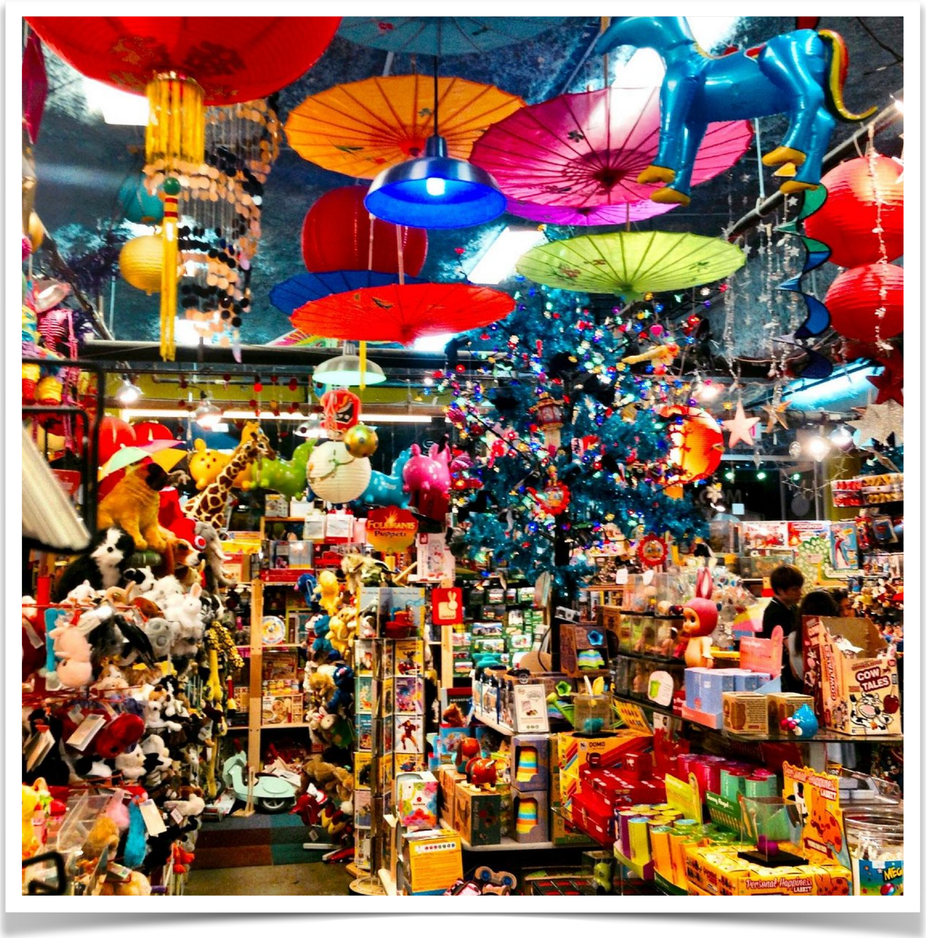


The Power of Toys

Lambda Days 2022

Vouch.io

- Clojurist

- Clojurist
  - Lead developer of ClojureScript ~11 years

- Clojurist
  - Lead developer of ClojureScript ~11 years
- Functional programming in anger for the last 8 years

1 year

5 years

10 years

10+ years

toy (n.)

c. 1300, "amorous playing, sport," later "piece of fun or entertainment" (c. 1500), "thing of little value, trifle" (1520s), and "thing for a child to play with" (1580s). Of uncertain origin, and there may be more than one word here. Compare Middle Dutch *toy*, Dutch *tuig* "tools, apparatus; stuff, trash," in *speeltuig* "play-toy, plaything;" German *Zeug* "stuff, matter, tools," *Spielzeug* "plaything, toy;" Danish *tøj*, Swedish *tyg* "stuff, gear." Applied as an adjective to things of diminutive size, especially dogs, from 1806. *Toy-boy* is from 1981.

toy model (n.)

a simplified set of objects and equations relating them so that they can nevertheless be used to understand a mechanism that is also useful in the full, non-simplified theory.

