Building a small-scale multiplatform automated software testing facility

Maxim Mozgovoy

The University of Aizu

mozgovoy@u-aizu.ac.jp

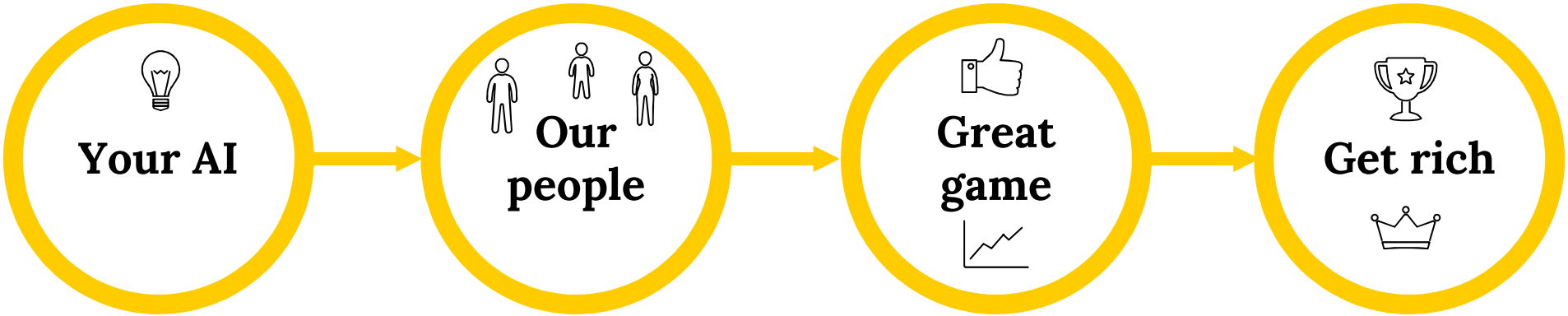
Hello!

Primary area: *human-like AI for computer games and simulations*.

“AI that needs to possess other qualities rather than being good”

Also I have strong interest in practical software development and previous industrial experience

My friend’s idea



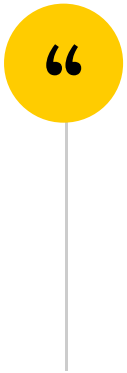
We made it (to some extent!)

A cool casual tennis game

- Released on Win, Mac, Android, and iOS
- Around 1mln downloads
- Really good reviews

- Released later than hoped
- Profits aren't stellar yet

Nothing works better than just improving your product.
— Joel Spolsky



What's special about game projects?

- Users won't tolerate bugs [1]
- Negative reviews & crashes cause downranking [2]
- Low-rank apps have no chance on the market

Frequent complaints	%
Functional error	26.68
Feature request	15.13
App crash	10.51
Network problem	7.39

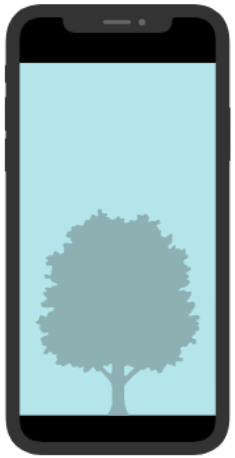
What's special about game projects?

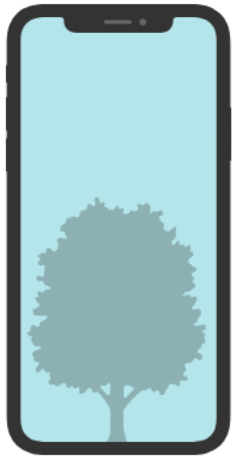
They are quite complex (really):

- Frontend/backend.
- User analytics.
- Billing/transactions.
- Integration with numerous 3rd-party systems (social media, advertisements, online profiles)

What's special about game projects?

