TYPEFIGHTER

Gameplay Features & Technical Design

Serious Game Project at JMU University of Würzburg, by

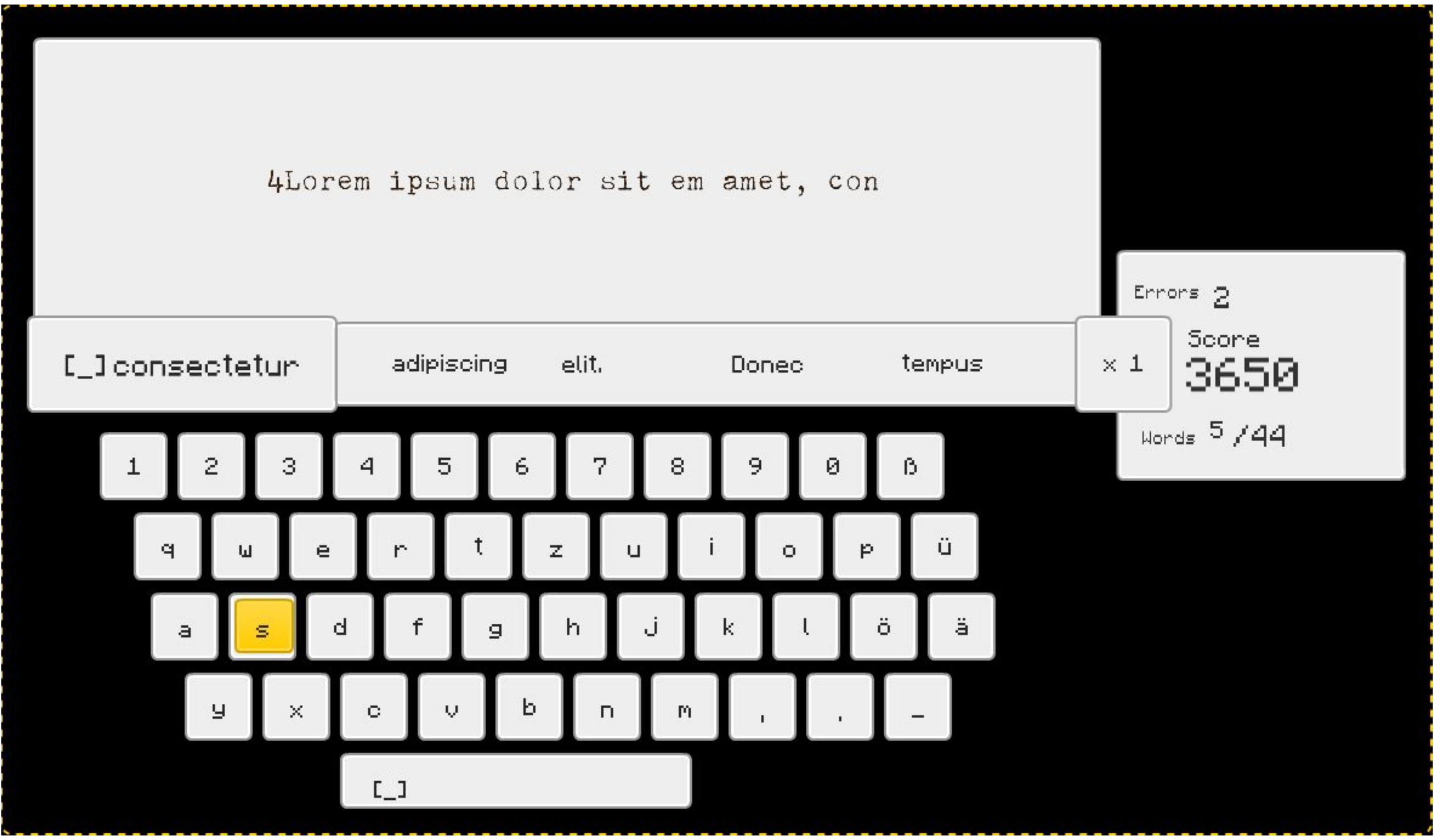
Lena Lang

Jessica Topel

Lydia Bartels

Marlon Franz

Benedict Bihlmaier


4Lorem ipsum dolor sit em amet, con
[_]consectetur
adipiscing
elit.
Donec
tempus
x 1
Errors 2
Score
3650
Words 5 /44
1
2
3
4
5
6
7
8
9
0
ß
q
w
e
r
t
z
u
i
o
p
ü
a
s
d
f
g
h
j
k
l
ö
ä
y
x
c
v
b
n
m
,
.
_
[_]

Purpose

This document shows the various features of the serious game TYPEFIGHTER and demonstrates the reasoning behind important design decisions.