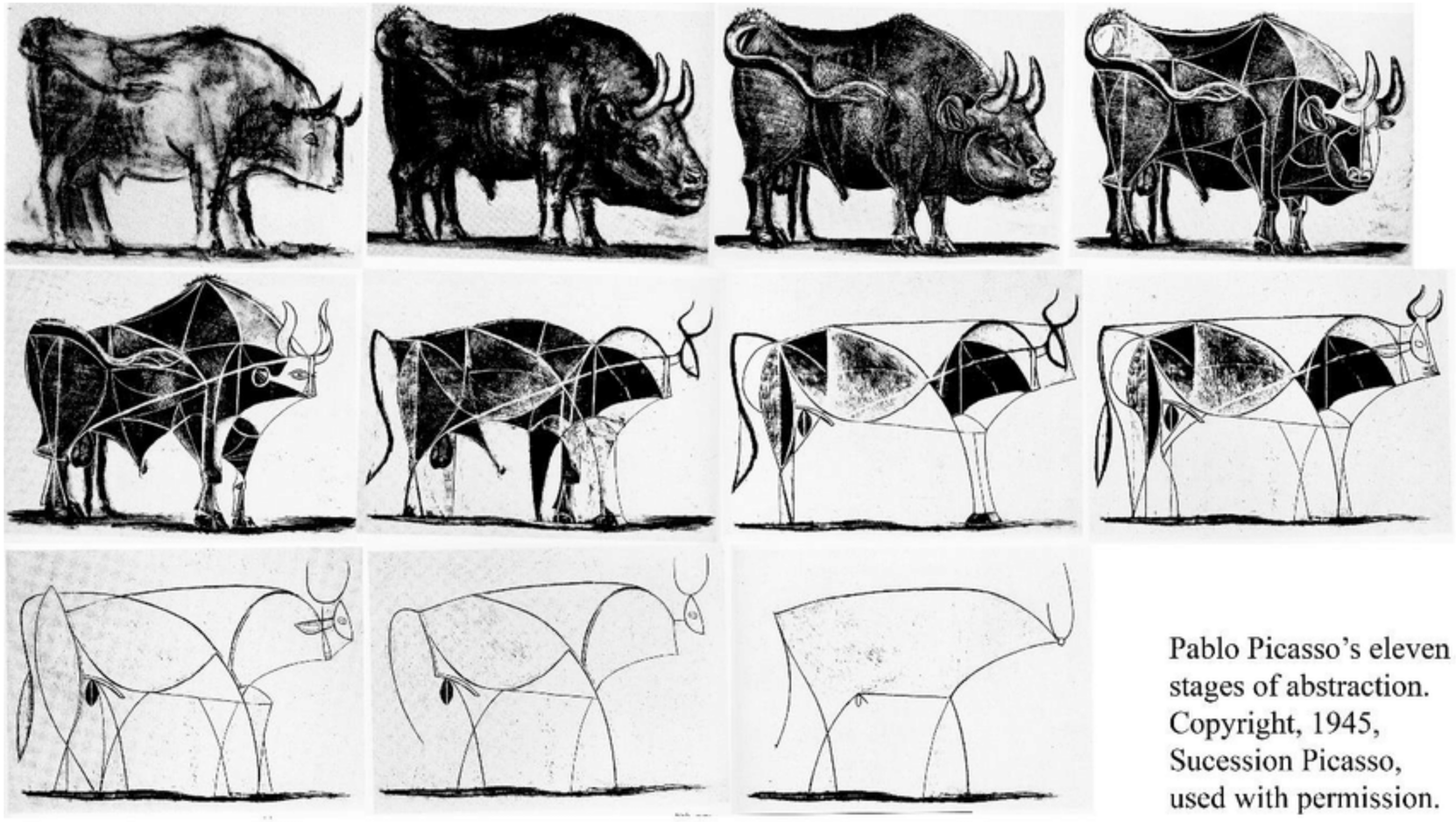

POST
1932

- Value Oriented Programming

- Value Oriented Programming
  - Including the databases!

- Value Oriented Programming
  - Including the databases!
- Symbolic representation / evaluators

- Value Oriented Programming
  - Including the databases!
- Symbolic representation / evaluators
- Stateful Property Based Testing



Pablo Picasso's eleven stages of abstraction. Copyright, 1945, Sucession Picasso, used with permission.

Abstraction - not as process of *generalization* (a risky endeavor) - but process of *toyification*

Value Oriented Programming

Value Oriented Programming

- Programming with values

Value Oriented Programming

- Programming with values
- Functions take state as value and return a new state

Value Oriented Programming

- Programming with values
- Functions take state as value and return a new state
- Make a toy. Everything of interest is in the state - no externals. A single "database"

Symbolic Commands

Symbolic Commands

- Toy model language - Do X, Do Y …

Symbolic Commands

- Toy model language - Do X, Do Y ...
- No arguments other than *names*

Symbolic Commands

- Toy model language - Do X, Do Y …
- No arguments other than *names*
  - `[:share-key :person-a :person-b]`

Symbolic Commands

- Toy model language - Do X, Do Y …
- No arguments other than *names*
  - `[:share-key :person-a :person-b]`
- No programmatic constructs of any kind

Stateful Property Based Testing

Stateful Property Based Testing

- Verify properties of interest of the toy model by *generating* instructions

Stateful Property Based Testing

- Verify properties of interest of the toy model by *generating* instructions
- Avoids implicit dependencies / constraints

Stateful Property Based Testing

- Verify properties of interest of the toy model by *generating* instructions
- Avoids implicit dependencies / constraints
- Toy-ify the search problem to avoid trivial scenarios.