- They rely on (unstable) 3rd-party libraries and tools
- They have to be updated regularly



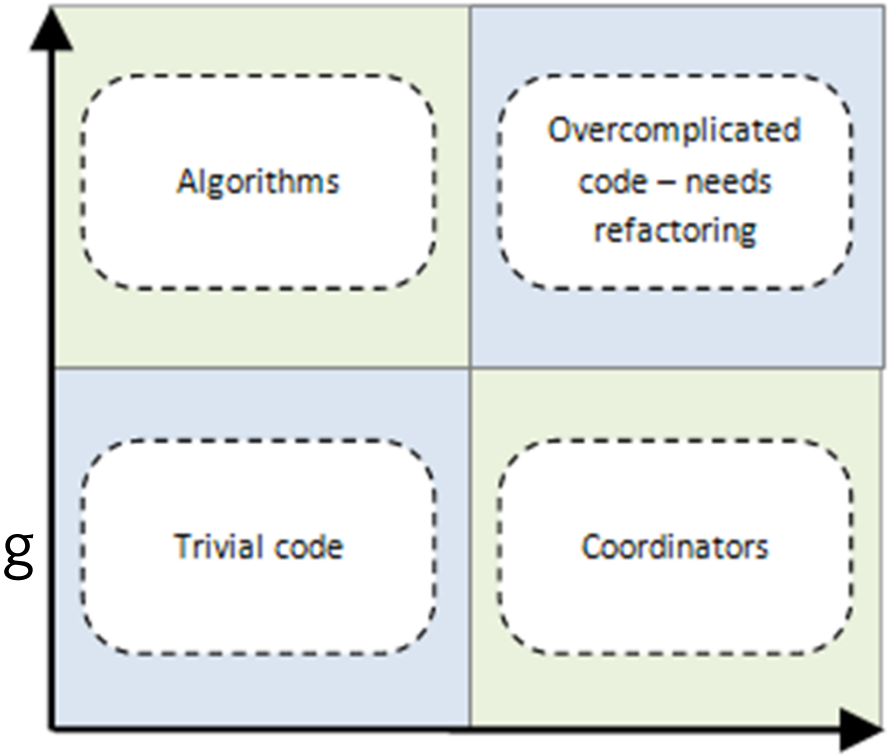


What's special about game projects?

- They must run on diverse hardware and software platforms
- They are prone to issues that are hard to test automatically (graphics, sound, animation)
- GUI and animation is deeply integrated into a game

What's special about game projects?

They contain a large proportion of code with high cost of unit testing [3]



Our approach

Primary emphasis on
extensive testing of the *complete* game

Synergy of diverse tools

Tool #1: Firebase Crashlytics

Embed Crashlytics reporting service into the app

Crashes are automatically reported to us
via Internet (along with stack traces)

Identified devices with insufficient RAM for the game

Tool #2: Autobugs and Manual Bugs

Use soft (reporting) asserts and manual reporting tools

Report failed non-fatal asserts automatically.
Give the users and testers tools to report easily.

Got numerous bug reports
(wrong physics, animation, GUI flaws, etc.)

Tool #3: Automated smoke testing (autotests)

Ensures that the most important subsystems work by preforming various simple test scenarios

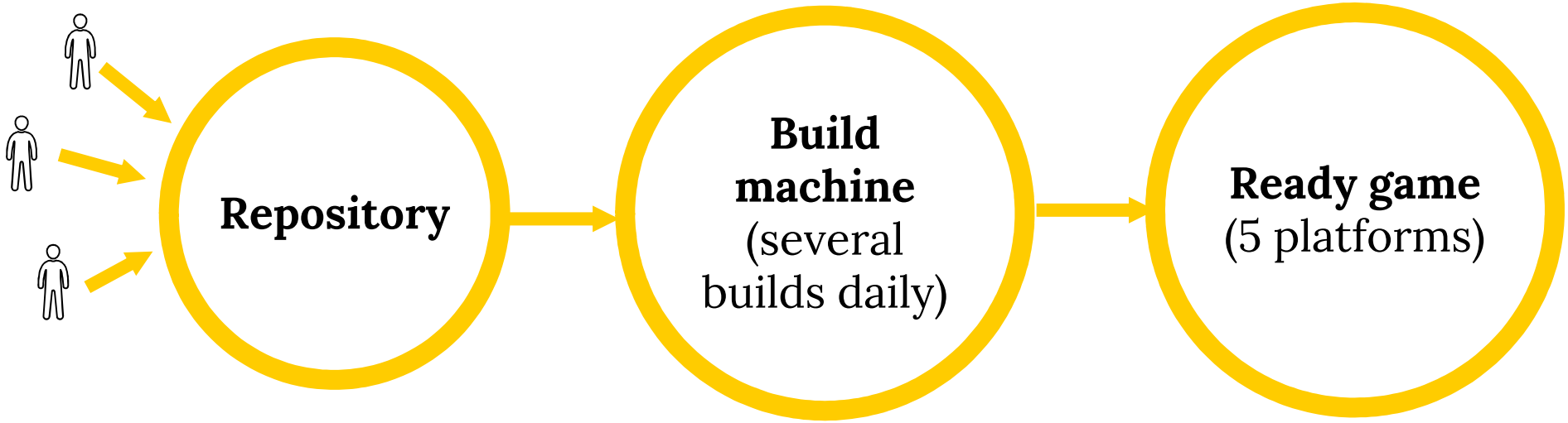
The most cost effective method for identifying and fixing defects in software [after code reviews]

