Append-only development with React

An intro to Behavioral Programming

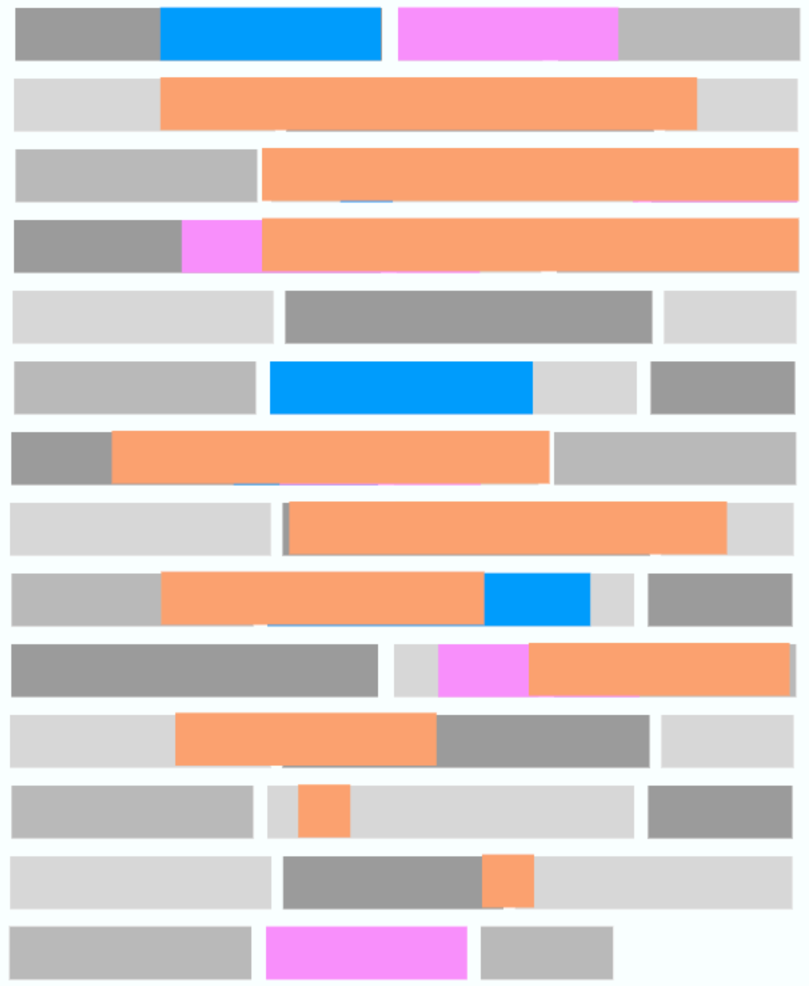
Luca Matteis
@lmatteis

Ask HN: Why Isn't Functional Programming Taking Over?

Because programming languages are practically irrelevant in making products. They are only important to developers. In the last thirty years of programming I have yet to see a project fail because of the programming language or code quality. More often than not it is because someone made something that no one wanted. From a reliability stand point Windows has always been a nightmare. It is still one of the best and most successful products of all time. It could be written in Urdu and no one would care.

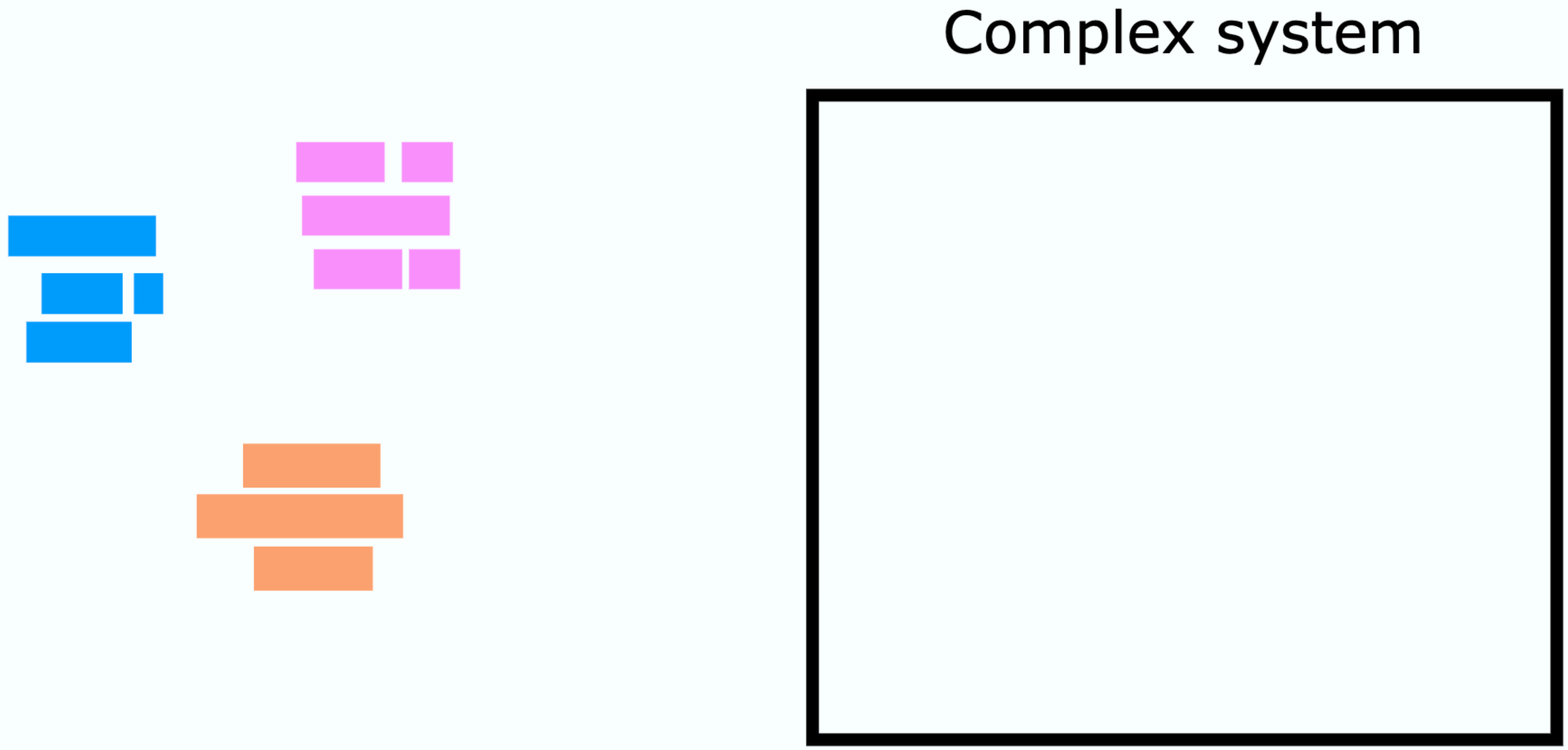
The problem

No matter the tools or abstractions we use, having to maintain a complex artifact we call “code” seems to be the root of the problem.



General intuition

What if we could make changes or understand how complex systems work without having to read and maintain an artifact?


Complex system

Behavioral Programming

David Harel
Weizmann Institute of Science

Assaf Marron
Weizmann Institute of Science

Gera Weiss
Ben Gurion University of the Negev

ABSTRACT

We describe an implementation-independent programming paradigm, *behavioral programming*, which allows programmers to build executable reactive systems from specifications of behavior that are aligned with the requirements. Behavioral programming simplifies the task of dealing with under-specification and conflicting requirements by enabling the addition of software modules that can not only add to but also modify existing behaviors. A behavioral program employs specialized programming idioms for expressing what must, may, or must not happen, and a novel method for the collective execution of the resulting scenarios. Behavioral programming grew out of the scenario-based language of *live sequence charts* (LSC), and is now implemented also in Java and in other environments. We illustrate the approach with detailed examples in Java and LSC, and also review recent work, including a visual trace-comprehension tool, model-checking assisted development, and extending behavioral programs to be adaptive.

1. INTRODUCTION

Spelling out the requirements for a software system under development is not an easy task, and translating captured requirements into correct operational software can be even harder. Many technologies (languages, modeling tools, programming paradigms) and methodologies (agile, test-driven, model-driven) were designed, among other things, to help address these challenges. One widely accepted practice is to formalize requirements in the form of use cases and scenarios. Our work extends this approach into using scenarios for actual programming. Specifically, we propose scenario coding techniques and design approaches for constructing reactive systems [25] incrementally from their expected behaviors.