PBT in a Nutshell

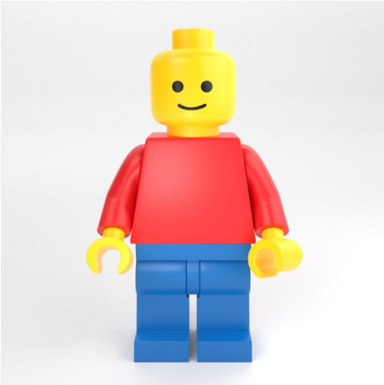
PBT in a Nutshell

- Instead of unit testing with hard coded values, *generate* values - with good distribution but also reproducibility (PRNG)

PBT in a Nutshell

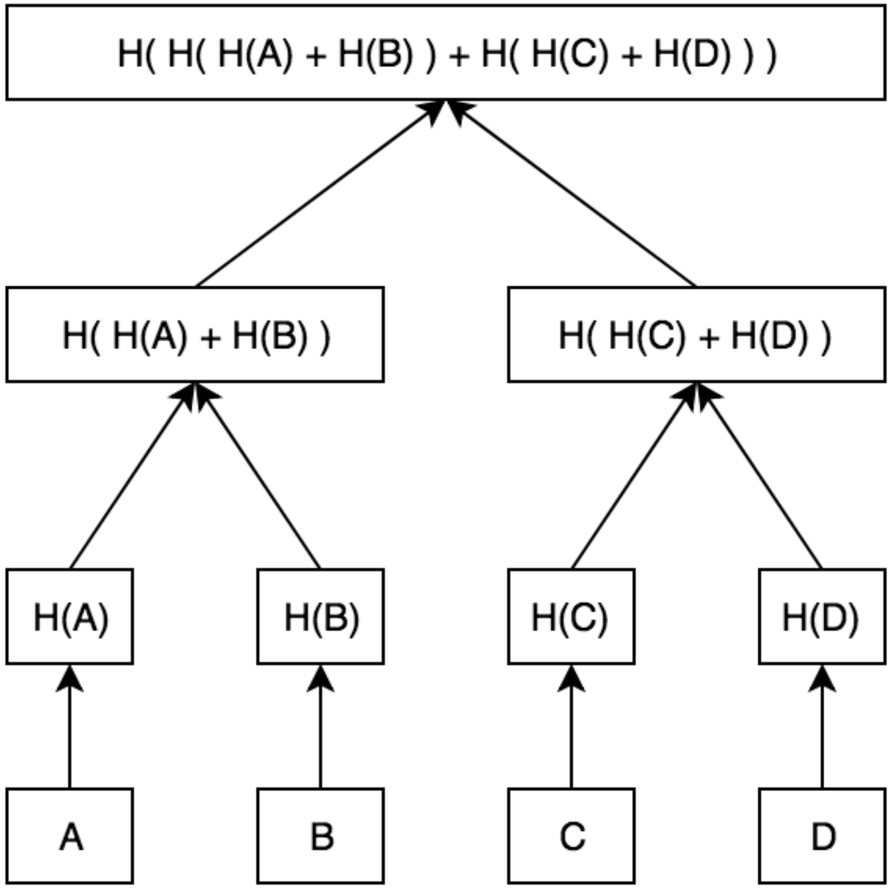
- Instead of unit testing with hard coded values, *generate* values - with good distribution but also reproducibility (PRNG)
- If a test fails - search for the “smallest” input that causes the failure - “shrinking”