— *Microsoft*

Our smoke tests

We invested a lot of effort
into building our own
smoke-testing facilities,
and now we find them essential
for subsequent planned projects

Our pipeline



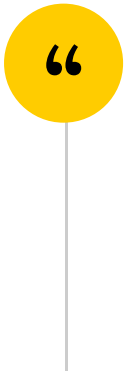
Basic levels of QA

- The game can be compiled (*ensured by the build machine*).
- The game doesn't crash on startup
  (it might due to fatal bugs or incorrect partial build)
- The game can connect to our backend server.

Early detection is crucial: we need to know
which changeset in our repo causes problems.

Let's just test that the game doesn't crash and is able to go online

– Our project manager



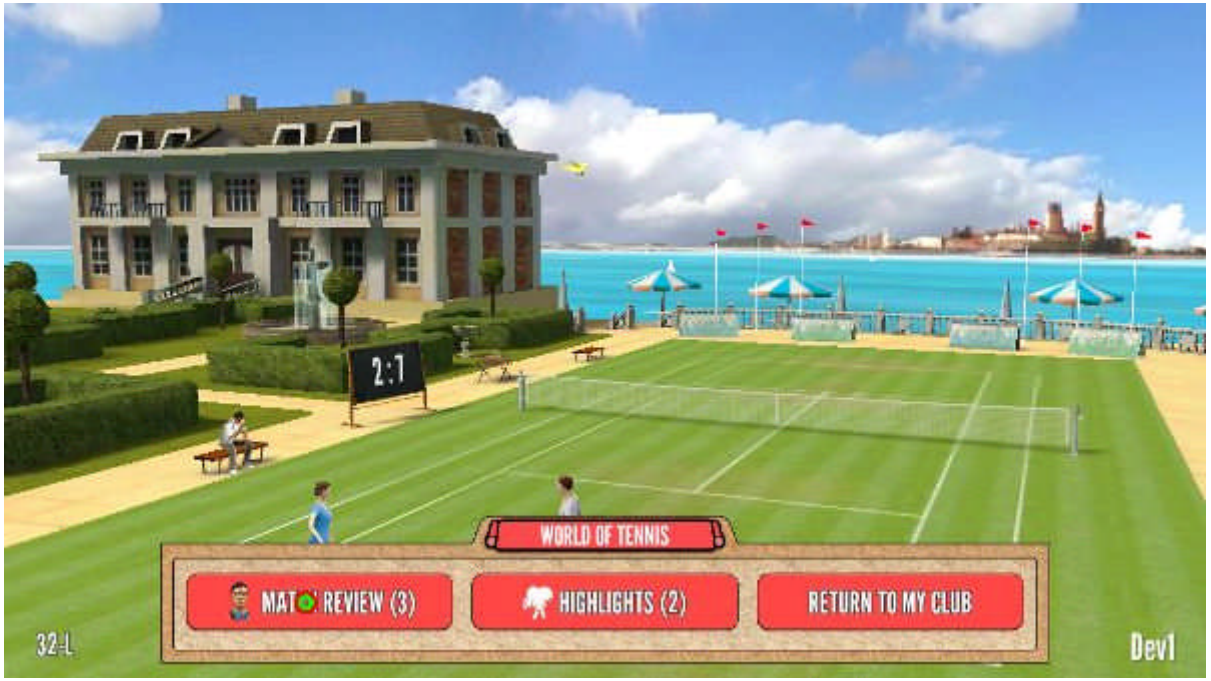
One year later

Every build goes through six test scenarios; each scenario takes 30–60 min to complete