Most claims and arguments stem from an understanding of Nielsen’s 10 heuristics of Usability, as well as a young lifetime of playing video games and a few years experience in analyzing, debating and making them.

Game Overview

High Concept & Justification

Typefighter is a gamified, responsive typing-trainer, for web, where high skill is rewarded with high scores. Gather achievements and see how you’re doing compared to your friends or the whole playerbase. Choose a text and start typing!

The term gamification (“gamified” above) implies high interactivity and agency, supported by positive reinforcement. Any input returns an immediate, highly perceptible (visible and audible) response. The player should feel motivated to to get better, rather than stressed by a lack of skill. They should feel that playing the game is the most enjoyable way to improve a hard skill that is applicable in everyday life and work. Gamification injects deeper meaning and emotional investment into everyday decisions.

In our case, a training tool, gamification demands that we favor positive reinforcement over negative feedback loops to keep the player interested in improving.

Text in lyrical and prose form is a very natural and compelling foundation for what in other games would be a “Level”, as it usually is designed with pacing in mind. The design challenge is the gameplay reflecting the pacing of any input text, but there is no need for additional level design or creation.

Features

1. Live Evaluation and Guidance

Concept

As opposed to letting Users type in a text and evaluating when they’re done, input is evaluated immediately. Via UI, the user can always see which input is required to advance.

In practice, we display live information on …
… Current score & combo-multiplier

… Text as it is written
… Which key to press next
… Which word is currently being evaluated
… Which words are to be written in the future
… How many words are left to complete the text
… How many wrong keypresses were made so far

Reasoning

The user can see how they are doing at any time during the game. They can see how each interaction affects their progress and immediately understand a mistake they made.

One of the advantages of using the Phaser game engine is the technical availability of this data as it is generated. It is used to technically evaluate how the player is doing, and to time gameplay events correctly.

2. Scoring & Combo system

Concept

Each time a word is completed, score increases based on the length of the word. Score gained is modified by a combo-multiplier, which increases each time a series of words were completed without mistakes.

As a wrong keypress is registered, the combo-multiplier resets, and the current word has to be written from the beginning.

The final score adds another bonus, which is based on the amount of time it took to finish the text, in relation to the length of the text.

Reasoning

Score can only increase, but never decreases. Negative feedback loops are avoided.

The time-bonus encourages writing fast, while the combo-system encourages being careful, which makes for an interesting trade-off and overall improves the player’s typing ability.

3. All skill-levels encouraged

Concept

The Keyboard used to write is represented in the UI, as buttons laid out to reflect a real keyboard. A uniquely colored cursor hovers above and moves from key to key, so players can see which button the have to press on their according keyboard.

The current word and upcoming words are displayed above the keyboard, so that players that memorized key locations can rely on their memory of the key-layout and the text to write faster.

The time-bonus on the final score encourages learning to be faster, while the majority of score-points still come from not making mistakes. Even perfect texts are improved based on the time it took the player to write them.

Reasoning

Almost everyone can read, so we should expect almost anyone to use our game as a training tool. The same UI enables us to pose a challenge to people who write all day, while Users who rarely see keyboards are encouraged to improve, as any good serious game should. We expect total beginners to use the keyboard UI and key-cursor to find the correct buttons, while adept typewriters, who know their keyboard, focus on the current and upcoming words. This idea is adapted from newer versions of Tetris, where future blocks and the order they will arrive in are displayed next to the playing field. Looking at footage of the game being played on the highest level, it’s easy to see that the skill-ceiling is only this high because players know what’s coming. On the other hand, if you’ve never played Tetris before, you will find that you struggle to decide where to put the block that’s already falling.