Identities & Assets on the Blockchain

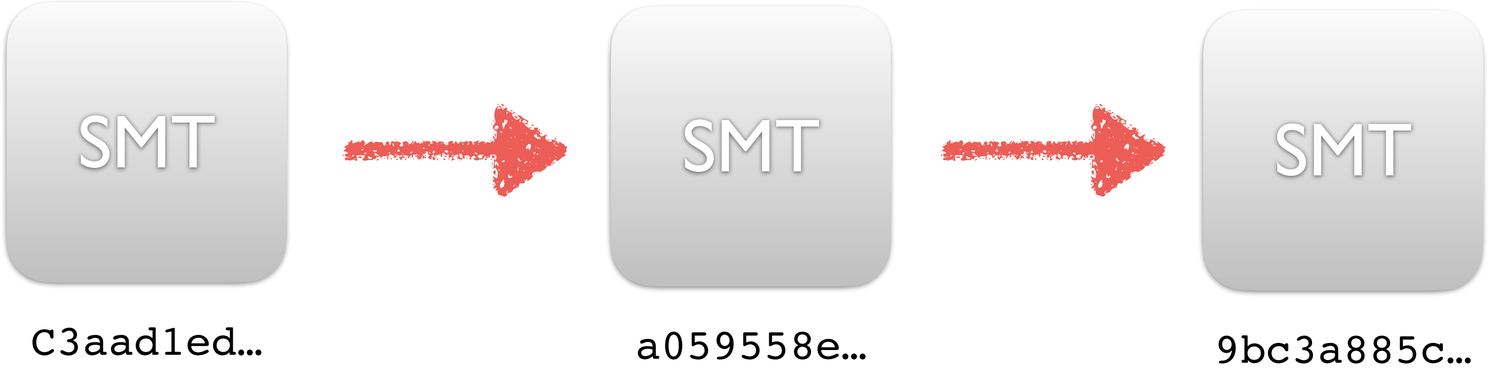


SMT

C3aad1ed…

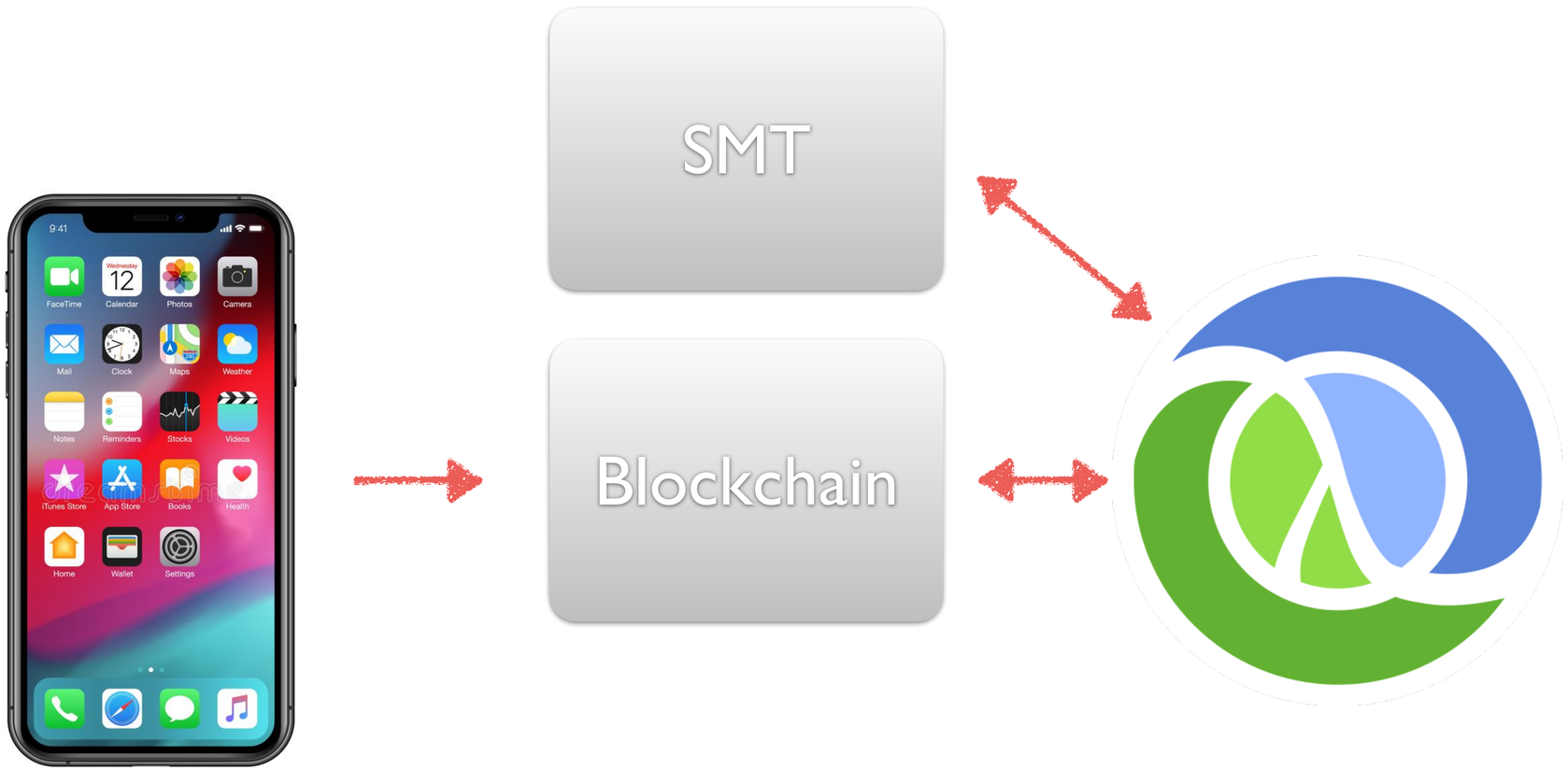


Sparse Merkle Tree (SMT)


SMT
SMT
SMT
C3aad1ed...
a059558e...
9bc3a885c...

- Every transaction to the blockchain is recorded

- Every transaction to the blockchain is recorded
- Can bring up a new node to the same state as the others by playing back transactions