Now we may already start playing. Later, the child may infer, or the teacher may suggest, some tactics:

`AddThirdO:` After placing two O marks in a line, the O player should try to mark the third square (to win the game);

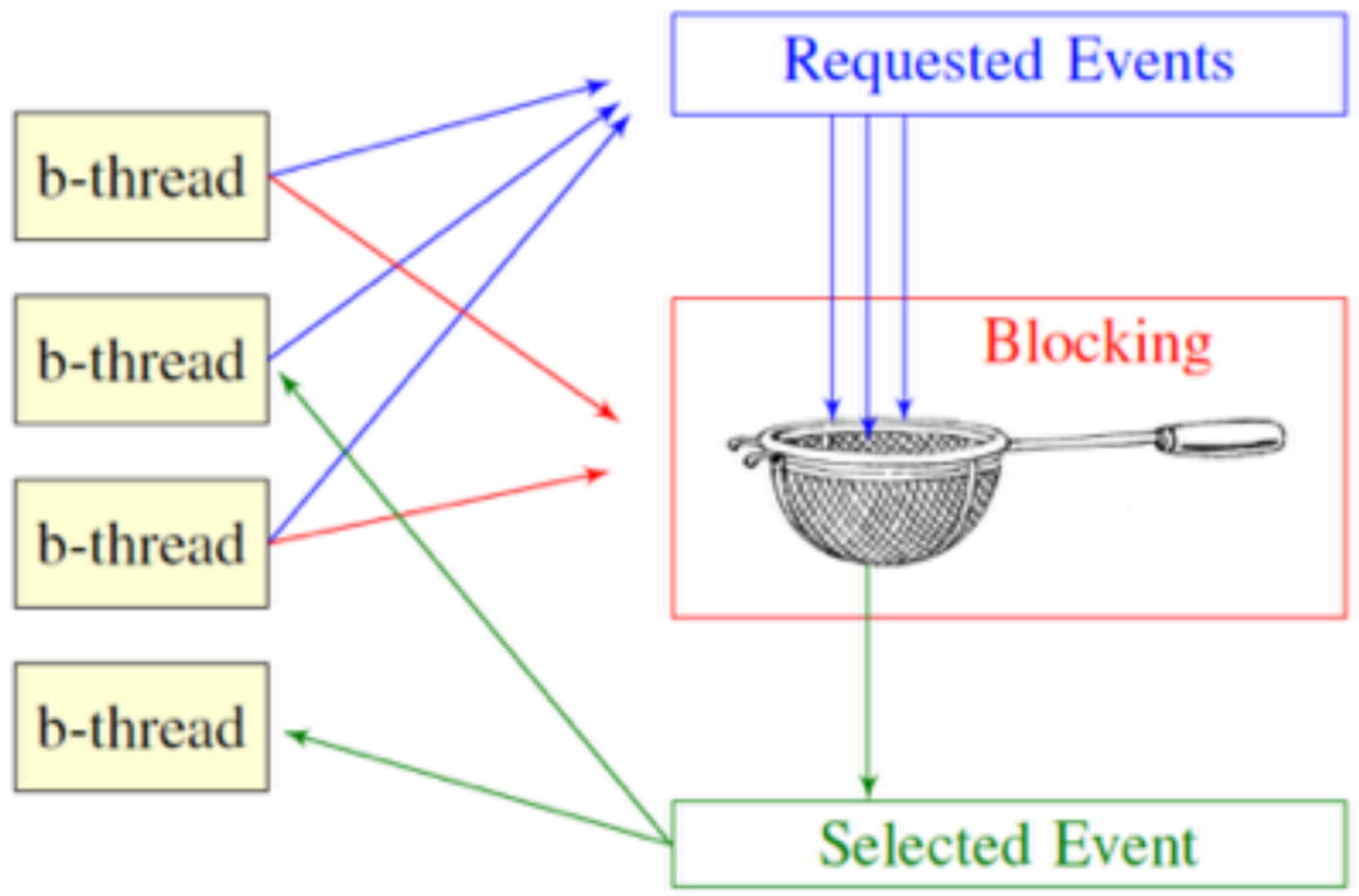
`PreventThirdX:` After the X player marks two squares in a line, the O player should try to mark the third square (to foil the attack);

`DefaultOMoves:` When other tactics are not applicable, player O should prefer the center square, then the corners, and mark an edge square only when there is no other choice;

Such required behaviors can be coded in executable software modules using behavioral programming idioms and infrastructure, as detailed in sections 2 and 3. Full behavioral implementations of the game, in Java and Erlang, are described in [22] and [48], respectively. In [18] we show how model-checking technologies allow discovery of unhandled scenarios, enabling the user to incrementally develop behaviors for new tactics (and forgotten rules) until a software system is achieved that plays legally and assures that the computer never loses.

This example already suggests the following advantages of behavioral programming. First, we were able to code the application incrementally in modules that are aligned with the requirements (game-rules and tactics), as perceived by users and programmers. Second, we added new tactics and rules (and still more can be added) without changing, or even looking at, existing code. Third, the resulting product is modular, in that tactics and rules can be flexibly added and removed to create versions with different functionalities, e.g., to play at different expertise levels.

Naturally, composing behaviors that were programmed without direct consideration of mutual dependencies raises questions about conflicts, under-specification, and synchronization. We deal with these issues by using composition operators that allow both

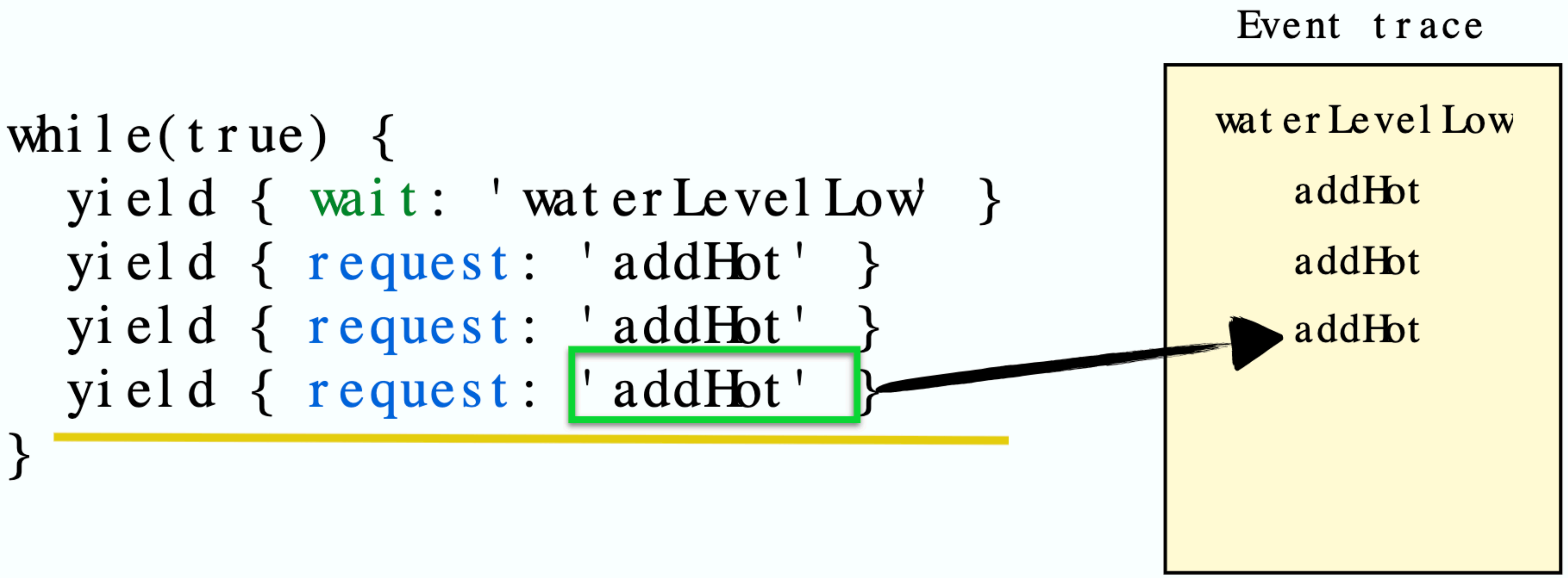


1. All b-threads synchronize and place their "bids":
   - Requesting an event: proposing that the event be considered for triggering, and asking to be notified when it is triggered;
   - Waiting for an event: without proposing its triggering, asking to be notified when the event is triggered;
   - Blocking an event: forbidding the triggering of the event, vetoing requests of other b-threads.
2. An event that is requested and not blocked is selected;
3. b-threads that requested or wait for the selected event are notified;
4. The notified b-threads progress to their next states, where they place new bids.