4. Responsive UI - Visuals

Concept

To guide the player and differentiate which information is important at any time, the various UI elements have different sizes: the current word and score texts are the largest text-elements on the screen.

The layered UI implies a hierarchy on what should be payed attention to before looking at other elements: current word, key-cursor and combo-multiplier are all top level-layer, as the provide the most precise information on how the player is doing and which exact challenge they face at this moment.

Furthermore, the key-cursor has a unique color to make it stand out on top of the grey keyboard.

We indicate which key to press next via a cursor, which jumps to the next key on the UI-keyboard.

Each keypress bumps the corresponding key on the UI-keyboard downward, imitating a pressed typewriter-key which jumps back up from a spring.

Each written letter appears on the writtenText-Panel as it is typed, while the panel gets a very subtle bump to further enhance the sense that the pressed (and bumped) key just had a meaningful effect on the environment and the game progress.

As a word is completed, the upcoming words move into the current word panel, the new current word changes and the upcoming words update. The motion supports the natural left-to-right reading direction.

As points are scored, the combo-multiplier panel bumps into the score-text on the score-panel, to give the impression that the combo affects the score increase.

Wrong keypresses bump the score-panel down, as the error-counter increases.

Reasoning

Visual feedback is essential for creating agency. The user can literally see the effect of their input-action and evaluate the decision behind it. Furthermore, movement can be used to draw attention to changing points of interest and helps user-orientation in an otherwise cluttered environment.

Bumps, shakes and targeted movement of individual elements (or the whole screen) are a powerful tool to create an emotional impression with the otherwise forgettable task of recreating text.

5. Responsive UI - Audio

Concept

Each registered keypress is answered by one randomly chosen of three type-writer typing sounds.

A wrong keypress triggers a chime-like “ding”-sound. It’s repetitive and annoying, so the player hopefully wants to avoid hearing it..!

Score increase triggers an affirming cue.

Combo increase triggers an uplifting melodic cue.

A combo drop (first time wrong keypress) triggers a throttling melodic cue.

Finishing the text triggers a glitchy explosion sound. This effectively covers up a missing visual transition to the post-game score screen.