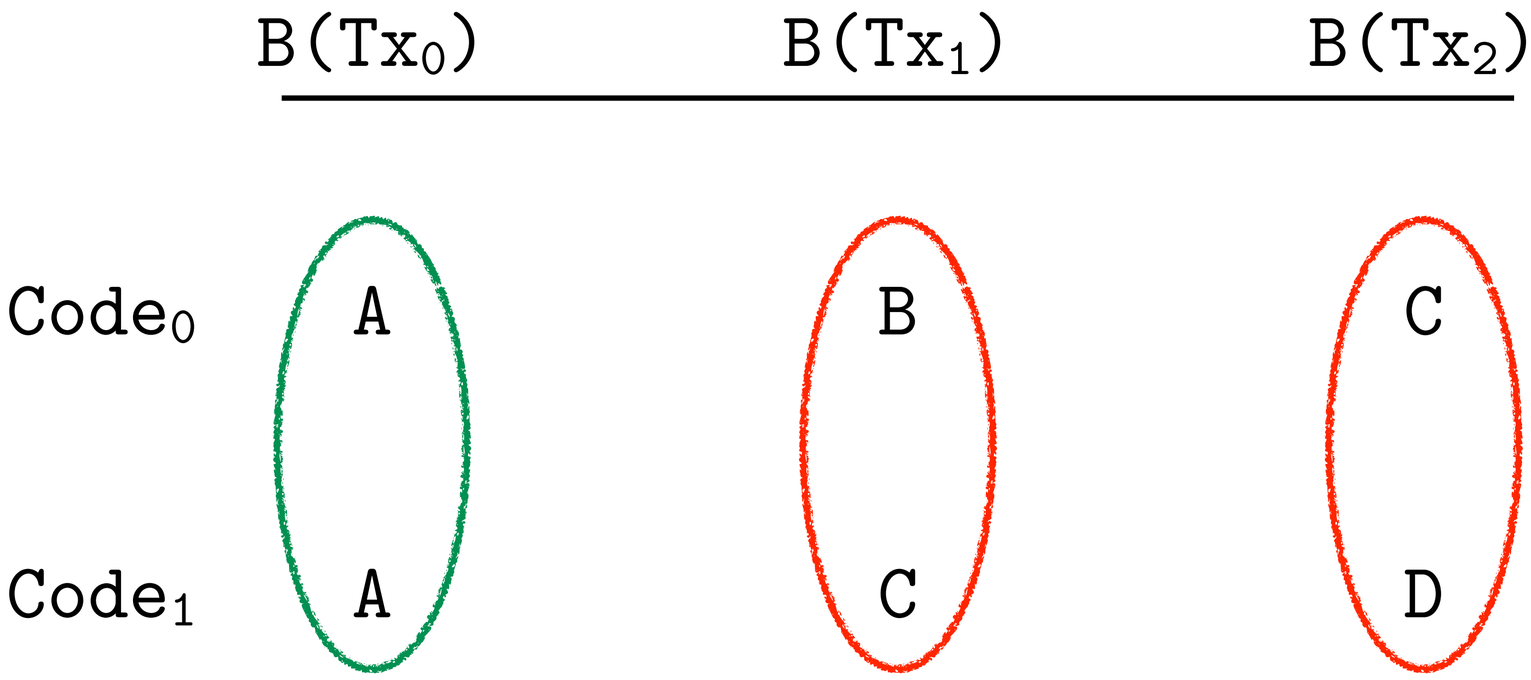

SMT
Blockchain

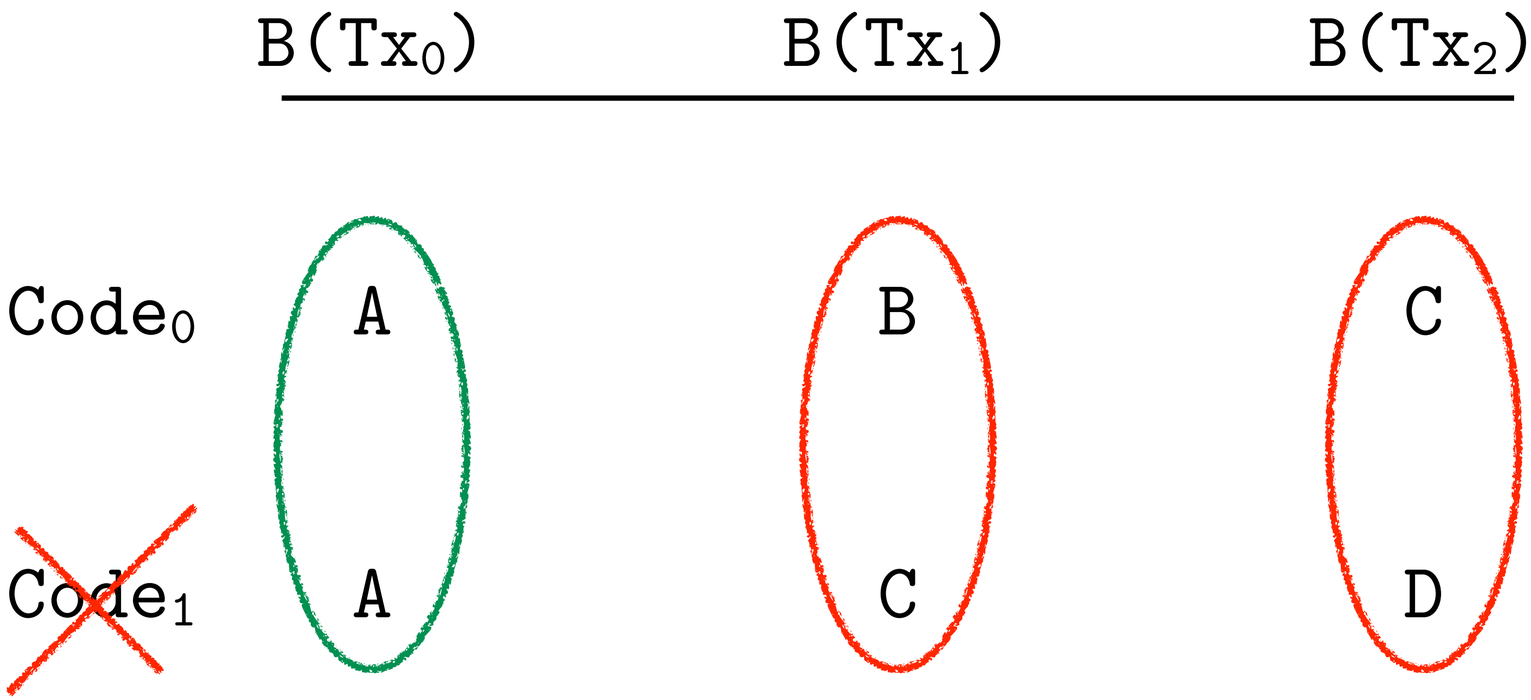
SMT

Blockchain

```
~/t/git-diff ❯❯❯ diff test.txt sample.txt
diff --git a/test.txt b/sample.txt
index ad01390..5629924 100644
--- a/test.txt
+++ b/sample.txt
@@ -1,3 +1,3 @@
-This is a test
-    * Hello
-    * world!
+This is a sample
+  * Hello
+  * universe!
~/t/git-diff ❯❯❯
```

	$B(Tx_0)$	$B(Tx_1)$	$B(Tx_2)$
$Code_0$	A	B	C
$Code_1$	A	C	D


B(Tx0)
B(Tx1)
B(Tx2)
Code0
Code1
A
A
B
C
C
D


B(Tx0)
B(Tx1)
B(Tx2)
Code0
Code1
A
A
B
C
C
D

- Feature flags - any new branch that might alter the SMT (even if it's a bug fix!) must be behind a feature flag