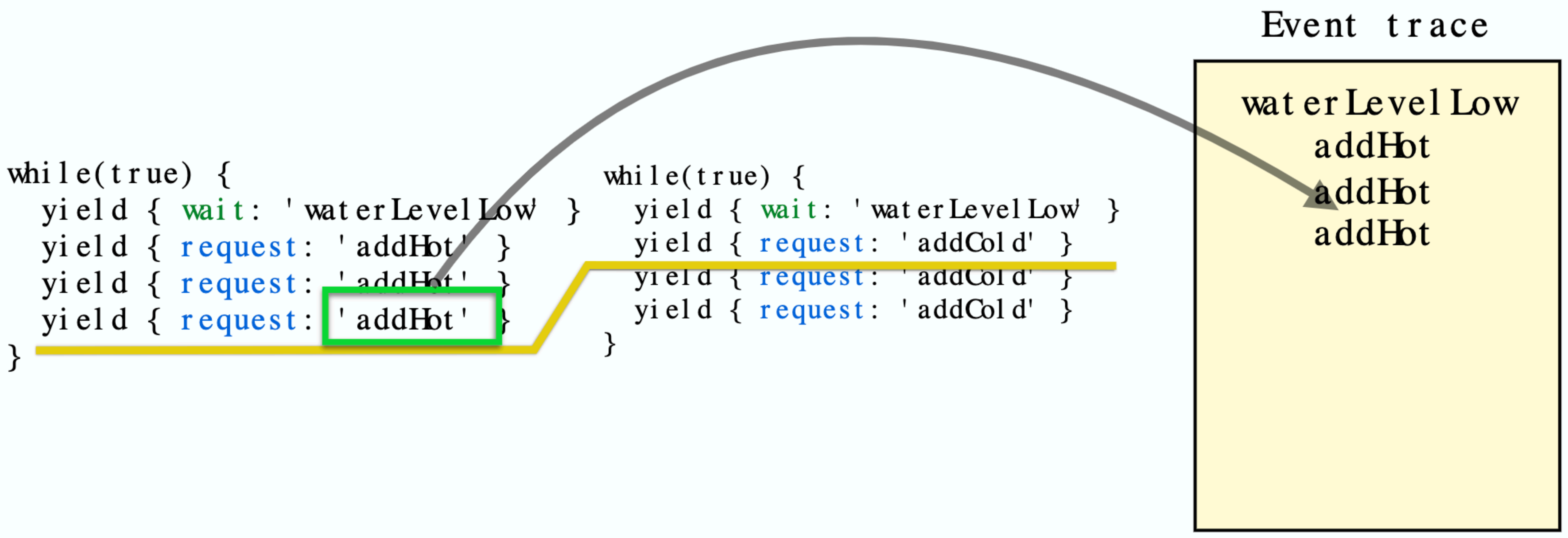
Request, wait and block

```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
}
```

Event trace

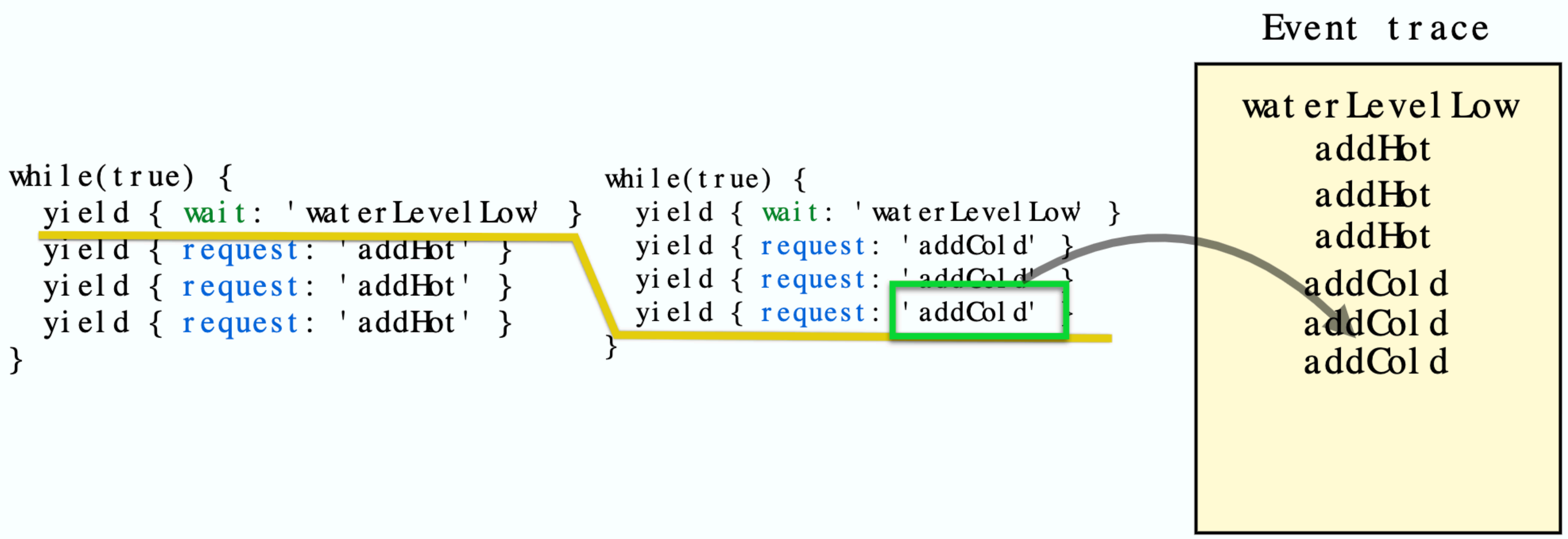


```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
}
```



```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
}
```

```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addCold' }
  yield { request: 'addCold' }
  yield { request: 'addCold' }
}
```



```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
}
```

```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addCold' }
  yield { request: 'addCold' }
  yield { request: 'addCold' }
}
```

Event trace

```
waterLevelLow
addHot
addHot
addHot
addCold
addCold
addCold
```

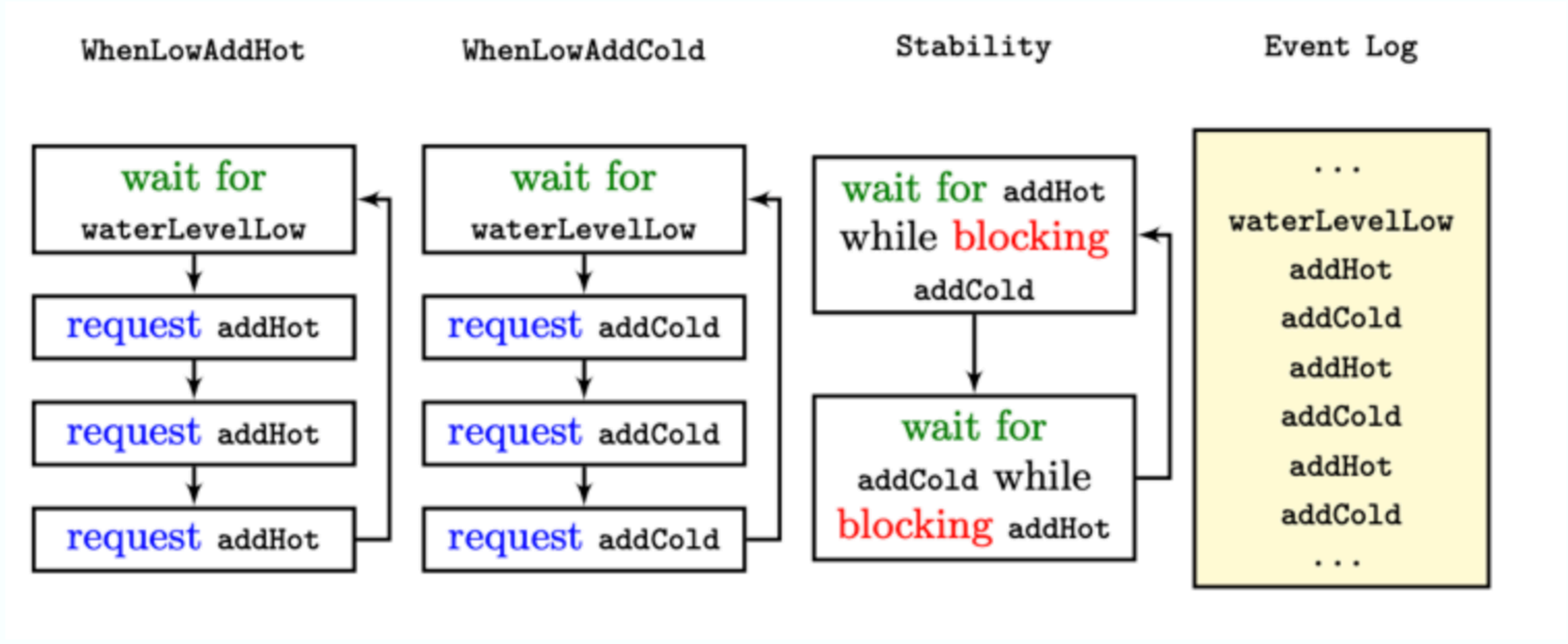
```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
  yield { request: 'addHot' }
}
```

```
while(true) {
  yield { wait: 'waterLevelLow' }
  yield { request: 'addCold' }
  yield { request: 'addCold' }
  yield { request: 'addCold' }
}
```

```
while (true) {
  yield {
    wait: 'addHot',
    block: 'addCold'
  }
  yield {
    wait: 'addCold',
    block: 'addHot'
  }
}
```

Event trace

```
waterLevelLow
addHot
addCold
addHot
addCold
addHot
addCold
```


WhenLowAddHot
WhenLowAddCold
Stability
Event Log
wait for waterLevelLow
request addHot
request addHot
request addHot
wait for waterLevelLow
request addCold
request addCold
request addCold
wait for addHot while blocking addCold
wait for addCold while blocking addHot
...
waterLevelLow
addHot
addCold
addHot
addCold
addHot
addCold
...