Points to consider

- The game has no “hidden interface”: the testing system relies on the ordinary user-end UI.
- Note the synergy: automated tests also generate autobugs and crash events, detectable by our other tools!
- Autotests are run on three Android, three iOS and one Windows devices (macOS is planned).
- Autotests also report FPS and memory consumption; They can be run for several hours as *stress tests*.

Example scenario



Other scenarios:

- Pass tutorial.
- Customize player, upgrade skills.
- Link a Facebook account

Farming and scripting

How to achieve it?

- You'll need to write test scripts.
- You'll need to have a *device farm* for running tests.

The easy way

Use existing device farms offered by Amazon, Bitbar, etc.

- **Pros**: easy setup, thousands of devices.
- **Cons**: quite expensive (~15 USD per minute per device) (in our case it translates into ~250 USD daily), device choice is still limited.
- **Notes**: you'll have to rely on platform-supplied scripting (before writing scripts one must choose the platform).

Our way

Let’s build a device farm ourselves!

Own farm: pros and cons

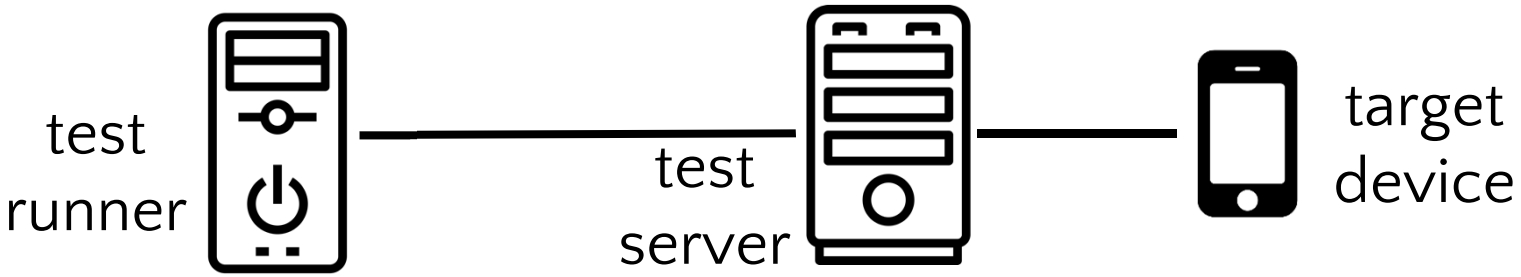
- Flexible: we can choose any devices we need.
- Inexpensive (in the long run).

- Limited to few specific devices.
- Requires regular maintenance.

Minimal setup

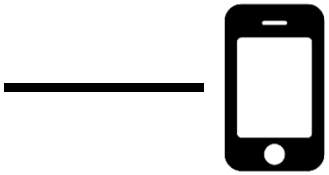
Logically, there are three components involved:

- A device executing the test scripts (runner).
- A device interfacing with the target platform (server).
- A target device running our game.



Software setup

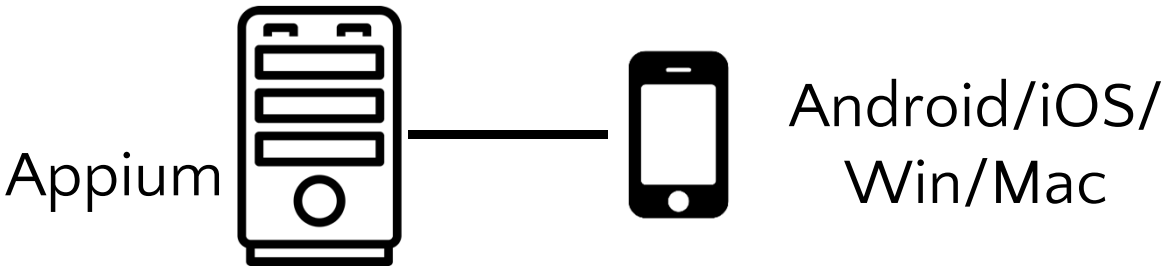
Testing capabilities are available on all major platforms. No software is necessary, but some configuration is required.