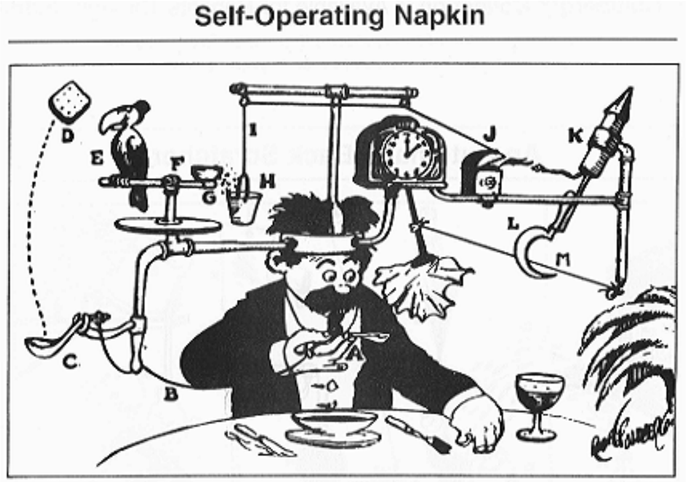
- Feature flags - any new branch that might alter the SMT (even if it's a bug fix!) must be behind a feature flag

- Feature flag toggling must be a blockchain transaction

- How can we increase confidence that old behavior is always supported?

- How can we increase confidence that old behavior is always supported?

- How can we increase confidence that when a feature has an undesirable affect that we can downgrade?



Integration testing

- Implement a toy model of the clients and assets and their interactions

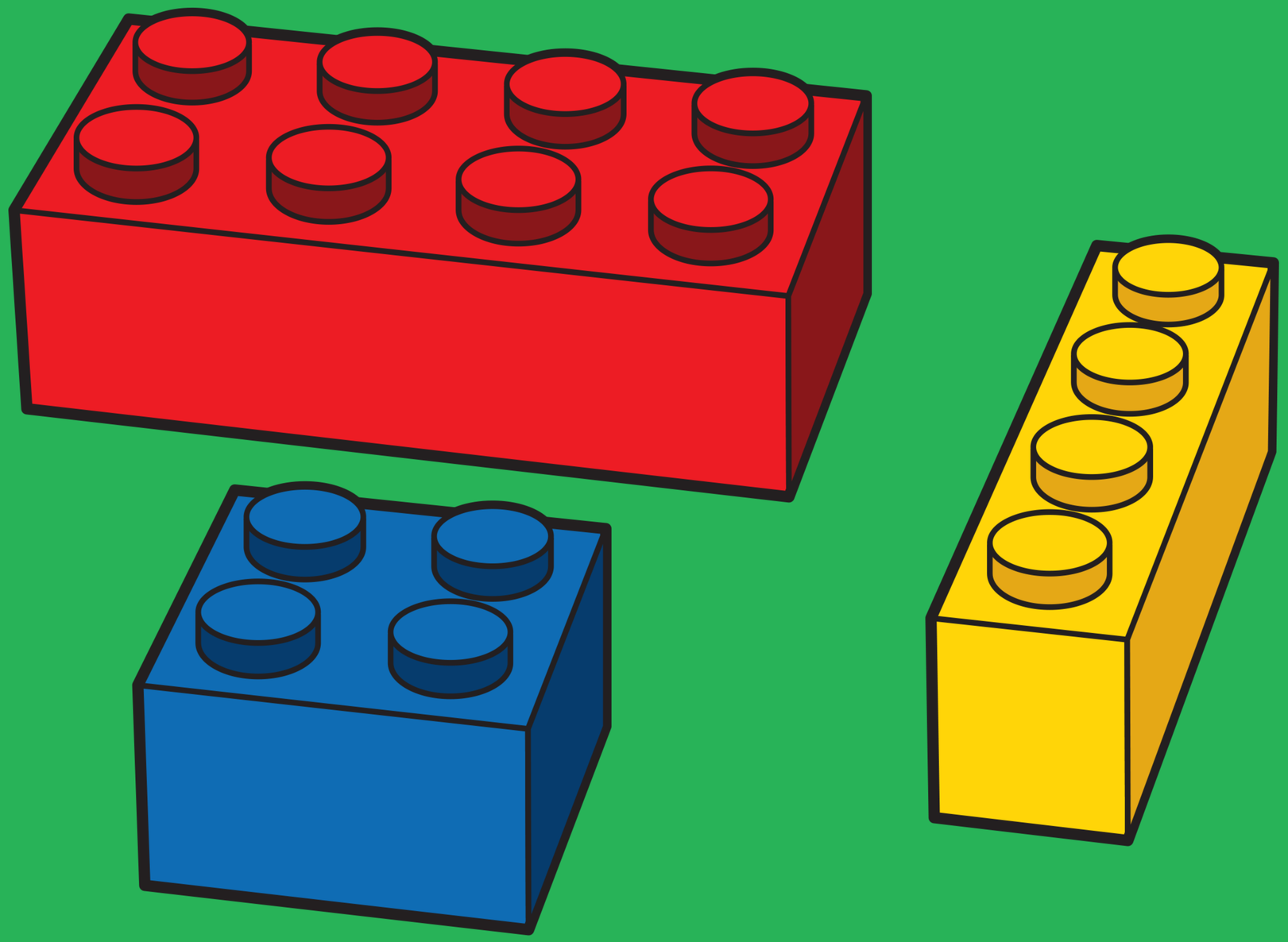
- Implement a toy model of the clients and assets and their interactions