Event trace

Event trace

Event trace

- cardInserted
- cardIsValid
- loadingAccount
- accountLoaded
- waitingForPin

Show advertisement before the account is loaded

Event trace

- cardInserted
- cardIsValid
- showAd
- adShown
- loadingAccount
- accountLoaded
- waitingForPin

Show advertisement before the account is loaded

What currently happens

- cardInserted
- cardIsValid
- loadingAccount
- accountLoaded
- waitingForPin

What we want

- cardInserted
- cardIsValid
- showAd
- adShown
- loadingAccount
- accountLoaded
- waitingForPin

Show advertisement before the

```
while (true) {
  yield { wait: 'cardIsValid' }
  yield {
    wait: 'adShown',
    block: 'loadingAccount'
  }
}
```

```
while (true) {
  yield { wait: 'cardIsValid' }
  yield { request: 'showAd' }
  yield showAdvertisment()
  yield { request: 'adShown' }
}
```

What currently happens

What we want

- cardInserted
- cardIsValid
- showAd
- adShown
- loadingAccount
- accountLoaded
- waitingForPin

Show advertisement before the

oa

```
while (true) {
  yield { wait: 'cardIsValid' }
  yield {
    wait: 'adShown',
    block: 'loadingAccount'
  }
}
```

```
while (true) {
  yield { wait: 'cardIsValid' }
  yield { request: 'showAd' }
  yield showAdvertisment()
  yield { request: 'adShown' }
}
```

What currently happens

- cardInserted
- cardIsValid
- showAd
- adShown
- loadingAccount
- accountLoaded
- waitingForPin

What we want

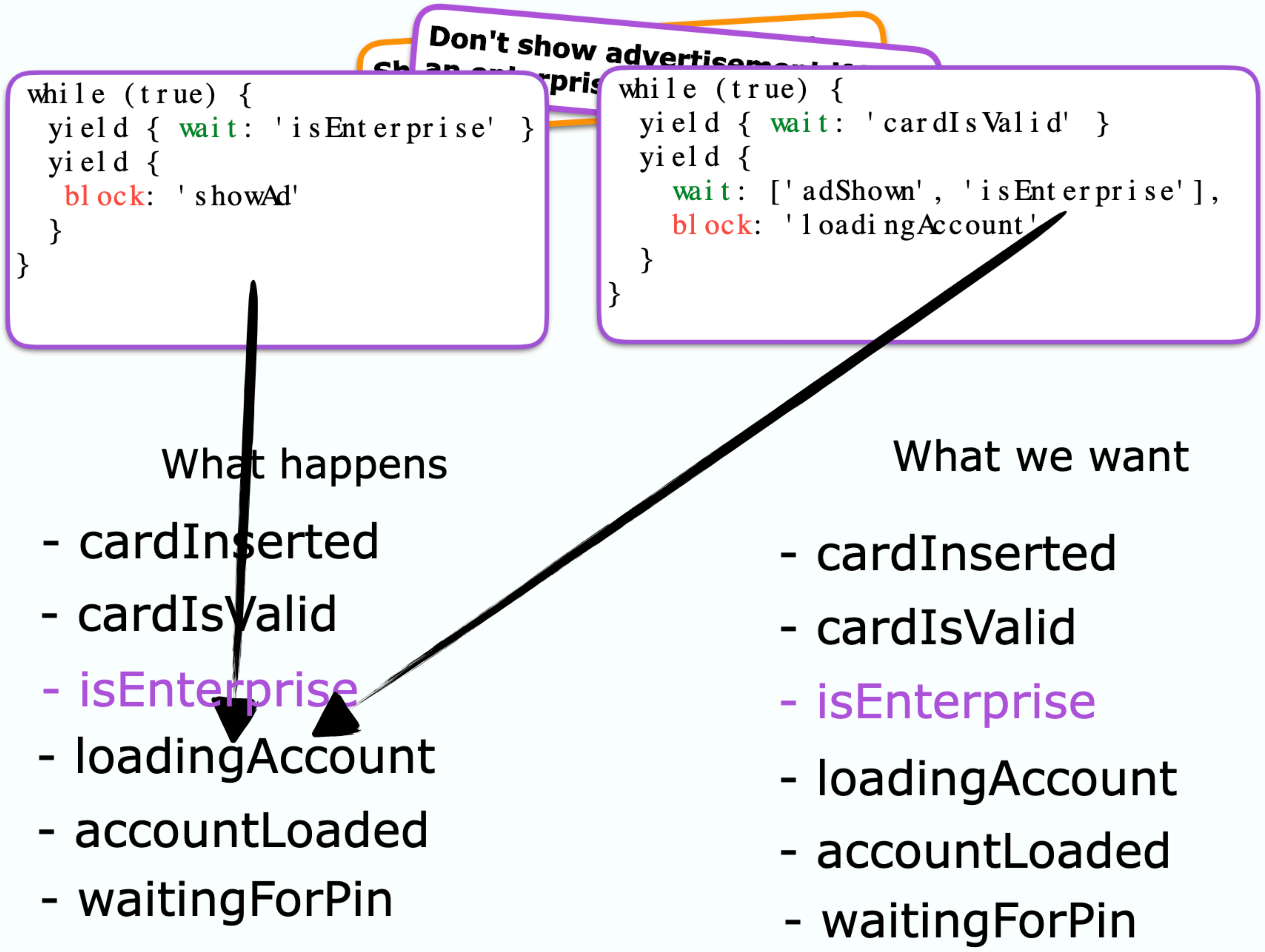
- cardInserted
- cardIsValid
- showAd
- adShown
- loadingAccount
- accountLoaded
- waitingForPin

Sh
accou

Don't show advertisement if it's an enterprise user

Event trace

- cardInserted
- cardIsValid
- isEnterprise
- sh d
- ad wn
- loadingAccount
- accountLoaded
- waitingForPin



```
while (true) {
  yield { wait: 'isEnterprise' }
  yield {
   block: 'showAd'
  }
}
```

```
while (true) {
  yield { wait: 'cardIsValid' }
  yield {
    wait: ['adShown', 'isEnterprise'],
    block: 'loadingAccount'
  }
}
```

What happens

- cardInserted
- cardIsValid
- isEnterprise
- loadingAccount
- accountLoaded
- waitingForPin

