Purposeful testing with Jest 🃏

twitter.com/ittordepam
github.com/pedrottimark

speakerdeck.com/pedrottimark/testing-with-jest-at-ims

Let's start with the ~~bug~~ big picture.

To get ready for the small picture,
clone or fork, and then install:

https://github.com/pedrottimark/whimper

The main reason to write tests is to ensure that your app works the way it should.

Test the high-value features.

You click an “Add to Cart” button.

The app had better add that item to the cart.

https://daveceddia.com/what-to-test-in-react-app/

When you test React components:

- Given properties and state, what structure?
- Behavior or interaction: is there a possibility to transition from state A to state B?

http://reactkungfu.com/2015/07/approaches-to-testing-react-components-an-overview/

Writing tests defines your component’s contract.

- From an outsider’s perspective,

  is this detail important?
- Don’t duplicate the application code.

https://medium.com/@suchipi/the-right-way-to-test-react-components-548a4736ab22

Communicate when you write a test.

- Correct: now of course, but is it practical to keep test correct when code changes?
- Clear: where to fix code when test fails
- Complete: fewer tests that fit your priorities for quality-scope-cost are better than more tests that don’t

Why test with Jest?

Fast and sandboxed testing with Jest

- Jest parallelizes test runs across workers.
- jest --watch runs tests only for changed files.
- Jest resets global state for each test

  so tests don’t conflict with each other.

Zero configuration testing with Jest

Jest is already configured

if you create a project with

- create-react-app
- react-native init

🙌🏾

Zero configuration testing with Jest

- Provides assertions from expect library.
- jest --env=jsdom to simulate DOM.
- Supports mocks for functions, timers, modules, and React Native components.
- jest --coverage to report code coverage.

Zero configuration testing with Jest

Jest finds test files

- in any __tests__ folder — like Facebook ✔
- with .spec.js extension — like Jasmine ✘
- with .test.js extension — unlike Facebook ✔

Learning resources for Jest

- Using Matchers and expect
- Snapshot Testing
- Testing Asynchronous Code
- Configuring package.json
- Jest CLI options

Other devDependencies

•	react-addons-test-utils	peer dependency
•	enzyme	shallow, mount, render
•	enzyme-to-json	for subset or snapshot
•	react-test-renderer	for subset or snapshot

enzyme

shallow(element) returns a wrapper around the rendered output, as in jQuery. Render component one level deep, to test it independent of <u>how</u> children are implemented. Traverse, optionally interact, and assert.

enzyme

mount(element) also returns a wrapper.

Render to full depth in simulated DOM.

- traverse: .find(selector).at(index)
- interact: .simulate(event)
- assert: .prop(key) or .state(key) or .text()

enzyme-to-json and serializer

- shallowToJson
- mountToJson

  mountToDeepObject proposed

  mountToShallowObject proposed
- renderToJson

react-test-renderer

- Render without simulated jsdom.
- Render without wrapper div.
- Render React elements as test objects compatible with toMatchSnapshot assertion.
- Doesn’t yet support interaction with DOM.

react-test-renderer

• renderer.create(element).toJSON()	
• renderAsTestObject(element)	proposed
• relevantTestObject(element)	proposed
• irrelevant	proposed

Beware of “one size fits all”

- “Use shallow rendering!”
- “Use snapshot testing!”

Instead, select a tool to fit your goal

Instead, select a tool to **fit your goal**

If you apply patterns and follow examples

for 20% of tests that occur 80% of the time,

then you save 80% of your time and energy

for <u>difficult tests</u> that occur 20% of the time.

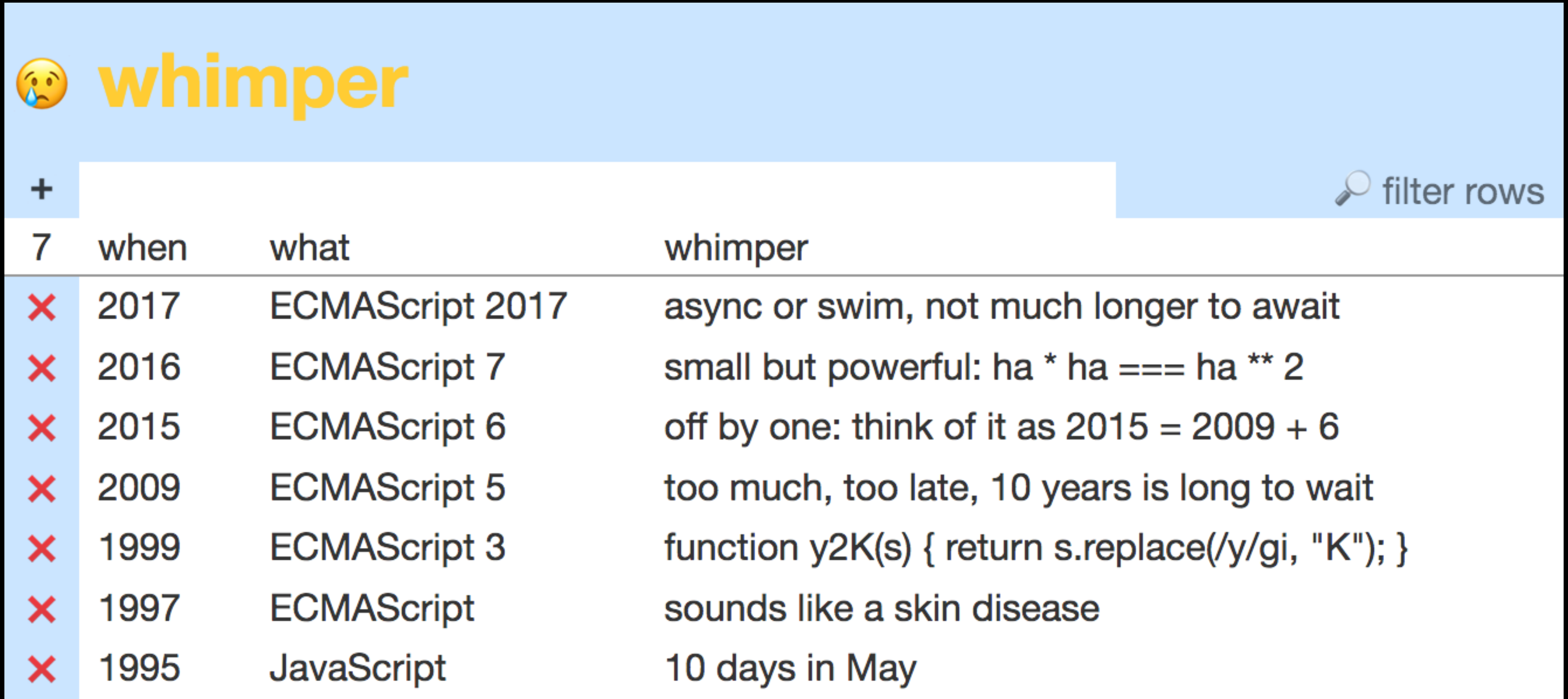
Let’s move to

a smaller picture.