Reasoning

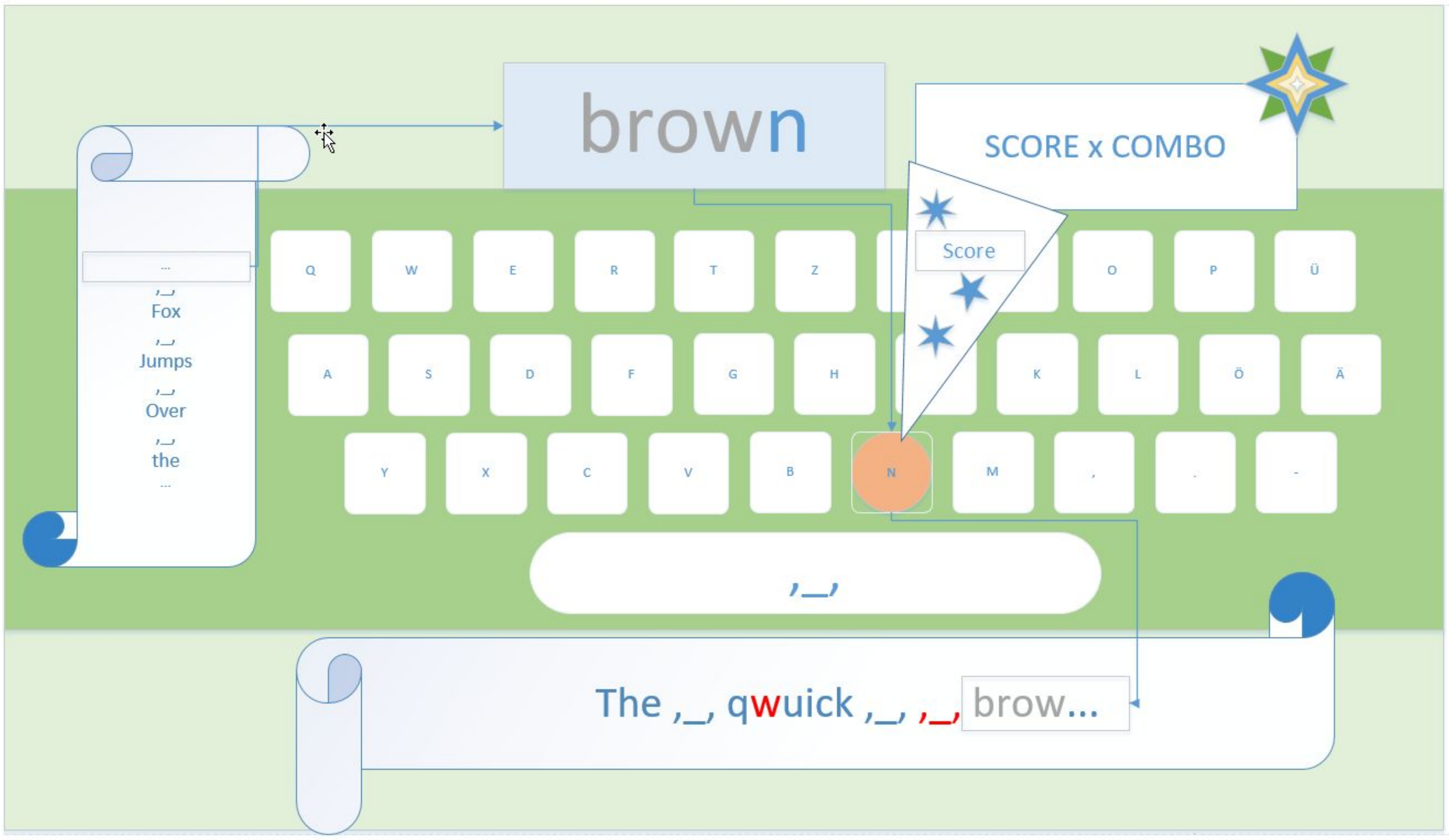
Audio feedback is processed faster than visual feedback and alleviates a one-dimensional cognitive burden. Giving understandable sound-cues at the right time, consistently, can drastically improve the training effect this game aims for. The player uses less of his attention on looking for the visual indicators and parsing them.

Generally, melodic cues are universally interpreted in a similar way, such as a minor-chord sounding depressed, compared to a major-chord sounding joyful, which makes them very suitable to attach emotional meaning and valence to an event.

Game Design

In Typefighter, the player chooses a short text, of up to 80 words, and re-creates it by typing it word for word. The score and combo-system reflect how the player is doing and ultimately rewards being both precise and fast.

The interface itself should explain the rules of the game. The initial Design was based on a single mock-up:



We see a keyboard, a cursor on the current button to press, the current word to write the output text, a score panel and the upcoming words. Moreover, letters are colored in a way that indicate whether the current word is finished, which symbols were input correctly and where mistakes were made. The cursor moves to the next correct key after each correct input.

Score increases by 100 * (length of the finished word) * (combo-multiplier), each time the current complete word has been input correctly.

A single keypress which doesn't match the currently expected symbol is considered a mistake. Upon a mistake, the current word has to be written again from the beginning, after a spacebar input. In fact, the spacebar input is always the first symbol of the current word.

The combo-multiplier has a base value of 1. It increases by .25 each time the player completes 3 words without mistakes. It is reset to 1 as the player makes a mistake.

After the text is completed, a time bonus is added to the final score. The time bonus is calculated as:

```
Math.max(0 ,scorePoints - Math.floor(RequiredTime/1.368));
```

- scorePoints is the amount of points gained by typing
- RequiredTime is the time difference of the timestamps of the first and last keypresses in ms.

This formula ensures that the bonus score is greater with decreased required time. It scales with the score acquired by typing, which in turn scales linearly with the length of the text, so that the time-bonus is a similar fraction across all played levels by a single player, as it heavily reflects that players skill. There can be no negative bonus.

Technical implementation

Typefighter is a web-application that utilizes the Phaser library for fast rendering responsive gameplay. This means, the game relies on JavaScript ES6, which is a powerful, widely used functional scripting language.