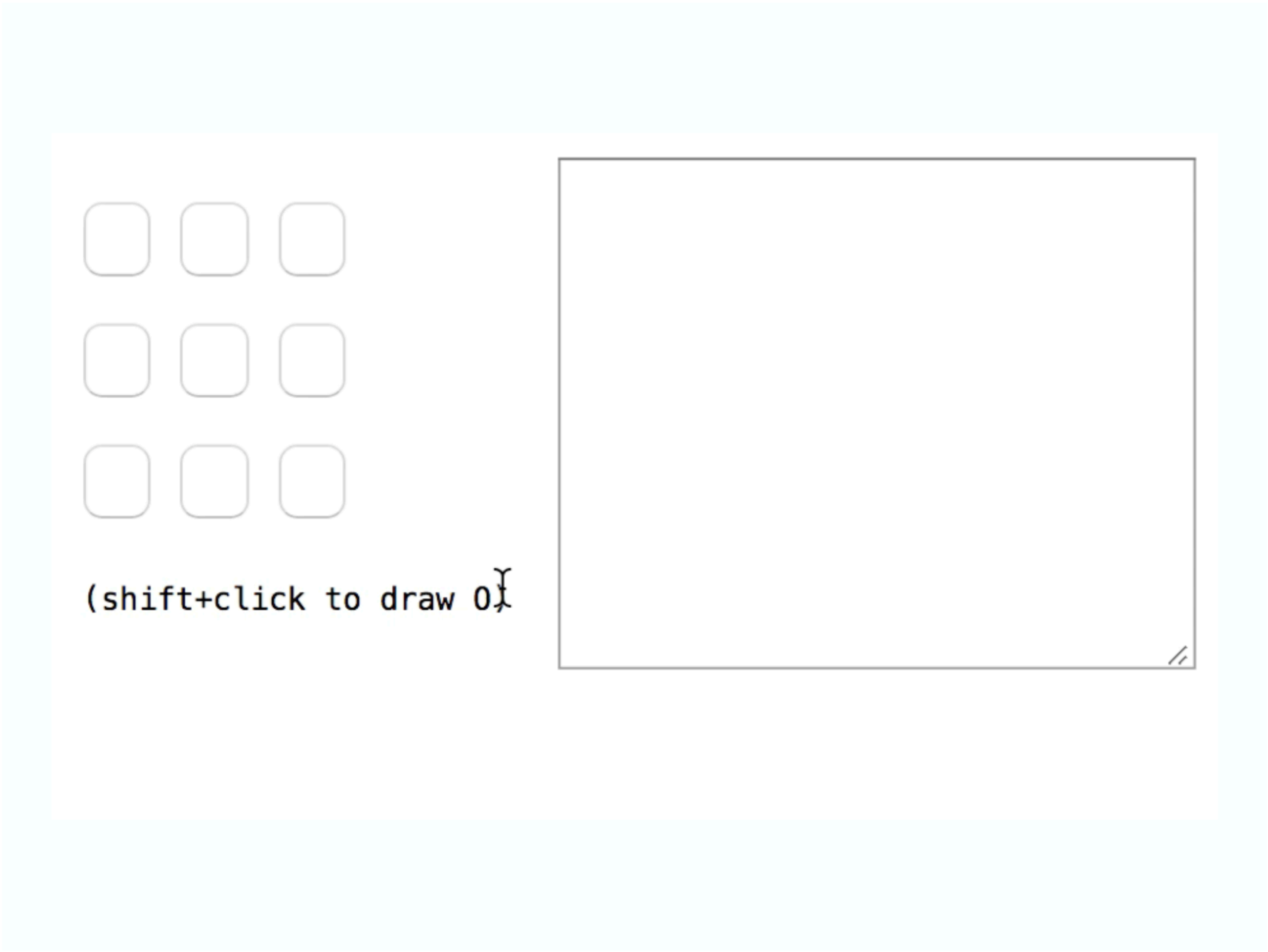
What we want

- cardInserted
- cardIsValid
- isEnterprise
- loadingAccount
- accountLoaded
- waitingForPin

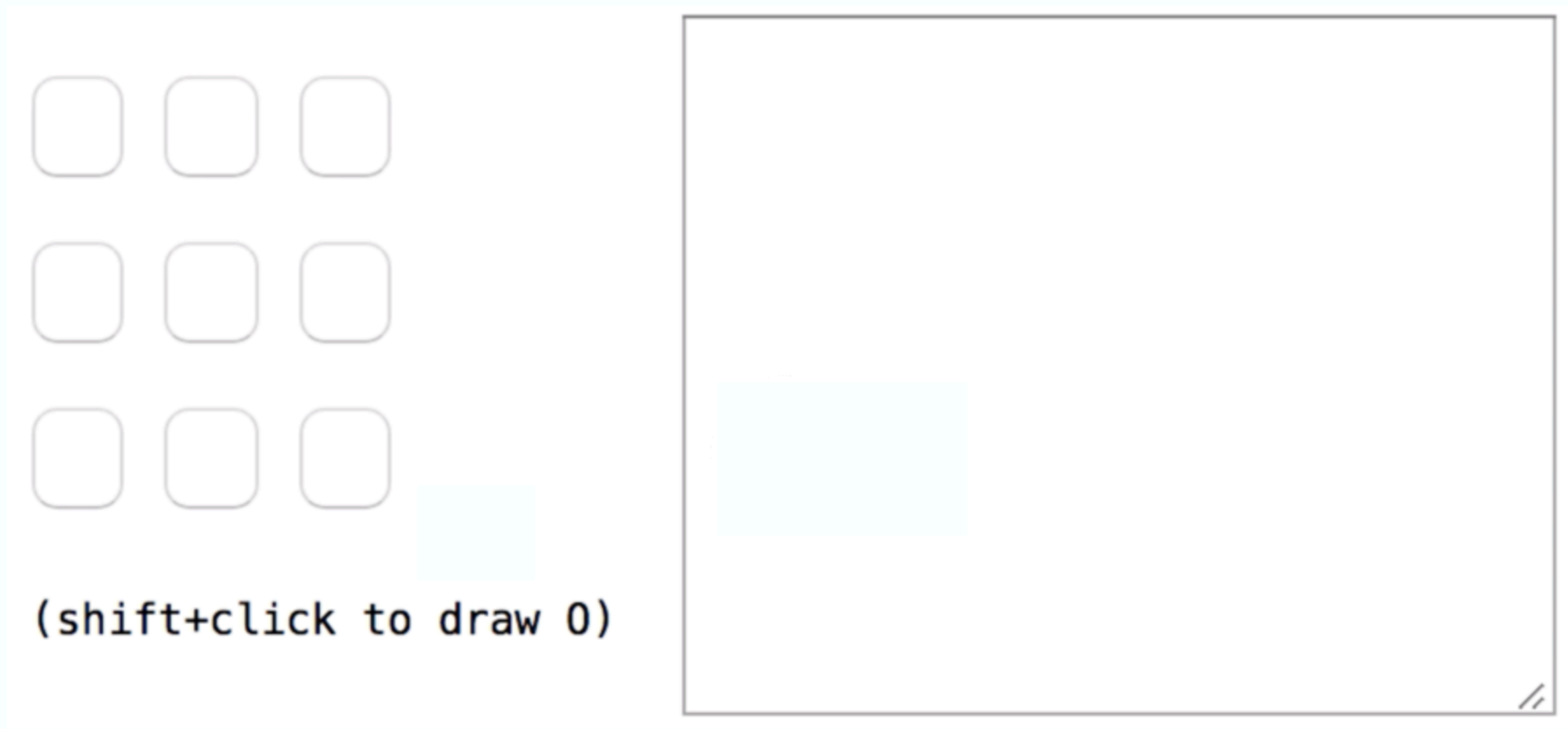
Main insight

Newly added code can change how old code is executed

Let’s play Tic Tac Toe


(shift+click to draw O)

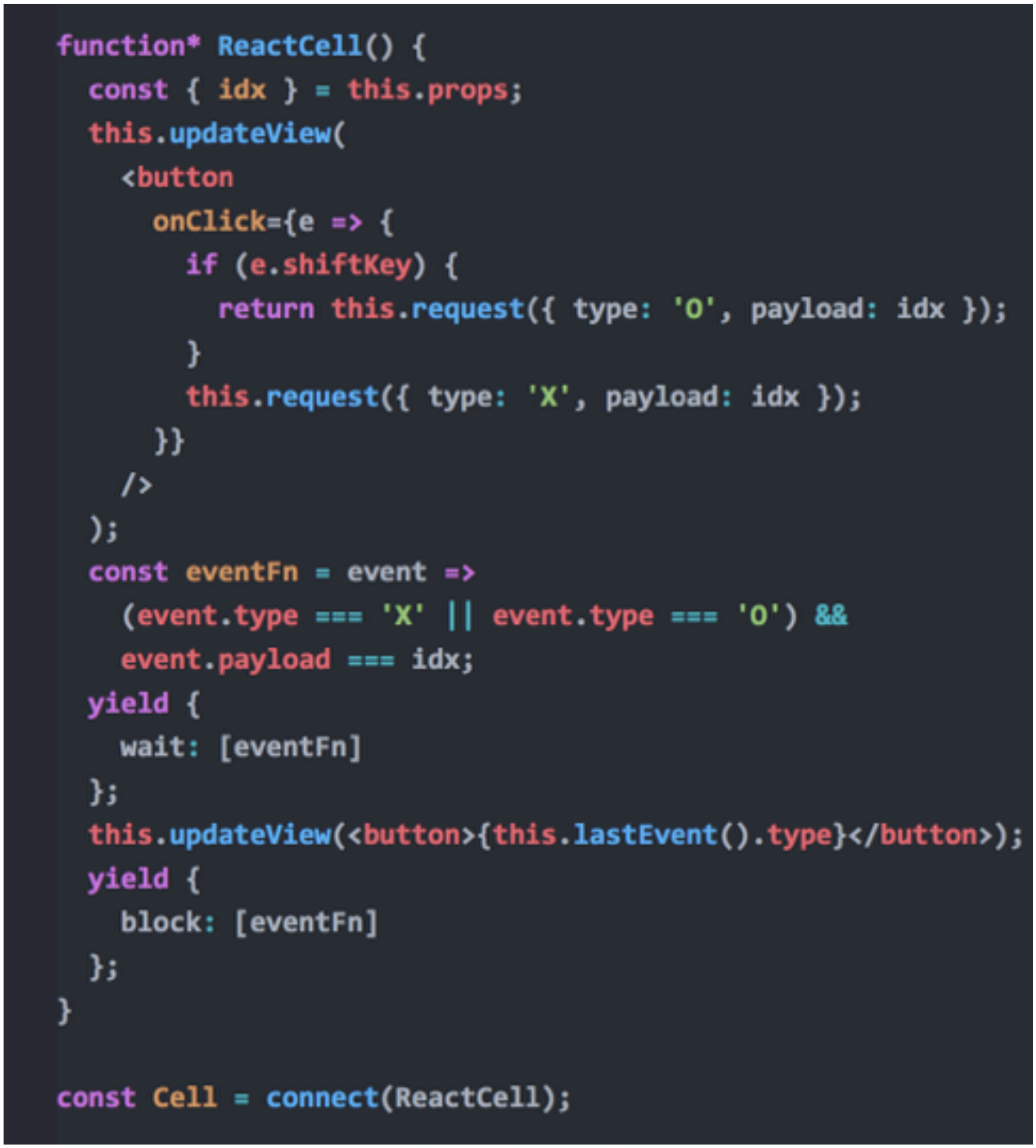
```
{ type: "X", payload: 8 }          { type: "X", payload: 8 }
{ type: "O", payload: 5 }          { type: "O", payload: 5 }
{ type: "X", payload: 2 }          { type: "X", payload: 2 }
{ type: "X", payload: 4 }   -->    { type: "O", payload: 3 }
{ type: "X", payload: 6 }          { type: "X", payload: 4 }
{ type: "O", payload: 3 }          { type: "O", payload: 0 }
{ type: "O", payload: 0 }          { type: "X", payload: 6 }
```