Android/iOS/
Win/Mac

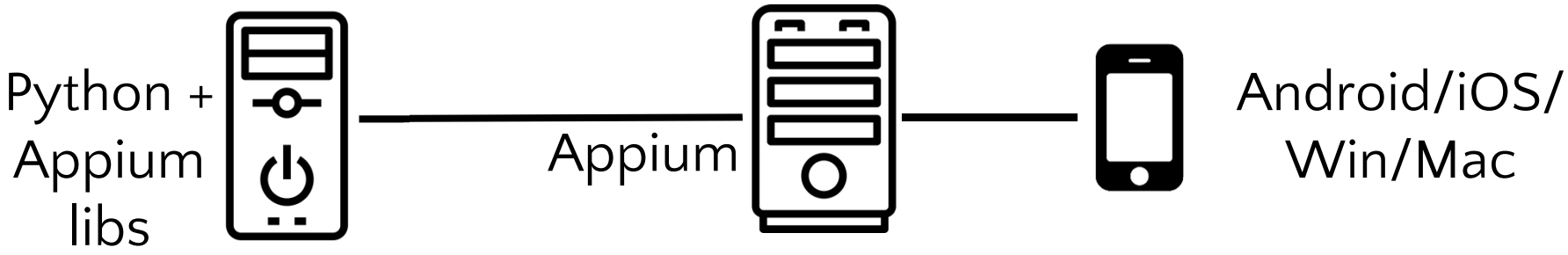
Software setup

Test server must run a 3rd-party testing framework.
We chose open-source Appium (https://appium.io).



Software setup

Test runner executes scripts written using a conventional programming language supported by the framework.



How test scripts look like?

Technically, they consist of code like this:

```
e = appium.find_element_by_class_name('android.widget.EditText')
e.send_keys("hello") # type “hello” into the EditText control

ok = appium.find_element_by_class_name('android.widget.Button')
ok.click() # click the first button on the screen
```

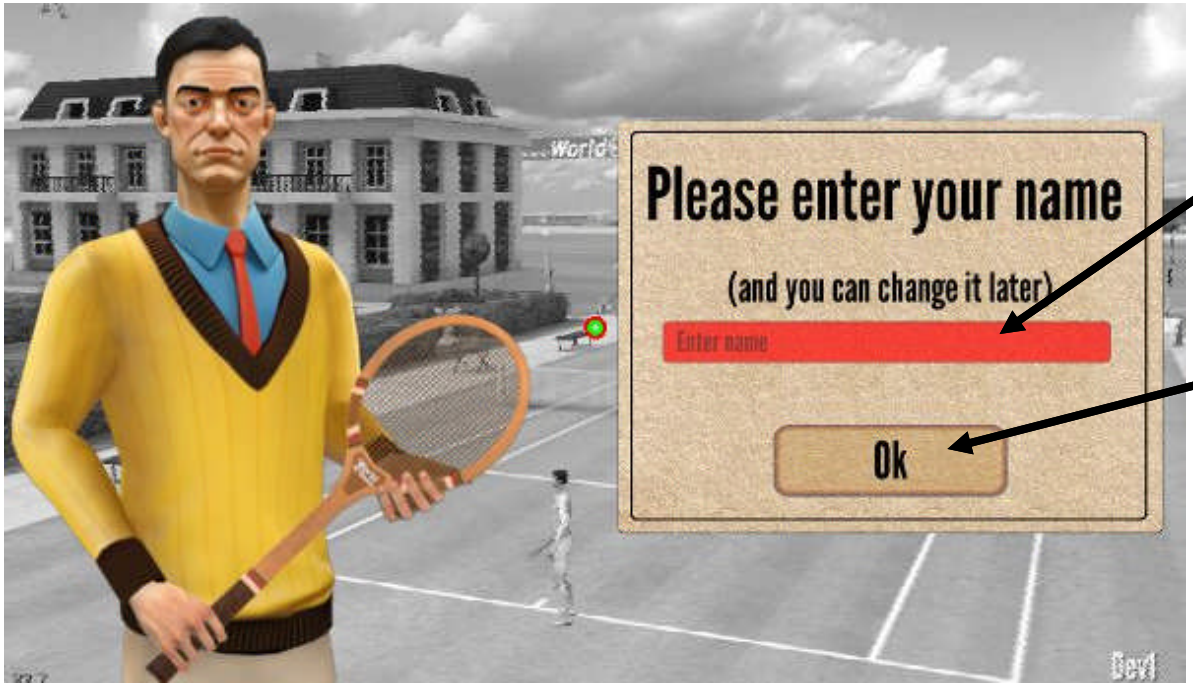
How test scripts look like?