Patterns for operations	Examples of tests
• Read or render	TableHead-R.test.js
• Interact	TableHead-I.test.js
• Create	Table-C.test.js
• Delete	Table-D.test.js
• View	Table-V.test.js
• Update or undo	Table-U.test.js

Examples of tests for whimper based on Whinepad from *React Up & Running*

https://github.com/pedrottimark/whimper



😢 whimper

+ filter rows

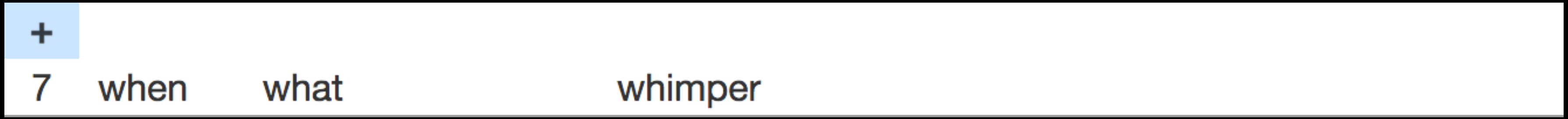
7	when	what	whimper
✕	2017	ECMAScript 2017	async or swim, not much longer to await
✕	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✕	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait
✕	1999	ECMAScript 3	function y2K(s) { return s.replace(/y/gi, "K"); }
✕	1997	ECMAScript	sounds like a skin disease
✕	1995	JavaScript	10 days in May

Read or render

Given combinations of props and state as input, the component renders correct output:

- what people “see” including accessibility
- what child components receive as props

Read or render example: table head



+			
7	when	what	whimper

This first example contrasts two methods

- baseline: typical “abstract” assertions 😐
- proposed: the toMatchSnapshot assertion 😊

  matches expected props and descendants

  in “descriptive” JSX

Read or render baseline tests 😐

TableHead renders a button and text.

- simulate child components or DOM nodes
- traverse by selector
- assert each expected value,

  but it can be hard to see too many criteria

Read or render baseline tests 😐

```
// src/components/__test0__/TableHead-R.test.js

import React from 'react';
import {shallow} from 'enzyme';

import TableHead from '../TableHead';
```

```
describe('TableHead', () => {
  it('renders button, row count, and fields', () => {
    const $it = shallow(
      <TableHead
        addRow={() => {}}
        count={7}
        fields={fields}
      />
    );
    expect($it.find('button').length).toBe(1); // add row
    expect($it.find('tr').at(1).find('th').at(0).text()).toBe('3');
    // and so on for field labels in column headings
  });
});
```

Read or render proposed tests 😊

A few snapshots which control changes

to a component do more good than harm,

- because it's easy to see descriptive criteria,
- if you know that's their goal,
- from the name of the test file.

Read or render proposed tests

```
// src/components/__tests__/TableHead-R.test.js

import React from 'react';
import renderer from 'react-test-renderer';

import TableHead from '../TableHead';
```

```
describe('TableHead', () => {
  it('renders button, row count, and fields', () => {
    expect(renderer.create(
      <TableHead
        addRow={() => {}}
        count={7}
        fields={fields}
      />
    ).toJSON()).toMatchSnapshot();
  });
});
```

```
// src/components/__tests__/__snapshots__/TableHead-R.test.js.snap

exports[`TableHead renders button, row count, and fields 1`] = `
<thead>
  <tr>
    <th
      onClick={[Function]}
      scope="col"
      title="add row"
    >
      <button>
        +
      </button>
    </th>
  </tr>
```

```
<tr>
  <th
    scope="col"
  >
    7
  </th>
  <th
    scope="col"
  >
    when
  </th>
  <th
    scope="col"
  >
```

```
      what
    </th>
    <th
      scope="col"
    >
      whimper
    </th>
</thead>
`;
```

Read or render updating tests, part 1

An expected relevant change affects the test.

To sort rows by a field, click a column heading.

Add to the th element for each field:

- onClick property
- span and abbr children

```
npm test -- TableHead-R

TypeError: Cannot read property 'sorting' of undefined
```

```
// Import initial state of new child reducer to sort records.
import {viewInitial} from '../../reducers/view';

// Add new properties of TableHead component.

    expect(renderer.create(
      <TableHead
        addRow={() => {}}
        count={7}
        fields={fields}
        sortRecords={() => {}}
        view={viewInitial}
      />
    ).toJSON()).toMatchSnapshot();
```

```
TableHead › renders button, count, and fields

expect(value).toMatchSnapshot()

Received value does not match stored snapshot 1.

- Snapshot
+ Received
```

```
  <tr>
    <th
+     onClick={[Function]}
      scope="col"
    >
      7
    </th>
```

```
     <th
+      onClick={[Function]}
       scope="col"
     >
-      when
+      <span>
+        when
+      </span>
+      <abbr
+        title=""
+      >
+
+      </abbr>
     </th>
```

```
     <th
+      onClick={[Function]}
       scope="col"
     >
+      <span>
         when
+      </span>
+      <abbr
+        title=""
+      >
+
+      </abbr>
     </th>
```

```
    <th
+     onClick={[Function]}
      scope="col"
    >
-     what
+     <span>
+       when
+     </span>
+     <abbr
+       title=""
+     >
+
+     </abbr>
    </th>
```

```
    <th
+     onClick={[Function]}
      scope="col"
    >
+     <span>
        what
+     </span>
+     <abbr
+       title=""
+     >
+
+     </abbr>
    </th>
```

```
     <th
+      onClick={[Function]}
       scope="col"
     >
-      whimper
+      <span>
+        when
+      </span>
+      <abbr
+        title=""
+      >
+
+      </abbr>
     </th>
```

```
     <th
+      onClick={[Function]}
       scope="col"
     >
+      <span>
         whimper
+      </span>
+      <abbr
+        title=""
+      >
+
+      </abbr>
     </th>
```

```
Watch Usage
 › Press a to run all tests.
 › Press o to only run tests related to changed files.
 › Press u to update failing snapshots.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.

u
```

Painless snapshot testing

Control changes in components:

- Prevent unexpected regression.

  If change is incorrect, then fix code.

- Confirm expected progress.

  If change is correct, then update snapshot.

Painful snapshot testing 😢

If you let the effort get out of balance:

- Too easy to write a test, which you do once.
- Too hard to understand if it fails, ever after.

  Which changes are correct or incorrect?

  Overlook a change that should be, but isn’t?

The danger of Jest snapshot testing is…

so much diff for each code change

that you wouldn’t see the actual bug

We will need to evolve patterns over time

and figure out the best balance.

https://twitter.com/cpojer/status/774427994077048832

A snapshot test does not tell you your code broke, only that it changed.