b-thread: EnforcePlayerTurns

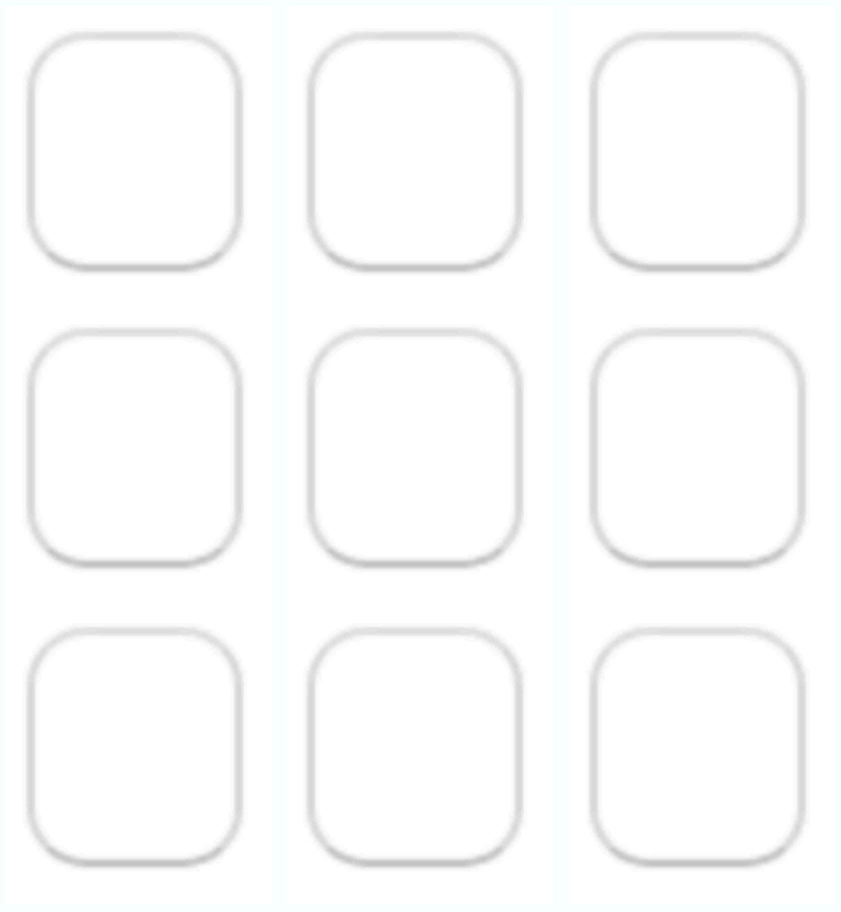
```
function* enforcePlayerTurns() {
  while (true) {
    yield { wait: 'X', block: 'O' };
    yield { wait: 'O', block: 'X' };
  }
}
```

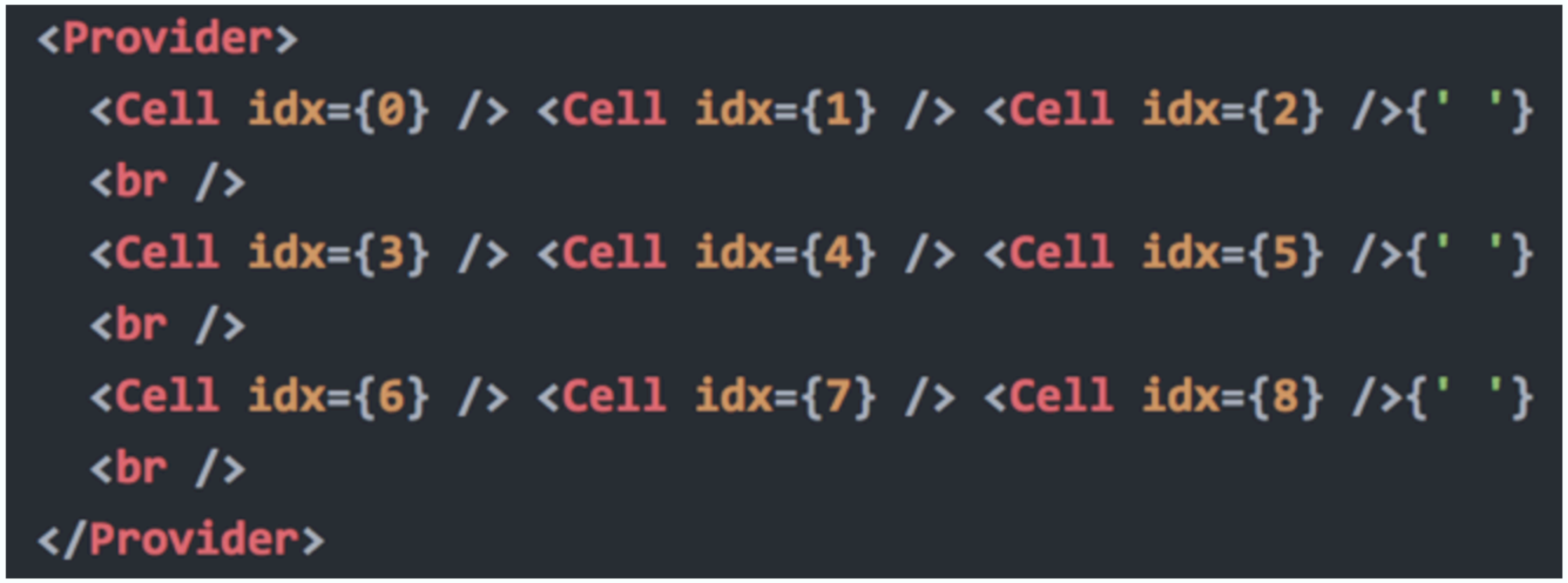
How to use it with React?



```
function* ReactCell() {
  const { idx } = this.props;
  this.updateView(
    <button
      onClick={e => {
        if (e.shiftKey) {
          return this.request({ type: 'O', payload: idx });
        }
        this.request({ type: 'X', payload: idx });
      }}
    />
  );
  const eventFn = event =>
    (event.type === 'X' || event.type === 'O') &&
    event.payload === idx;
  yield {
    wait: [eventFn]
  };
  this.updateView(<button>{this.lastEvent().type}</button>);
  yield {
    block: [eventFn]
  };
}

const Cell = connect(ReactCell);
```





```
<Provider>
  <Cell idx={0} /> <Cell idx={1} /> <Cell idx={2} />{' '}
  <br />
  <Cell idx={3} /> <Cell idx={4} /> <Cell idx={5} />{' '}
  <br />
  <Cell idx={6} /> <Cell idx={7} /> <Cell idx={8} />{' '}
  <br />
</Provider>
```

Event trace = behavior

Making changes to these projects didn't require us to understand how they were implemented. We simply fed the system new scenarios based on the event traces we wanted to see happen.