- All toy model functions take toy model state (which is a value) as the first argument

$$f(\mathrm{A},\ \mathrm{B},\ \mathrm{C},\ \ldots) \rightarrow \mathrm{Z}$$

$$f(\mathrm{S}_0,\ \mathrm{A},\ \mathrm{B},\ \mathrm{C},\ \ldots) \rightarrow \mathrm{S}_1$$

```
{:db                DatomicDB
 :smt               VouchSMT
 :crypto-devices    {:customer-a …}
 :assets            {:vehicle-a …}}
```



```clojure
(defn installer+dealer+customer+one-asset
  "Return a state with one installer, one dealer, one customer, and one asset
  enrolled by the installer."
  [{:keys [org-data vin customer-mobile-number]}]
  (-> (test-org/persist (state/create tu/*conn*) org-data)
    (state/add-crypto-device :installer
      (test-org/add-token-data (crypto-device/create) org-data))
    (crypto-device/enroll-oidc :installer)
    (test-org/endorse-membership :installer [:enroll-asset-device])
    (test-assets/enroll-asset+device :installer
      (test-assets/create-asset {:asset-mfg-id vin}))
    (state/add-crypto-device :dealer
      (test-org/add-token-data (crypto-device/create) org-data))
    (state/add-crypto-device :customer
      (crypto-device/create {:token-data {:mobile-number customer-mobile-number}}))
    (crypto-device/enroll-oidc-all)
    (state/tag-state :all-enrolled)
    (test-org/endorse-membership :dealer [:operate :transfer-owner])))
```