It is easier to explain exactly which pieces you care about with the imperative approach…

https://medium.com/@suchipi/thanks-for-your-response-e8e9217db08f

…but I would love to see tooling change that opinion. 😍

https://medium.com/@suchipi/thanks-for-your-response-e8e9217db08f

Purposeful testing

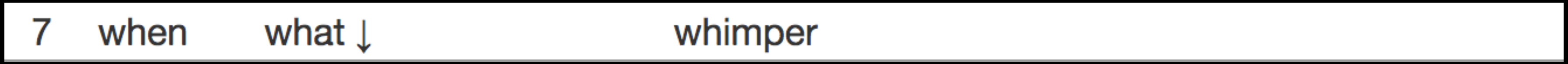
When you write a test, minimize

- irrelevant details, which cause
- unnecessary updates, which risk
- incorrect decisions, especially
- false negatives, failing to report an error

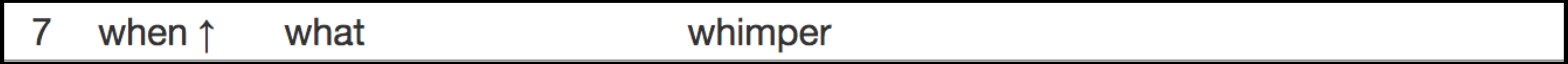
The rest of examples replace toMatchSnapshot with toMatchObject to match a relevant subset of props and descendants in descriptive JSX.

Read or render example: sort indicator

TableHead renders ascending or descending indicator only in heading of primary sort field.



7	when	what ↓	whimper



7	when ↑	what	whimper

Read or render baseline tests

```
// src/components/__test0__/TableHead-R.test.js

import React from 'react';
import {shallow} from 'enzyme';

import TableHead, {ascending, descending} from '../TableHead';

const $abbrAt = (i) => $it.find('tr').at(1).find('abbr').at(i);
```

```
it('renders ascending indicator in `what` heading', () => {
  const $it = shallow(
    <TableHead
      addRow={() => {}}
      count={7}
      fields={fields}
      sortRecords={() => {}}
      view={Object.assign({}, viewInitial, {sorting: [
        {fieldKey: 'what', descending: false},
      ]})}
    />
  );
  let $abbr;

  $abbr = $abbrAt(0);
  expect($abbr.prop('title')).toBe('');
  expect($abbr.text()).toBe('');

  $abbr = $abbrAt(1);
  expect($abbr.prop('title')).toBe('ascending');
  expect($abbr.text()).toBe(ascending);

  $abbr = $abbrAt(2);
  expect($abbr.prop('title')).toBe('');
  expect($abbr.text()).toBe('');
});
```

```
it('renders descending indicator in `when` heading', () => {
  const $it = shallow(
    <TableHead
      addRow={() => {}}
      count={7}
      fields={fields}
      sortRecords={() => {}}
      view={Object.assign({}, viewInitial, {sorting: [
        {fieldKey: 'when', descending: true},
        {fieldKey: 'what', descending: false},
      ]})}
    />
  );
  let $abbr;

  $abbr = $abbrAt(0);
  expect($abbr.prop('title')).toBe('descending');
  expect($abbr.text()).toBe(descending);

  $abbr = $abbrAt(1);
  expect($abbr.prop('title')).toBe('');
  expect($abbr.text()).toBe('');

  $abbr = $abbrAt(2);
  expect($abbr.prop('title')).toBe('');
  expect($abbr.text()).toBe('');
});
```

Read or render proposed tests 😊

- Content of non-heading cell is irrelevant.
- Content of span is irrelevant.
- Omit props from expected elements.

  Test onClick prop elsewhere as interaction.

Read or render proposed tests 😊

```
// src/components/__tests__/TableHead-R.test.js

import React from 'react';
import {
  irrelevant,
  relevantTestObject,
  renderAsTestObject,
} from 'react-test-renderer';                              // proposed

import TableHead, {ascending, descending} from '../TableHead';
```

```
it('renders ascending indicator in `what` heading', () => {
  expect(renderAsTestObject(
    <TableHead
      addRow={() => {}}
      count={7}
      fields={fields}
      sortRecords={() => {}}
      view={Object.assign({}, viewInitial, {sorting: [
        {fieldKey: 'what', descending: false},
      ]})}
    />
  ).children[1]).toMatchObject(relevantTestObject(
    <tr>
      <th>{irrelevant}</th>
      <th>
        <span>{irrelevant}</span>
        <abbr title=""></abbr>
      </th>
      <th>
        <span>{irrelevant}</span>
        <abbr title="ascending">{ascending}</abbr>
      </th>
      <th>
        <span>{irrelevant}</span>
        <abbr title=""></abbr>
      </th>
    </tr>
  ));
});
```

```
it('renders descending indicator in `when` heading', () => {
  expect(renderAsTestObject(
    <TableHead
      addRow={() => {}}
      count={7}
      fields={fields}
      sortRecords={() => {}}
      view={Object.assign({}, viewInitial, {sorting: [
        {fieldKey: 'when', descending: true},
        {fieldKey: 'what', descending: false},
      ]})}
    />
  ).children[1]).toMatchObject(relevantTestObject(
    <tr>
      <th>{irrelevant}</th>
      <th>
        <span>{irrelevant}</span>
        <abbr title="descending">{descending}</abbr>
      </th>
      <th>
        <span>{irrelevant}</span>
        <abbr title=""></abbr>
      </th>
      <th>
        <span>{irrelevant}</span>
        <abbr title=""></abbr>
      </th>
    </tr>
  ));
});
```

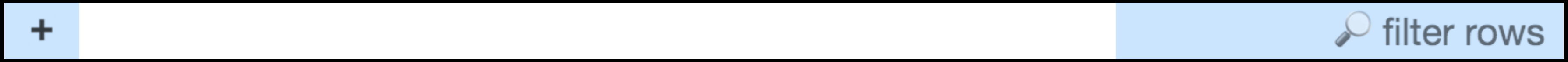
How do you get the relevant JSX?	When
• Type it, based on render method.	TDD
• Copy from existing Read snapshot, and delete whatever is irrelevant.	TDD
• Copy from temporary snapshot…	non-TDD
• Maybe someday, paste by editor integration…	

Read or render updating tests, part 2

An expected relevant change affects the test.

To filter rows, type a substring.

Add input element at right of first tr.



Read or render updating tests, part 2 🚦

- Because one proposed snapshot fails, you must decide to update it. 😊
- Because the baseline assertion passes, you must remember to update it. 😐

```
TableHead › renders button, count, and fields

expect(value).toMatchSnapshot()

Received value does not match stored snapshot 1.

- Snapshot
+ Received
```

```diff
      <th
         colSpan={3}
         scope="colgroup"
-     />
+     >
+       <input
+         onChange={[Function]}
+         placeholder="🔎 filter rows"
+       />
+     </th>
    </tr>
```

```
Watch Usage
 › Press a to run all tests.
 › Press o to only run tests related to changed files.
 › Press u to update failing snapshots.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.

u
```

Interact

If you don’t have time to write tests
for the rest of the patterns,
or if components render simple views of data,
then you might just test rendering and
that interface events cause correct actions.