Driving directions

- Start driving on I-78 W [135 mi]
- Merge onto I-81 S [36.6 mi]
- Take ramp onto I-76 W [152 mi]
- Merge onto I-70 W [613 mi]
- Merge onto I-44 W [497.2 mi]
- Continue to I-40 W [1,214 mi]
- Merge onto I-15 S [72.6 mi]
- Merge onto I-10 W [38.9 mi]

Daily schedule

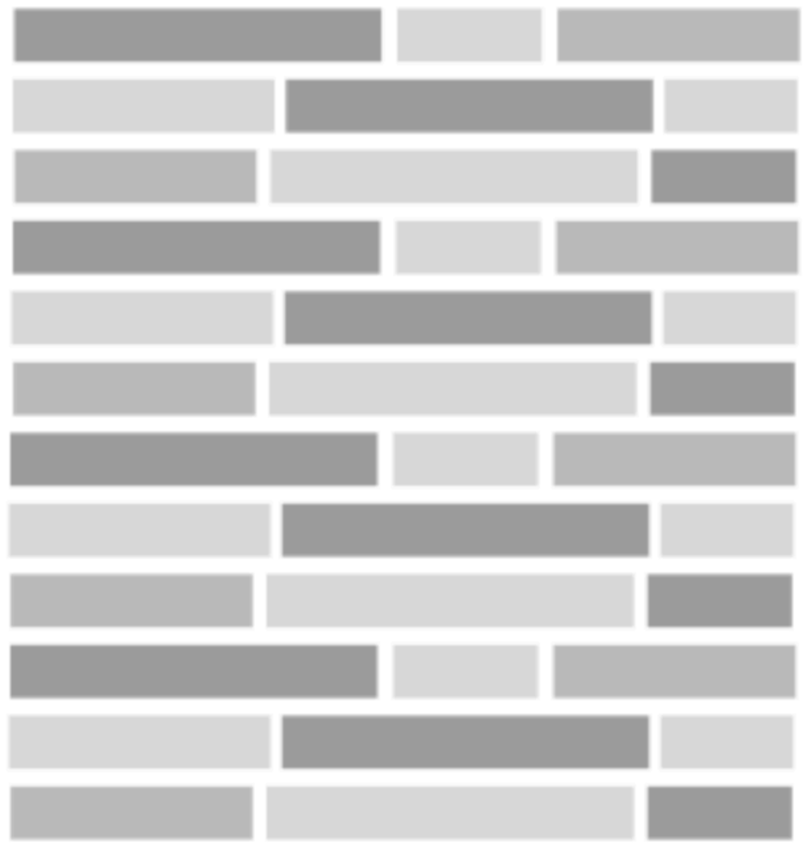
Repeatedly:
- Drive for 5 h; look for restaurant
- Stop the car; have lunch
- Drive for 5 h; look for restaurant
- Stop the car; have dinner
- Drive for 2 h; look for hotel
- Stop the car
- Sleep until morning

A 6-day trip from NY to LA

scenarios

aka: requirements, stories, threads, b-threads, etc.