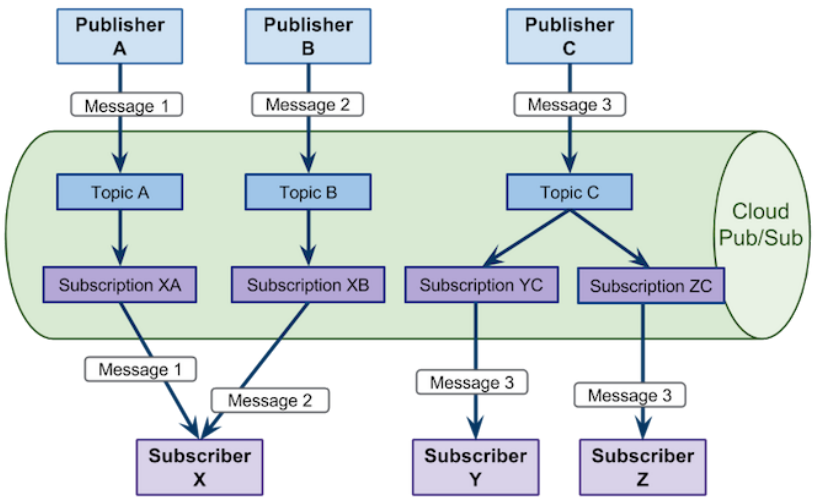
Video games typically rely on an object-oriented framework and lightweight scripting. Object orientation usually yields high maintainability for complex projects and offers great optimization tools, which are important for real-time rendering and frame-based loops. Since Typefighter has to run on JavaScript, which does not support traditional object-orientation, we want to keep a small scope for the game code.

The Phaser game library offers a robust and proven backend to conveniently use the real-time rendering capabilities of modern browsers (WebGL & Canvas) for games. It offers simple graphics and sound APIs, state management, input-handling, asset management and Plugin-support.
Richard Snijders' SlickUI plugin is used to reduce the amount of code needed for dynamic UI-Elements-grouping, so that most code is written for actual functionality.

Structure

PubSub pattern (gameEvents)

The game-logic is built around the PubSub JavaScript-pattern. It uses an object (gameEvents), which registers strings (events or Topics), and maps function objects to those strings (Subscriptions). Other objects can then Publish events - which effectively calls all functions registered to a specific event-string - and pass objects along each function call. This pattern allows a highly modular structure, where objects do not have to know anything about each other and still be highly interactive.