Interact

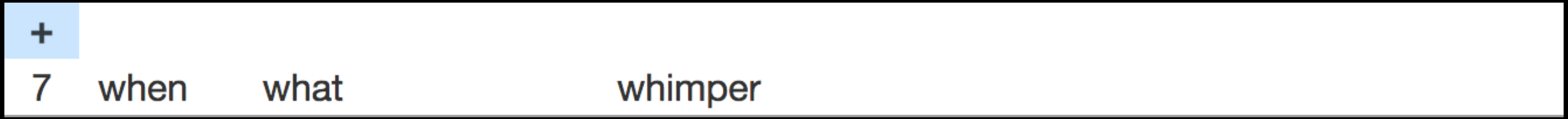
jest.fn() returns a mock function,

also known as a spy,

to assert behavior of calling code,

not just output.

Interact example: click cells in table head



+			
7	when	what	whimper

Interact tests

```
// src/components/__tests__/TableHead-I.test.js

import React from 'react';
import {mount} from 'enzyme';

import TableHead from '../TableHead';
```

```
describe('TableHead', () => {
  const addRow = jest.fn();
  const sortRecords = jest.fn();
  const $it = mount(
    <TableHead
      addRow={addRow}
      count={7}
      fields={fields}
      sortRecords={sortRecords}
      view={viewInitial}
    />
  );
```

```
// Click every cell in table head
$it.find('thead tr').forEach($tr => {
  $tr.find('th').forEach($th => {
    $th.simulate('click');
  });
});
```

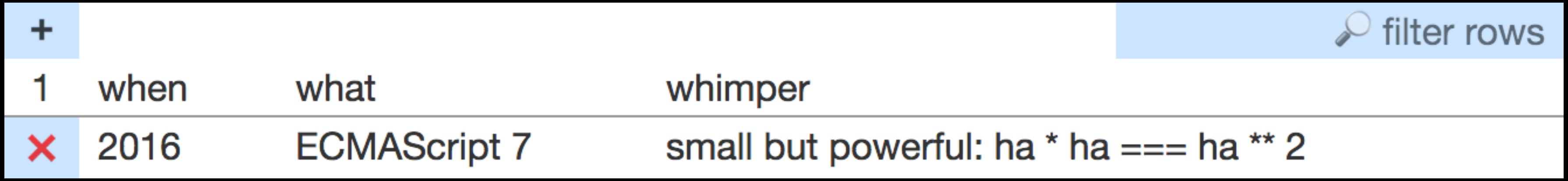
```
  // Interface events cause correct actions.

  it('adds a row', () => {
    expect(addRow).toHaveBeenCalledTimes(1);
    expect(addRow).toHaveBeenCalledWith();
  });

  it('sorts rows', () => {
    // [] from click non-field heading at left to reset sort order.
    expect(sortRecords.mock.calls).toEqual([[]].concat(
      fields.map(({key}) => [key])
    ));
  })
});
```

Create

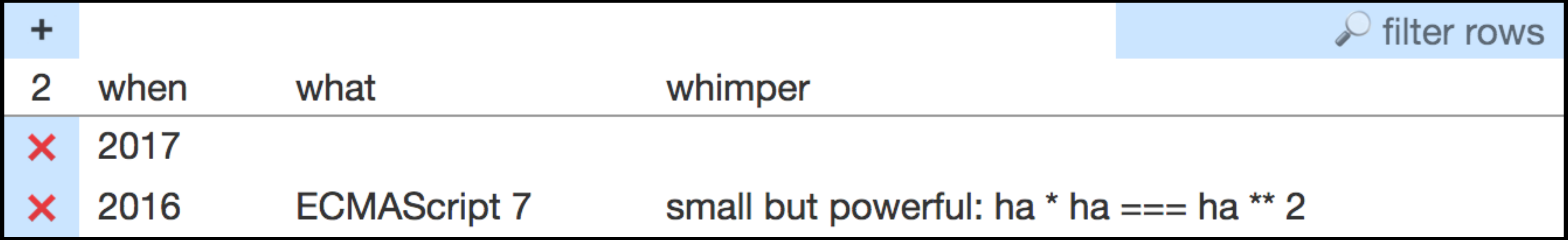
An action adds a child to a component.

- where: add to correct place in siblings
- what: delegate details about children
- what else: update the (derived) state?

Create example: add row



+			filter rows
1	when	what	whimper
✕	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2



+			filter rows
2	when	what	whimper
✕	2017		
✕	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2

Create baseline tests

```
// src/components/__test0__/Table-C.test.js

import React from 'react';
import {mount} from 'enzyme';

import Table from '../Table';
```

```
// helpers for operation on component

const clickAdd = ($it) => {
  $it.find('thead tr').at(0).find('th').at(0).simulate('click');
};

const countRows = ($it) =>
  Number($it.find('thead tr').at(1).find('th').at(0).text());

const countTableRows = ($it) => $it.find('TableRow').length;

const recordAtTableRow = ($it, i) =>
  $it.find('TableRow').at(i).prop('record');
```

```
describe('Table', () =>
  it('creates a row preceding one existing row', () => {
    const store = createStore(reducer);
    const records = [recordB];
    store.dispatch(receiveData(fields, records));
    const $it = mount(
      <Provider store={store}>
        <Table />
      </Provider>
    );
    clickAdd($it);
    expect(countRows($it)).toBe(records.length + 1);
    expect(countTableRows($it)).toBe(records.length + 1);
    expect(recordAtTableRow($it, 0)).toEqual(recordDefault(fields));
    expect(recordAtTableRow($it, 1)).toEqual(recordB);
  });

  // and so on
});
```

Create proposed tests

```
// src/components/__tests__/Table-C.test.js

import React from 'react';
import {mount} from 'enzyme';
import {mountToShallowObject} from 'enzyme-to-json';       // proposed
import {relevantTestObject} from 'react-test-renderer';   // proposed

import Table from '../Table';
const TableRow = () => {}; // mock, and provide only relevant props
```

```
// helpers for operation on component

const clickAdd = ($it) => {
  $it.find('thead tr').at(0).find('th').at(0).simulate('click');
};

const countRows = ($it) =>
  Number($it.find('thead tr').at(1).find('th').at(0).text());

const tbodyShallow = ($it) =>
  mountToShallowObject($it.find('tbody'));
```