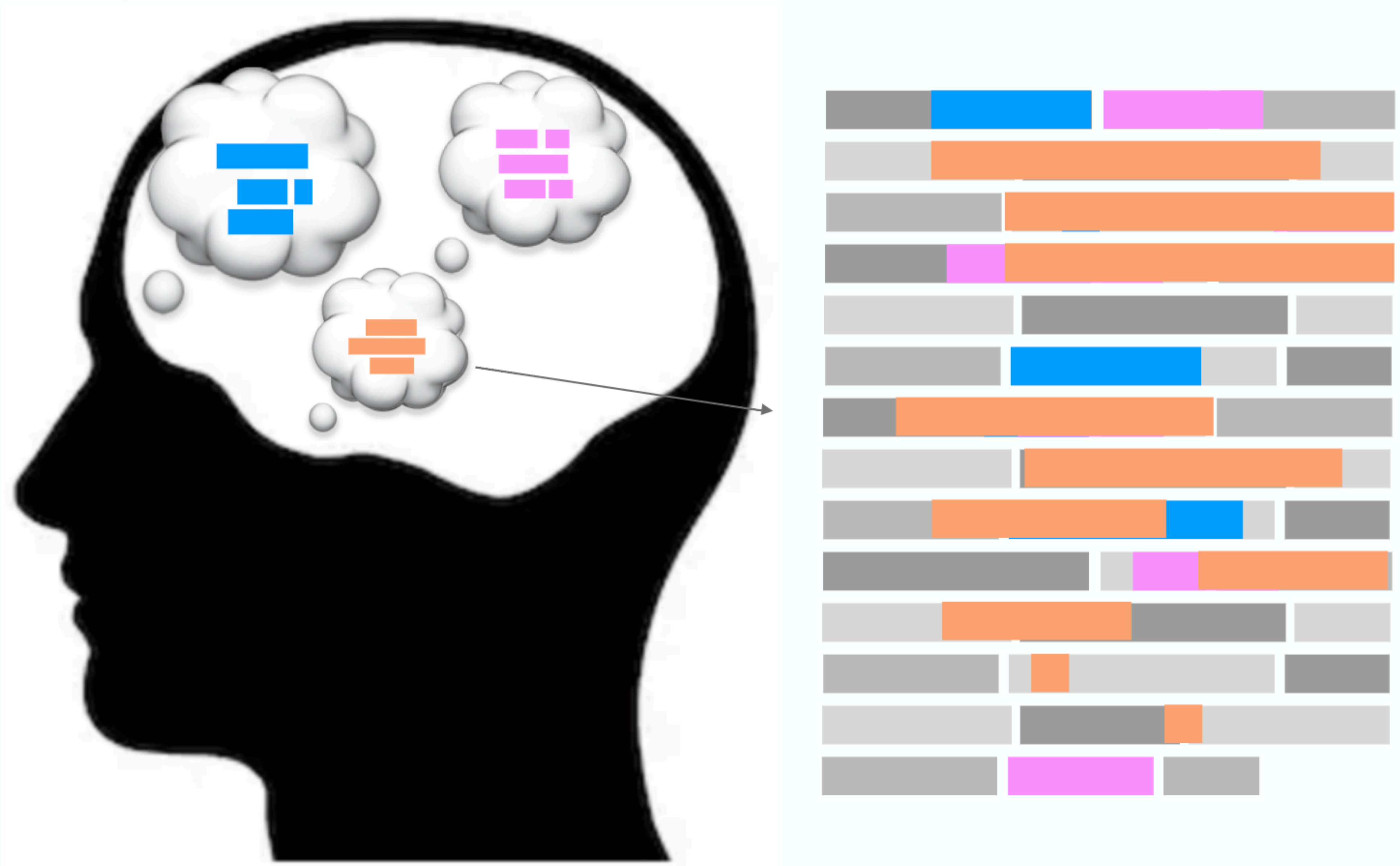
code



scenarios

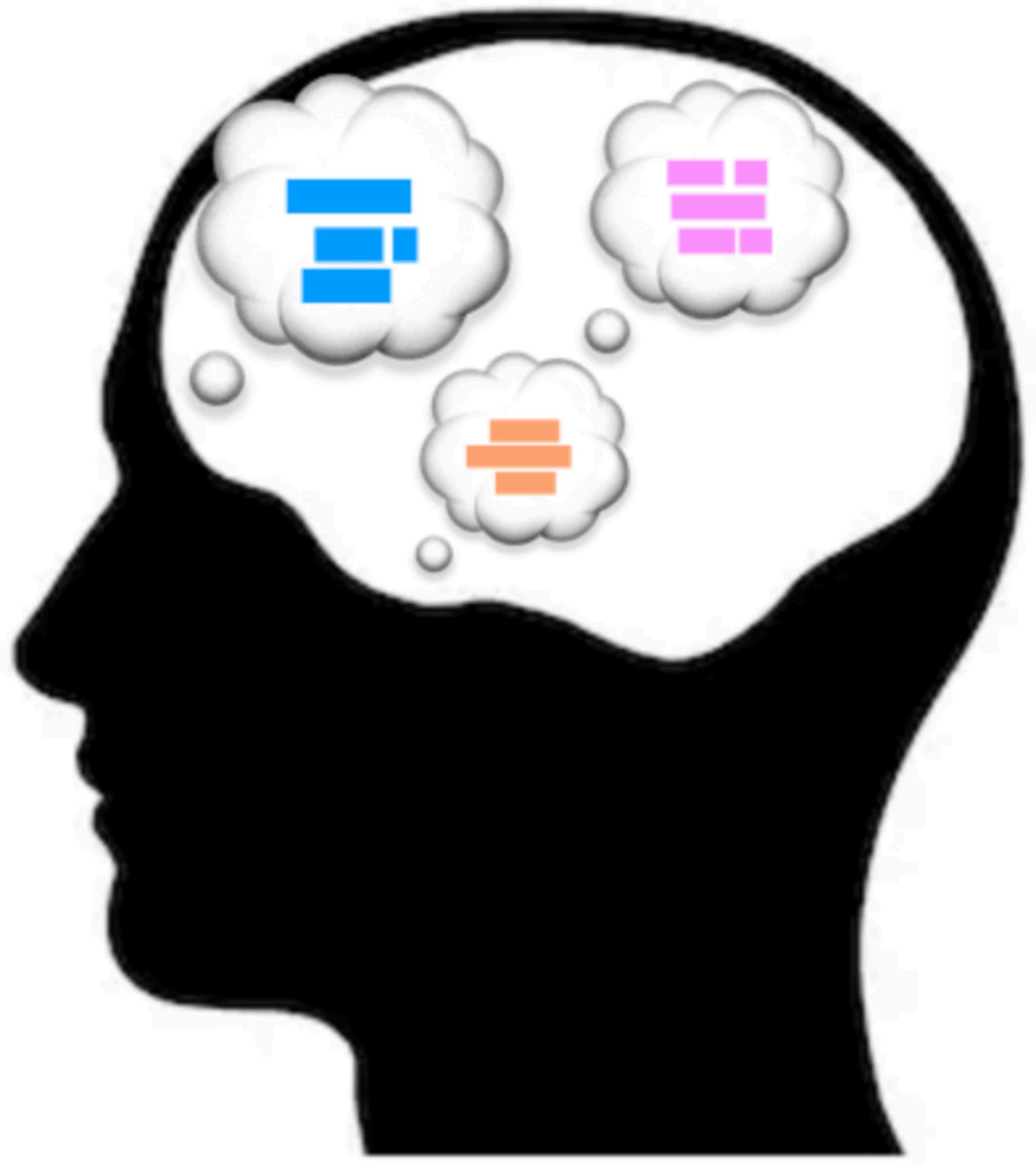
aka: requirements, stories, threads, b-threads, etc.

code

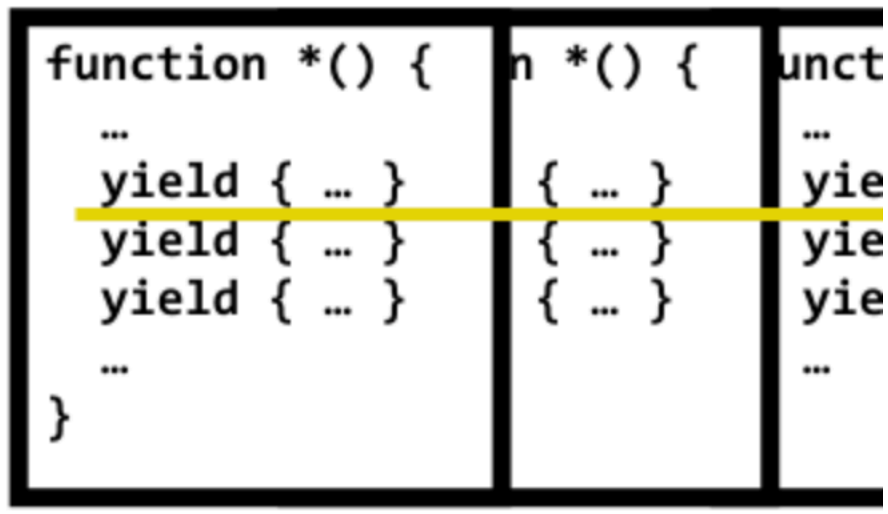


scenarios

aka: requirements, stories, threads, b-threads, specs, etc.

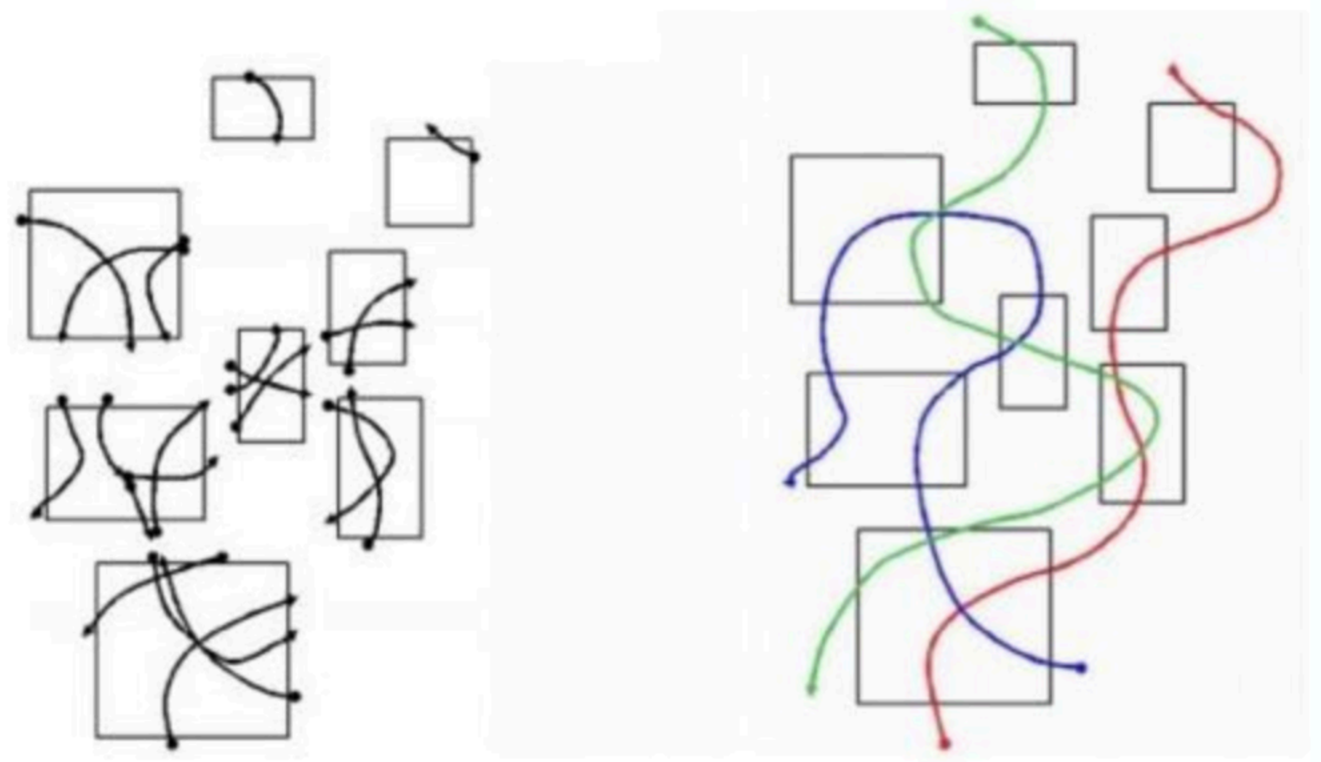


b-threads



```
function *() {   n *() {   unct
  …                          …
  yield { … }     { … }      yie
  yield { … }     { … }      yie
  yield { … }     { … }      yie
  …                          …
}
```

Harel, David, and Assaf Marron. "The quest for runware: on compositional, executable and intuitive models." Software & Systems Modeling 11.4 (2012): 599-608.



In our vision, the units of the specification and models are not assembled in detail like resistors or chips on a computer board, or methods and fields in an OO-programming object class. The interweaving of behavioral modules will be facilitated by their reference to common aspects of system behavior described using shared vocabularies (for example, common events), and not via mutual awareness and direct communication between components. From the point of view of such a module, the other modules could be transparently replaced by new ones.

Takeaways

- “Moving” the imaginary line by requesting, waiting and blocking events allows for incremental development
- Alignment with requirements and how humans think about behavior
- B-threads are “piled-atop” with no component-specific interface, connectivity or ordering requirements

Cons

- Different way of thinking about programming.
- Ironically, more natural ways of programming are perceived as unnatural because of our past training.
- Does it scale with thousands or million concurrent b-threads?
- Not currently used by many people.
- Lack of best-practices, tools, community, etc.

Resources

- https://github.com/lmatteis/behavioral
- https://github.com/lmatteis/react-behavioral
- https://github.com/lmatteis/redux-behavioral

- Harel, David, Assaf Marron, and Gera Weiss. "Behavioral programming." Commun. ACM 55.7 (2012): 90-100.