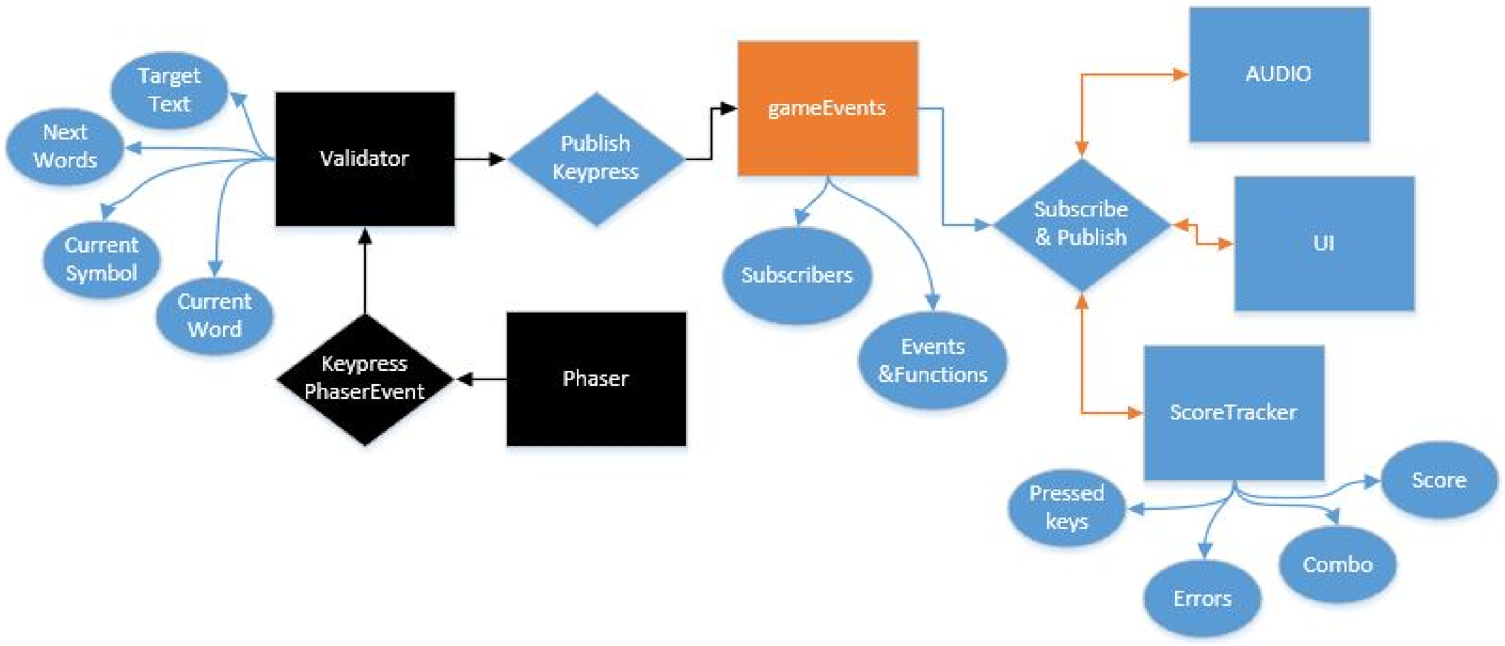
Source: https://tctechcrunch2011.files.wordpress.com/2015/03/many-to-many.png

Setup & Flow in practice

The entry point for our Game code is the browser's window.onload callback, which is run after all other resources and scripts are loaded. There, we create our game-instance, add each state and start the play state.

Each state in Phaser runs a preload, then a create function. Preload is used to load each state's required assets, then create initiates our UI and Audio modules, which each set up their own necessary objects and event-handlers (via init_state functions); Our main Game module also subscribes to keyboard input with phasers own input system.

On any keyboard-keypress registered by Phaser, the Game module emits the Keypress event via our gameEvents object. In practice, this triggers a cascade of other events, where the various objects validate input, manage scoring, update the UI, start tweens and run audio cues.



Code Quality

Naming and structure conventions are borrowed from object-orientation practices, in order to increase readability for JavaScript-beginners like us. Components are extended, custom JavaScript Prototypes. The general structure of any Component object looks as follows:

Initiation-functions are at the top. Most objects call this property create(), which returns the object itself. Inside create(), the object subscribes to the various gameEvents and sets up any other properties that aren't functions using the this keyword.

The Methods section contains function objects that are called repeatedly. They manage the non-function properties, as well as emitting events as needed.

The Properties section contains function objects which return information about the state of the component.

Classes are not really classes - they're a syntactical shortcut for Prototypes. They're used for prototypes that need more than one instance, because they make handling *this*-context in these cases less of a pain, and the new-keyword is easier to read than the create-syntax.

Centralized globals

For centralized modification, the Game_globals.js script holds important variables, like instances of our modules and central components, and constants, like asset paths, which can be accessed by JavaScript globally.

Core Component - Validator

The Validator object validates keyboard input and decides the sequence of PubSub-events. It manages in detail the progress in writing the target text.

Core Component - scoreTracker

The scoreTracker keeps track of which keys where pressed when and manages the various scoring-variables. It publishes events that mainly concern UI and Audio components. The final game score is calculated and published here.

UI

UI.js is a collection of components which are created by the main UI-module on game create. They each represent an element in the graphical interface and respond to various events with rendering-updates and animations. They are all very front-end, compared to the Game.js components, and thus don't affect the game logic. This means magic numbers are not problematic here as they are in fact heavily used for rapid iteration.

Audio

Audio is likely the most simple module: it loads soundfiles and plays them at the right time. Again, magic numbers are used, mainly for balancing sound volume.