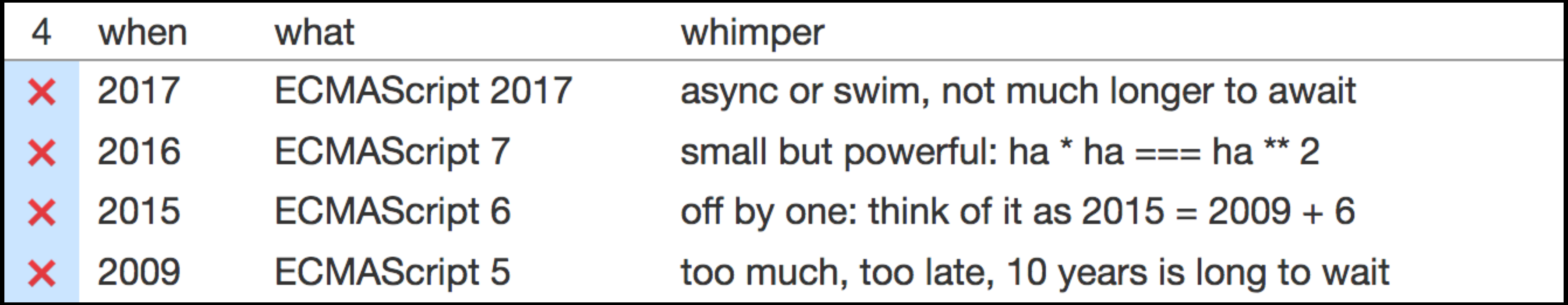
```
describe('Table', () =>
  it('creates a row preceding one existing row', () => {
    const store = createStore(reducer);
    const records = [recordB];
    store.dispatch(receiveData(fields, records));
    const $it = mount(
      <Provider store={store}>
        <Table />
      </Provider>
    );
    clickAdd($it);
    expect(countRows($it)).toBe(records.length + 1);
    expect(tbodyShallow($it)).toMatchObject(relevantTestObject(
      <tbody>
        <TableRow record={recordDefault(fields)} />
        <TableRow record={recordB} />
      </tbody>
    ));
  });

  // and so on
});
```

Delete

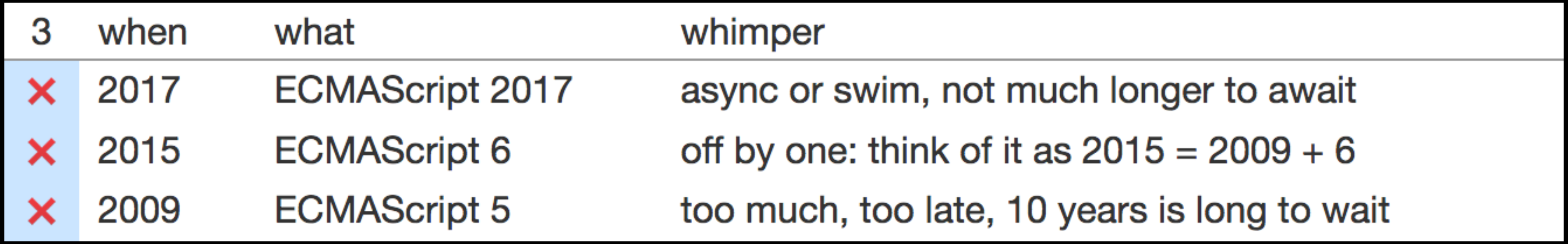
An action **removes a child** from a component.

- **where**: remove from correct place in siblings
- **what**: delegate details about children
- **what else**: update the (derived) state?

Delete example: delete row



4	when	what	whimper
✖	2017	ECMAScript 2017	async or swim, not much longer to await
✖	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✖	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✖	2009	ECMAScript 5	too much, too late, 10 years is long to wait



3	when	what	whimper
✖	2017	ECMAScript 2017	async or swim, not much longer to await
✖	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✖	2009	ECMAScript 5	too much, too late, 10 years is long to wait

Delete baseline tests

```
// src/components/__test0__/Table-D.test.js

import React from 'react';
import {mount} from 'enzyme';

import Table from '../Table';
```

```
// helpers for operation on component

const clickDelete = ($it, i) => {
  $it.find('tbody tr').at(i).find('td').at(0).simulate('click');
};

const countRows = ($it) =>
  Number($it.find('thead tr').at(1).find('th').at(0).text());

const countTableRows = ($it) => $it.find('TableRow').length;

const recordAtTableRow = ($it, i) =>
  $it.find('TableRow').at(i).prop('record');
```

```
describe('Table deletes records', () => {
  const store = createStore(reducer);
  const records = [recordA, recordB, recordC, recordD];
  store.dispatch(receiveData(fields, records));
  const $it = mount(
    <Provider store={store}>
      <Table />
    </Provider>
  );
  test('in the middle', () => {
    clickDelete($it, 1); // recordB
    expect(countRows($it)).toEqual(records.length - 1);
    expect(countTableRows($it)).toEqual(records.length - 1);
    expect(recordAtTableRow($it, 0)).toEqual(recordA);
    expect(recordAtTableRow($it, 1)).toEqual(recordC);
    expect(recordAtTableRow($it, 2)).toEqual(recordD);
  });

  // and so on
});
```

Delete proposed tests

```
// src/components/__tests__/Table-D.test.js

import React from 'react';
import {mount} from 'enzyme';
import {mountToShallowObject} from 'enzyme-to-json';       // proposed
import {relevantTestObject} from 'react-test-renderer';   // proposed

import Table from '../Table';
const TableRow = () => {}; // mock, and provide only relevant props
```

```
// helpers for operation on component

const clickDelete = ($it, i) => {
  $it.find('tbody tr').at(i).find('td').at(0).simulate('click');
};

const countRows = ($it) =>
  Number($it.find('thead tr').at(1).find('th').at(0).text());

const tbodyShallow = ($it) =>
  mountToShallowObject($it.find('tbody'));
```

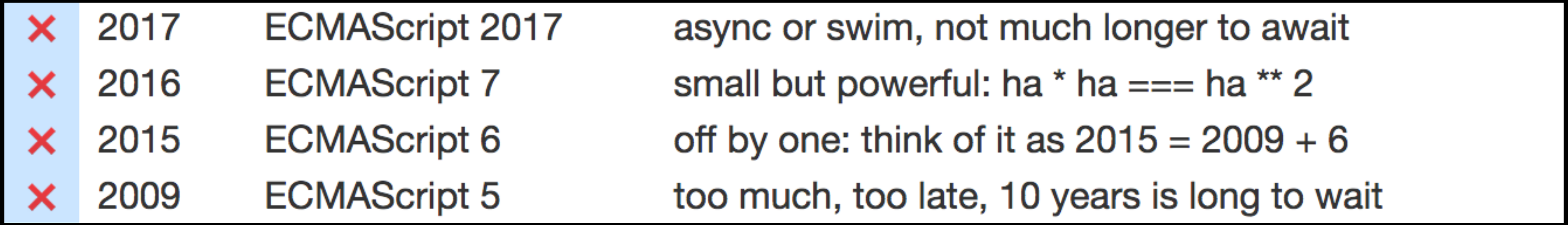
```
describe('Table deletes records', () => {
  const store = createStore(reducer);
  const records = [recordA, recordB, recordC, recordD];
  store.dispatch(receiveData(fields, records));
  const $it = mount(
    <Provider store={store}>
      <Table />
    </Provider>
  );
 test('in the middle', () => {
    clickDelete($it, 1); // recordB
    expect(countRows($it)).toEqual(records.length - 1);
    expect(tbodyShallow($it)).toMatchObject(relevantTestObject(
      <tbody>
        <TableRow record={recordA} />
        <TableRow record={recordC} />
        <TableRow record={recordD} />
      </tbody>
    ));
  });

  // and so on
});
```