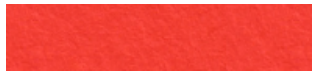
In our game most GUI elements are drawn on the screen surface and thus not considered “UI” by the operating system.

Thus, we use image recognition:

```
ib_loc = find_image("input_box.png") # fail test if not found
click_location(ib_loc)
ok_loc = find_image("ok_button.png") # fail test if not found
click_location(ok_loc)
```

How test scripts look like?





input_box.png

ok_button.png

OpenCV

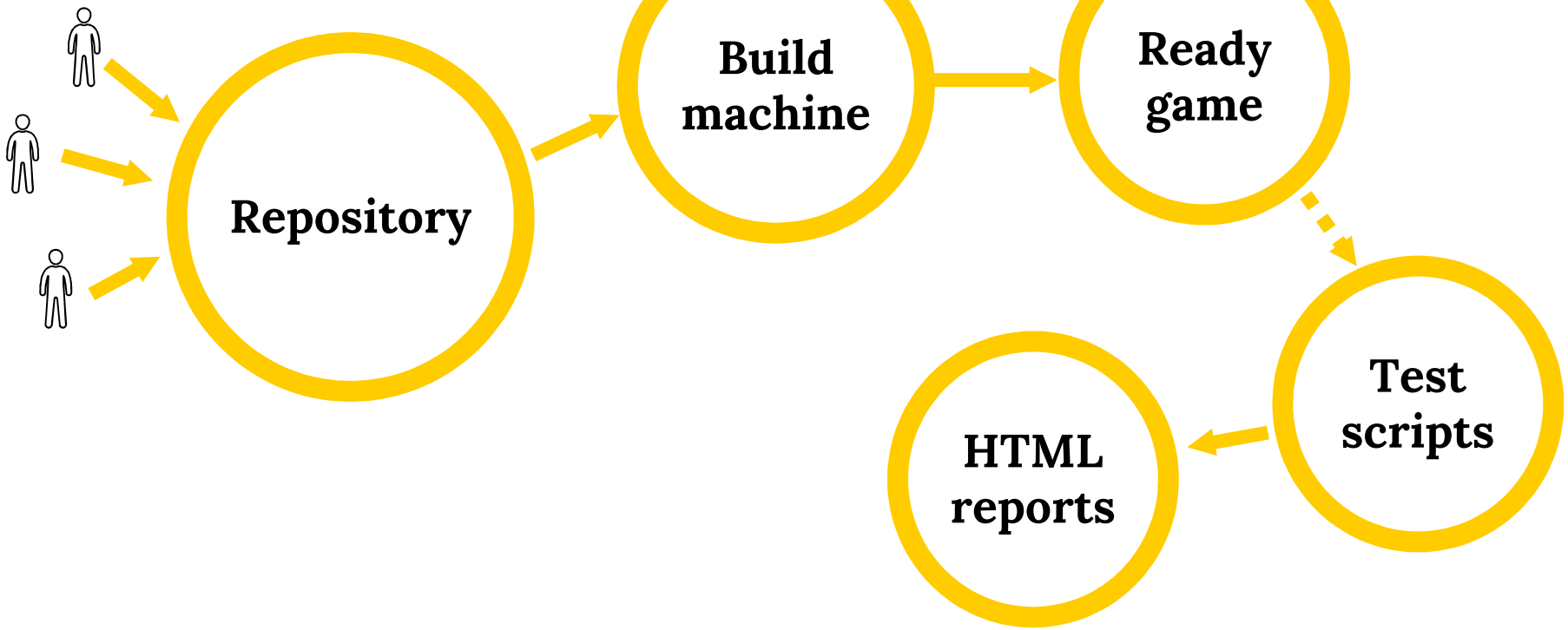
How test scripts look like?

