? current-game)
35      (println "You lost!")
36
37      (won? current-game)
38      (println "You won!")
39
40      :else
41      (when-let [guess (ask-for-guess)]
42        (if (won? current-game guess)
43          (println "You won!")
44          (recur (play-game current-game guess))))))
```

```
32  (loop [current-game (assoc game :attempt 1)]
33    (cond
34      (game-over? current-game)
35      (println "You lost!")
36
37      (won? current-game)
38      (println "You won!")
39
40      :else
41      (when-let [guess (ask-for-guess)]
42        (recur (play-game current-game guess)))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      (won? current-game)
      (println "You won!")

      :else
      (when-let [guess (ask-for-guess)]
        (recur (play-game current-game guess))))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (assoc game :attempt 1)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      (won? current-game)
      (println "You won!")

      :else
      (when-let [guess (ask-for-guess)]
        (recur (play-game current-game guess))))))
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (new-game game)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      (won? current-game)
      (println "You won!")

      :else
      (when-let [guess (ask-for-guess)]
        (recur (play-game current-game guess))))))
```

```clojure
(defn new-game
  [game]
  (assoc game :attempt 1))

(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (new-game game)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      (won? current-game)
      (println "You won!")
```

```clojure
(defn start
  [{:as game :keys [code]}]
  (println "Welcome!")
  (loop [current-game (new-game game)]
    (cond
      (game-over? current-game)
      (println "You lost!")

      (won? current-game)
      (println "You won!")

      :else
      (when-let [guess (ask-for-guess)]
        (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [{:as game :keys [code]}]
35   (println "Welcome!")
36   (loop [current-game (new-game game)]
37     (cond
38       (game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (new-game game)]
37     (cond
38       (game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (new-game game)]
37     (cond
38       (game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (core/game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (core/game-over? current-game)
39       (println "You lost!")
40
41       (won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (core/game-over? current-game)
39       (println "You lost!")
40
41       (core/won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

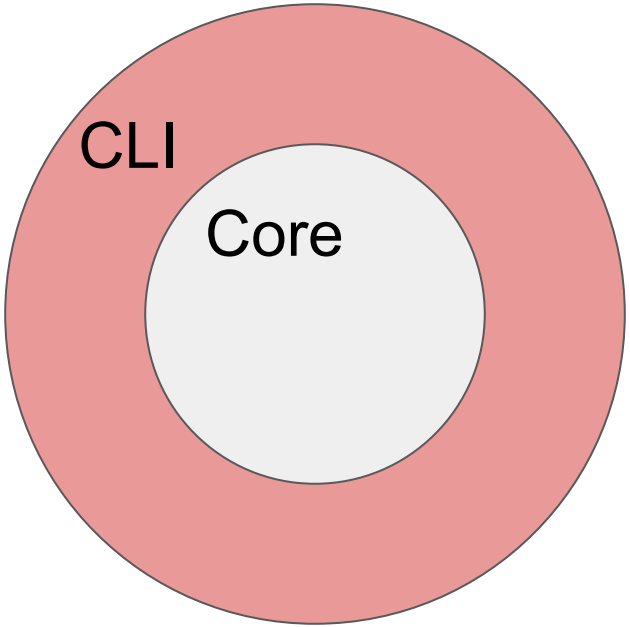
```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (core/game-over? current-game)
39       (println "You lost!")
40
41       (core/won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (play-game current-game guess))))))
```

```
33 (defn start
34   [game]
35   (println "Welcome!")
36   (loop [current-game (core/new-game game)]
37     (cond
38       (core/game-over? current-game)
39       (println "You lost!")
40
41       (core/won? current-game)
42       (println "You won!")
43
44       :else
45       (when-let [guess (ask-for-guess)]
46         (recur (core/play-game current-game guess))))))
```

```
 1 (ns codebreaker.cli
 2   (:require [codebreaker.core :as core]))
 3
 4 (defn prompt []
 5   (print "> ")
 6   (flush))
 7
 8 (defn ask-for-guess []
 9   (prompt)
10   (read-line))
11
12 (defn start
13   [game]
14   (println "Welcome!")
```

```
 1 (ns codebreaker.core)
 2
 3 (defn game-over?
 4   [game]
 5   (< 8 (:attempt game)))
 6
 7 (defn won?
 8   [game]
 9   (= (:code game) (:guess game)))
10
11 (defn play-game
12   [game guess]
13   (-> game
14       (update :attempt inc)
```

Done!



EQUAL EXPERTS

What We've Learned

EQUAL EXPERTS

What We've Learned

Accidental complexity needs to be controlled.

EQUAL EXPERTS

What We've Learned

Accidental complexity needs to be controlled.

High quality software is cheaper b/c it decreases complexity!

EQUAL EXPERTS

What We've Learned

Accidental complexity needs to be controlled.

High quality software is cheaper b/c it decreases complexity!

Feedback loops help us to move faster (refactoring without fear).

EQUAL EXPERTS

What We've Learned

Accidental complexity needs to be controlled.

High quality software is cheaper b/c it decreases complexity!

Feedback loops help us to move faster (refactoring without fear).

Architecting separation of concerns make changes easier.

EQUAL EXPERTS

Thank you!

@jivagoalves

Equal Experts is hiring!

EQUAL EXPERTS