View

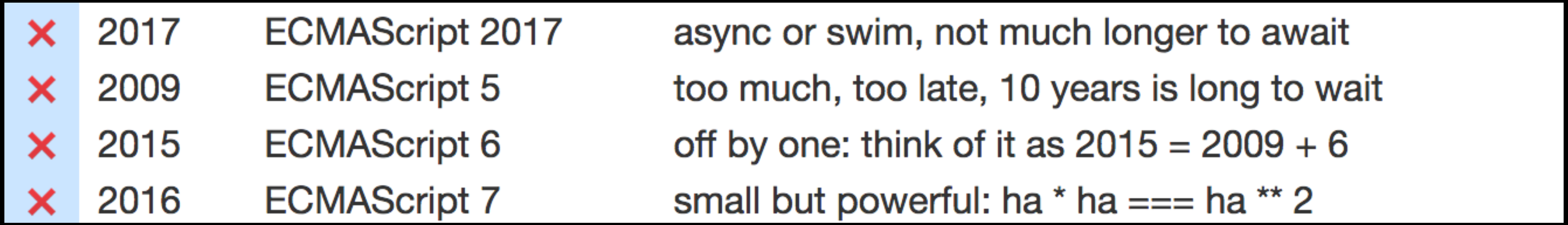
An action changes derived state of a component.

- Create: filter to "add" children
- Delete: filter to "remove" children
- Create and Delete: sort to reorder children
- Update: indicate state in user interface

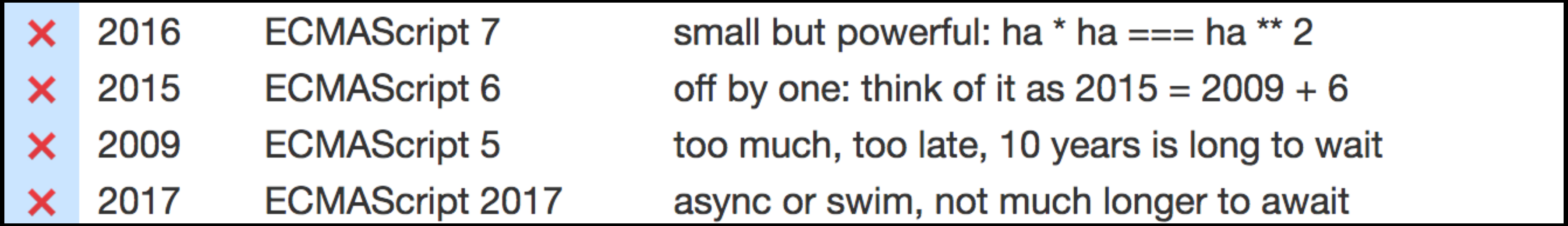
View example: sort rows



✕	2017	ECMAScript 2017	async or swim, not much longer to await
✕	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✕	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait



✕	2017	ECMAScript 2017	async or swim, not much longer to await
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait
✕	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✕	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2



✕	2016	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✕	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait
✕	2017	ECMAScript 2017	async or swim, not much longer to await

View baseline tests

```
// src/components/__test0__/Table-V.test.js

import React from 'react';
import {mount} from 'enzyme';

import Table from '../Table';
```

```
// helpers for operation on component

const clickHeading = ($it, i) => {
  $it.find('thead tr').at(1).find('th').at(1 + i).simulate('click');
};

const recordAtTableRow = ($it, i) =>
  $it.find('TableRow').at(i).prop('record');
```

```
describe('Table sorting', () => {
  const store = createStore(reducer);
  const records = [recordA, recordB, recordC, recordD];
  store.dispatch(receiveData(fields, records));
  const $it = mount(
    <Provider store={store}>
      <Table />
    </Provider>
  );
```

```
it('is ascending on click `what` heading', () => {
  clickHeading($it, 1);
  expect(recordAtTableRow($it, 0)).toEqual(recordA);
  expect(recordAtTableRow($it, 1)).toEqual(recordD);
  expect(recordAtTableRow($it, 2)).toEqual(recordC);
  expect(recordAtTableRow($it, 3)).toEqual(recordB);
});
```

```
it('is descending on click `what` heading again', () => {
  clickHeading($it, 1);
  expect(recordAtTableRow($it, 0)).toEqual(recordB);
  expect(recordAtTableRow($it, 1)).toEqual(recordC);
  expect(recordAtTableRow($it, 2)).toEqual(recordD);
  expect(recordAtTableRow($it, 3)).toEqual(recordA);
});

// and so on
```

```
  it('resets on click non-field heading at left', () => {
    clickHeading($it, -1);
    expect(recordAtTableRow($it, 0)).toEqual(recordA);
    expect(recordAtTableRow($it, 1)).toEqual(recordB);
    expect(recordAtTableRow($it, 2)).toEqual(recordC);
    expect(recordAtTableRow($it, 3)).toEqual(recordD);
  });
});
```

View proposed tests 😊

```
// src/components/__tests__/Table-V.test.js

import React from 'react';
import {mount} from 'enzyme';
import {mountToShallowObject} from 'enzyme-to-json';      // proposed
import {relevantTestObject} from 'react-test-renderer';   // proposed

import Table from '../Table';
const TableRow = () => {}; // mock, and provide only relevant props
```

```
// helpers for operation on component

const clickHeading = ($it, i) => {
  $it.find('thead tr').at(1).find('th').at(1 + i).simulate('click');
};

const tbodyShallow = ($it) =>
  mountToShallowObject($it.find('tbody'));
```

```
describe('Table sorting', () => {
  const store = createStore(reducer);
  const records = [recordA, recordB, recordC, recordD];
  store.dispatch(receiveData(fields, records));
  const $it = mount(
    <Provider store={store}>
      <Table />
    </Provider>
  );
```

```
it('is ascending on click `what` heading', () => {
  clickHeading($it, 1);
  expect(tbodyShallow($it)).toMatchObject(relevantTestObject(
    <tbody>
      <TableRow record={recordA} />
      <TableRow record={recordD} />
      <TableRow record={recordC} />
      <TableRow record={recordB} />
    </tbody>
  ));
});
```

```
it('is descending on click `what` heading again', () => {
   clickHeading($it, 1);
   expect(tbodyShallow($it)).toMatchObject(relevantTestObject(
     <tbody>
       <TableRow record={recordB} />
       <TableRow record={recordC} />
       <TableRow record={recordD} />
       <TableRow record={recordA} />
     </tbody>
   ));
 });

 // and so on
```

```
  it('resets on click non-field heading at left', () => {
    clickHeading($it, -1);
    expect(tbodyShallow($it)).toMatchObject(relevantTestObject(
      <tbody>
        <TableRow record={recordA} />
        <TableRow record={recordB} />
        <TableRow record={recordC} />
        <TableRow record={recordD} />
      </tbody>
    ));
  });
});
```

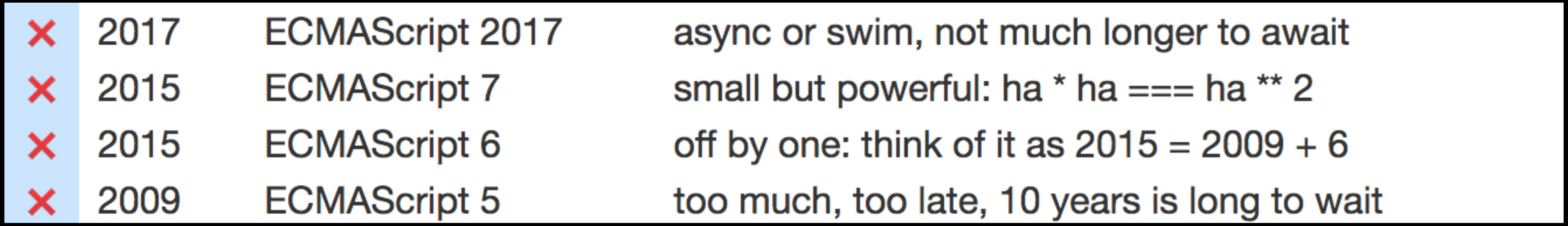
Update or undo

An action changes the state of a component.