Our test scripts are integrated into the whole build process:

- A test script checks whether a new build is available.
- If yes, this version is tested with a set of scenarios.
- Results are summarized and published as an HTML report.
- The process is repeated.

- HTML reports are used by the testers to check visual glitches and understand why certain tests fail.

Our pipeline



Complete hardware setup

Test runner:

Zotac ZBox MI
(Windows)

Test server 1:

Zotac ZBox E
(Windows)

Test server 2:

Mac Mini

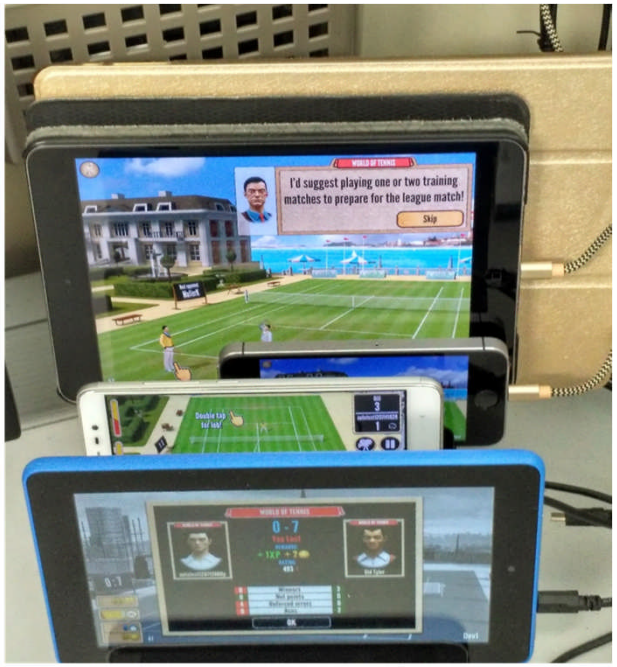
3 Android
devices

3 iOS
devices

Complete hardware setup

Servers

Mobile devices



Tips, tricks, and notes

Farming isn’t easy!
There are numerous pitfalls...

Choosing test devices

When choosing devices, we tried to focus on hardware diversity and get some low-level models.

- Sometimes code fails on certain hardware (it happens, e.g., on different GPU chips)
- Developers usually have reasonable hardware, so autotests help to make sure the game is still runs smoothly on low-end devices.

Choosing USB hubs

- Mobile devices are connected to a server via USB hubs.
- Devices should get power through the hubs
  (otherwise they will simply discharge).
- Surprisingly, it is very difficult to find a hub,
  able to charge several attached devices fast enough!

Quirks of particular devices

- Installs the app after several attempts.
- Doesn’t unlock the screen.
- Doesn’t want to charge from a USB hub.
- Asks for regular updates, blocking tests (iOS)

It is still worth the effort

The safety net feeling we have now brought peace to our lives

Thanks!

[1] A. Moscaritolo (2017). Google Play Now Favoring 'High-Quality Apps'. *PC Magazine*, https://www.pcmag.com/news/355375/google-play-now-favoring-high-quality-apps
[2] H. Khalid et al. (2015) What Do Mobile App Users Complain About? *Software*, vol. 32, pp. 70-77
[3] S. Sanderson (2009). Selective Unit Testing – Costs and Benefits. https://blog.stevensanderson.com/2009/11/04/selective-unit-testing-costs-and-benefits/

Contact me:
mozgovoy@u-aizu.ac.jp