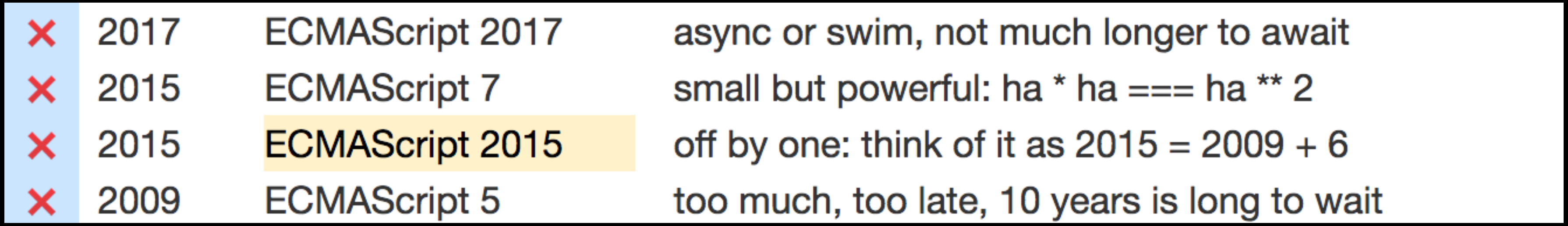
Assert relevant attributes, content, or structure:

- prev state: before the action
- next state: after the action
- prev state: undo the action

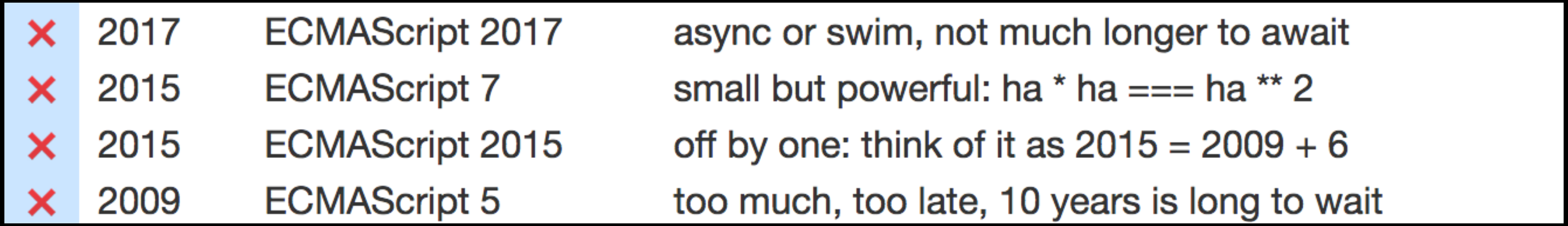
Update or undo example: input or edit text



✕	2017	ECMAScript 2017	async or swim, not much longer to await
✕	2015	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✕	2015	ECMAScript 6	off by one: think of it as 2015 = 2009 + 6
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait



✕	2017	ECMAScript 2017	async or swim, not much longer to await
✕	2015	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✕	2015	ECMAScript 2015	off by one: think of it as 2015 = 2009 + 6
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait



✕	2017	ECMAScript 2017	async or swim, not much longer to await
✕	2015	ECMAScript 7	small but powerful: ha * ha === ha ** 2
✕	2015	ECMAScript 2015	off by one: think of it as 2015 = 2009 + 6
✕	2009	ECMAScript 5	too much, too late, 10 years is long to wait

Update or undo baseline tests

```
// src/components/__test0__/Table-U.test.js

import React from 'react';
import {mount} from 'enzyme';

import invariant from 'invariant';

import Table from '../Table';
```

```
// helpers for operation on component

const $tdAtIndex = ($it, rowIndex, fieldIndex) =>
  $it.find('tbody tr').at(rowIndex).find('td').at(1 + fieldIndex);
```

```
describe('Table', () => {
  it('updates a text field', () => {
    const store = createStore(reducer);
    store.dispatch(receiveData(fields, records));
    const $it = mount(
      <Provider store={store}>
        <Table />
      </Provider>
    );

    const rowIndex = 2;
    const fieldIndex = 1;
    const field = fields[fieldIndex];
    invariant(field.type === 'text', 'testing a text field');
```

```
const $td = $tdAtIndex($it, rowIndex, fieldIndex);     // wrapper
const td = $td.get(0);                                 // element
if (!td.dataset) {
  // Make up for limitation of jsdom
  td.dataset = {
    rowIndex: td.getAttribute('data-record-id'),
    fieldKey: td.getAttribute('data-field-key'),
  };
}
$td.simulate('doubleClick');
```

```
const textInitial = records[rowIndex][fields[fieldIndex].key];
expect($td.find('span').at(0).text()).toBe(textInitial);
expect($td.find('input').at(0).props()).toMatchObject({
  defaultValue: textInitial,
  type: 'text',
});
```

```
    const textUpdated = 'ECMAScript 2015';
    $td.find('input').get(0).value = textUpdated;

    $td.find('form').simulate('submit');
    expect($td.text()).toBe(textUpdated);
  });
});
```

Update or undo proposed tests 😊

```
// src/components/__tests__/Table-U.test.js

import React from 'react';
import {mount} from 'enzyme';
import {mountToDeepObject} from 'enzyme-to-json';           // proposed
import {relevantTestObject} from 'react-test-renderer';    // proposed

import invariant from 'invariant';

import Table from '../Table';
```

```
// helpers for operation on component

const $tdAtIndex = ($it, rowIndex, fieldIndex) =>
  $it.find('tbody tr').at(rowIndex).find('td').at(1 + fieldIndex);
```

```
describe('Table', () => {
  it('updates a text field', () => {
    const store = createStore(reducer);
    store.dispatch(receiveData(fields, records));
    const $it = mount(
      <Provider store={store}>
        <Table />
      </Provider>
    );

    const rowIndex = 2;
    const fieldIndex = 1;
    const field = fields[fieldIndex];
    invariant(field.type === 'text', 'testing a text field');
```

```
const $td = $tdAtIndex($it, rowIndex, fieldIndex);     // wrapper
const td = $td.get(0);                                 // element
if (!td.dataset) {
  // Make up for limitation of jsdom
  td.dataset = {
    rowIndex: td.getAttribute('data-record-id'),
    fieldKey: td.getAttribute('data-field-key'),
  };
}
$td.simulate('doubleClick');
```

```
const textInitial = records[rowIndex][fields[fieldIndex].key];
expect(mountToDeepObject($td)).toMatchObject(relevantTestObject(
  <td>
    <div>
      <span>{textInitial}</span>
      <form>
        <input
          defaultValue={textInitial}
          type="text"
        />
      </form>
    </div>
  </td>
));
```

Copy from temporary snapshot.
Delete whatever is irrelevant.

```
<td
  data-field-key="what"
  data-record-id={2}
>
  <div>
    <span>
      ECMAScript 6
    </span>
    <form
      onSubmit={([Function])}
    >
      <input
        autoFocus={true}
        defaultValue="ECMAScript 6"
        type="text"
      />
    </form>
  </div>
</td>
```

```
    const textUpdated = 'ECMAScript 2015';
    $td.find('input').get(0).value = textUpdated;

    $td.find('form').simulate('submit');
    expect(mountToDeepObject($td)).toMatchObject(relevantTestObject(
      <td>{textUpdated}</td>
    ));
  });
});
```

```
describe('Table', () => {
  it('cancels update on double-click input', () => {
    // initialize as in preceding examples

    const prev = mountToDeepObject($td);
    $td.simulate('doubleClick');
    $td.find('input').simulate('doubleClick');
    const next = mountToDeepObject($td);

    expect(next).toEqual(prev);
  });
});
```

It seems that perfection is attained

not when there is nothing more to add,

but when there is nothing more to remove.

Antoine de Saint Exupéry

Add as many abstract assertions as you can? 😐

Delete as many irrelevant details as you can! 😊

Bonus to swim in deep water

Update or undo binding callbacks 🚦

```
// src/components/__tests__/TableHead-U.test.js

describe('TableHead sorting indicator', () => {
  // initialize $it
  const trInitial = trShallow($it);
  // test various updates, and then add one more assertion:
  it('resets on click cell at left', () => {
    clickHeading($it, -1);
    expect(trShallow($it)).toEqual(trInitial);
  });
});
```

```
TableHead sorting indicator › resets on click cell at left

expect(received).toEqual(expected)

Expected value to equal:
{"$$typeof": Symbol(react.test.json), "children": [{"$$typeof":
Symbol(react.test.json), "children": [4], "props": { …

Received:
{"$$typeof": Symbol(react.test.json), "children": [{"$$typeof":
Symbol(react.test.json), "children": [4], "props": { …

Difference:
Compared values have no visual difference.
```

```
// If render method has arrow function as value of onClick prop,
// it creates a new function each time React re-renders TableHead.
// So toEqual assertion fails. With an unhelpful error in Jest 19.

fields.map(field => (
  <th
    key={field.key}
    onClick={() => sortRecords(field.key)}
    scope="col"
  >
    …
  </th>
)}
```

```
// An alternative is bind callback functions once in constructor:

class TableHead extends Component {
  props: Props;
  _sortRecords: Array<Function>;

  constructor(props: Props) {
    super(props);
    const {fields, sortRecords} = props;
    this._sortRecords = fields.map(field => () => {
      sortRecords(field.key);
    });
  }
```

```
// If render method always provides same value of onClick prop,
// then the toEqual assertion suceeds.

fields.map((field, i) => (
  <th
    key={field.key}
    onClick={this._sortRecords[i]}
    scope="col"
  >
    …
  </th>
)}
```

React element is a plain object that describes a component instance or DOM node: type and properties, including children.

https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html

```
// JSX compiles to React.createElement

<div>
  <span>ECMAScript 6</span>
  <form onSubmit={onSubmit}>
    <input
      autoFocus={true}
      defaultValue="ECMAScript 6"
      type="text"
    />
  </form>
</div>

React.createElement(type, props, ...children)
```

```
// props and no children

<input
  autoFocus={true}
  defaultValue="ECMAScript 6"
  type="text"
/>

React.createElement('input', {
  autoFocus: true,
  defaultValue: 'ECMAScript 6',
  type: 'text',
})
```

```
// no props and a text child

<span>ECMAScript 6</span>

React.createElement('span', null, 'ECMAScript 6')
```

```
// a prop and an element child

<form onSubmit={onSubmit}>
  <input
    autoFocus={true}
    defaultValue="ECMAScript 6"
    type="text"
  />
</form>

React.createElement('form', {onSubmit}, input)
// input = React.createElement('input', { … })
// no props and 2 children

<div>
  <span>ECMAScript 6</span>
  <form onSubmit={onSubmit}><input … /></form>
</div>

React.createElement('div', null, span, form)
// span = React.createElement('span', null, 'ECMAScript 6')
// form = React.createElement('form', {onSubmit}, input)
// no props nor children

<br />

React.createElement('br')
```

```
// React element and test object

const element = React.createElement(type, props, ...children)

const object =  renderer.create(element).toJSON()      // does render

                renderAsTestObject(element)            // does render
                relevantTestObject(element)         // doesn’t render

                shallowToJson(shallow(element))      // only children
                mountToJson(mount(element))    // includes components

                mountToShallowObject($it.find(selector).at(index))
                mountToDeepObject($it.find(selector).at(index))
```

```
// React element

{
  $$typeof: Symbol(…),
  type,
  props: {
     …,
     children: …,
  },
  key: null,
  ref: null,
  _owner: null,
  _store: {}
}
```

```
// test object

{
  $$typeof: Symbol.for(…),
  type,
  props: {
    …,
  },
  children: …,
}
```

```
// props and no children

{
  $$typeof: Symbol(…),
  type: 'input',
  props: {
    autoFocus: true,
    defaultValue: 'ECMAScript 6',
    type: 'text',
  },
}
```

```
{
  $$typeof: Symbol.for(…),
  type: 'input',
  props: {
    autoFocus: true,
    defaultValue: 'ECMAScript 6',
    type: 'text',
  },
  children: null,
}
```

```
// no props and a text child

{
  $$typeof: Symbol(…),
  type: 'span',
  props: {
    children: 'ECMAScript 6',
  },
}
```

```
{
  $$typeof: Symbol.for(…),
  type: 'input',
  props: {},
  children: ['ECMAScript 6'],
}
```

```
// a prop and an element child
```

```
{
  $$typeof: Symbol(…),
  type: 'form',
  props: {
    onSubmit,
    children: input,
  },
}
```

```
{
  $$typeof: Symbol.for(…),
  type: 'form',
  props: {
    onSubmit,
  },
  children: [input],
}
```

```
// no props and 2 children

{
  $$typeof: Symbol(…),
  type: 'div',
  props: {
   children: [span, form],
  },
}
```

```
{
  $$typeof: Symbol.for(…),
  type: 'div',
  props: {},
  children: [span, form],
}
```

```
// no props nor children

{
  $$typeof: Symbol(…),
  type: 'br',
  props: {},
}
```

```
{
  $$typeof: Symbol.for(…),
  type: 'br',
  props: {},
  children: null,
}
```