Functional Scenarios

Functional Scenarios

- We now write scenarios close to the business language

Functional Scenarios

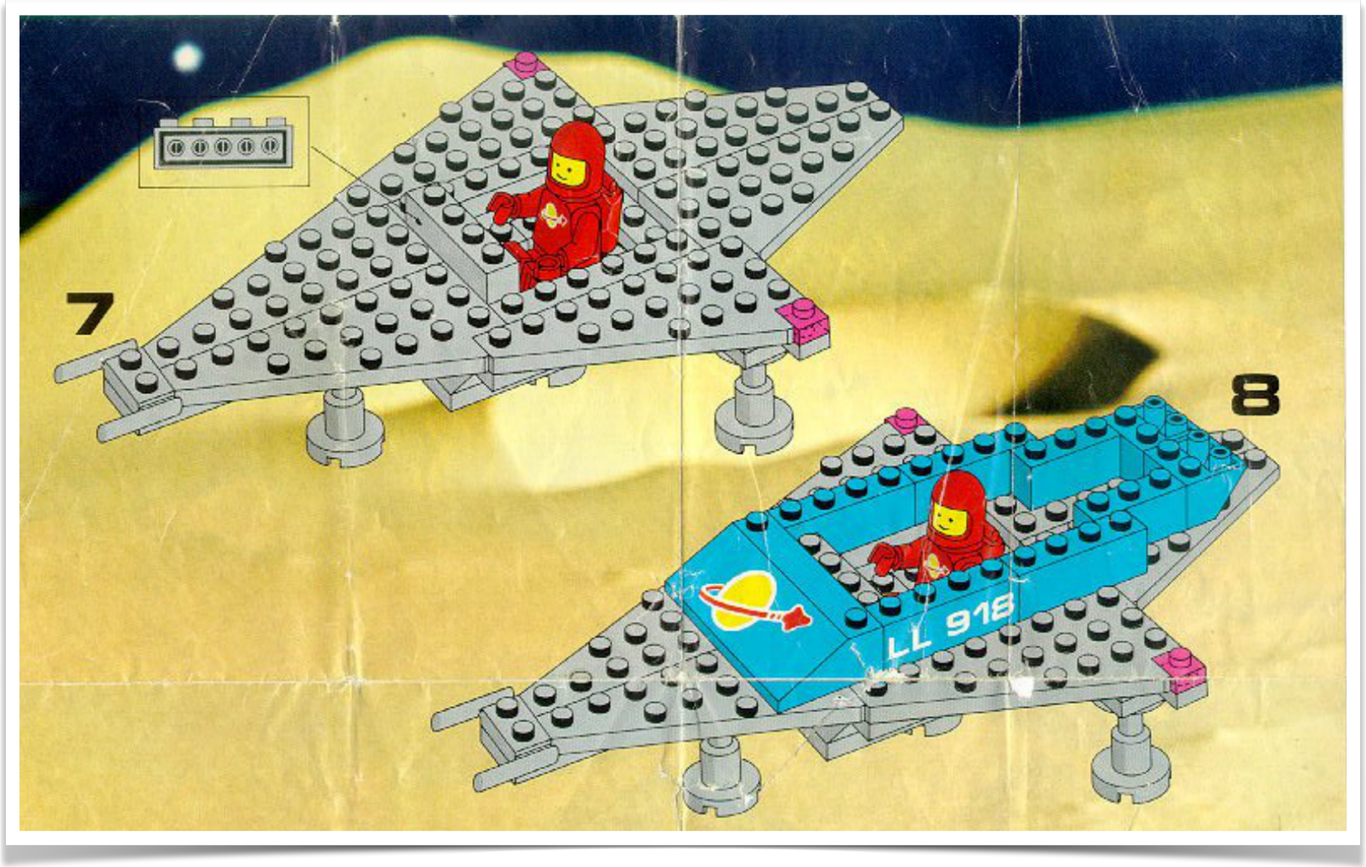
- We now write scenarios close to the business language
- SMT & Datomic functional storage - we can “run" a scenario - yet go to any step and make db assertions, run db queries, SMT queries, check transitions, etc.

- A significant improvement over previous unit tests - less ad-hoc.

- A significant improvement over previous unit tests - less ad-hoc.

- For complex scenario, no need to build Android/iOS clients, no need to boot Docker, etc.

- A significant improvement over previous unit tests - less ad-hoc.
- For complex scenario, no need to build Android/iOS clients, no need to boot Docker, etc.
- Can often convert a good customer bug report to a matching test scenario in 15-30 minutes.



Generating Instructions

Stateful PBT

Stateful PBT

- The toy stateful model should be carefully considered - esp. granularity

Stateful PBT

- The toy stateful model should be carefully considered - esp. granularity
- While fuzzing is useful, we want sequences that do match our expectations - managing the search space is important

- Dealers can sell a vehicle to a customer

- Dealers can sell a vehicle to a customer
- Vehicle operators can share keys

- Dealers can sell a vehicle to a customer
- Vehicle operators can share keys
- Vehicle operators can revoke keys

- Dealers can sell a vehicle to a customer
- Vehicle operators can share keys
- Vehicle operators can revoke keys
- System admins can enable a bugfix / feature

```
({:command :transfer, :args [:dealer-2 :customer-10 :asset-10]}
 {:command :share-key, :args [:dealer-1 :customer-7 :asset-11]}
 {:command :transfer, :args [:dealer-2 :customer-3 :asset-18]}
 {:command :revoke-key, :args [:dealer-1 :customer-7 :asset-11]}
 {:command :share-key, :args [:customer-10 :customer-4 :asset-10]}
 {:command :connect, :args [:dealer-1 :asset-11]}
 {:command :share-key, :args [:dealer-1 :customer-3 :asset-6]}
 {:command :share-key, :args [:dealer-1 :customer-5 :asset-9]}
 {:command :connect, :args [:dealer-1 :asset-3]}
 {:command :transfer, :args [:dealer-1 :customer-1 :asset-11]}
 {:command :transfer, :args [:dealer-2 :customer-6 :asset-16]}
 {:command :share-key, :args [:dealer-1 :customer-10 :asset-7]}
 {:command :share-key, :args [:dealer-2 :customer-8 :asset-15]}
 {:command :share-key, :args [:dealer-1 :customer-5 :asset-6]}
 {:command :transfer, :args [:dealer-2 :customer-4 :asset-6]}
 {:command :connect, :args [:dealer-1 :asset-8]}
 {:command :share-key, :args [:customer-3 :customer-6 :asset-18]}
 {:command :revoke-key, :args [:dealer-2 :customer-8 :asset-15]}
 {:command :share-key, :args [:dealer-1 :customer-8 :asset-20]}
 {:command :connect, :args [:dealer-2 :asset-14]}
 {:command :connect, :args [:dealer-2 :asset-17]}
 {:command :revoke-key, :args [:dealer-1 :customer-8 :asset-20]}
 {:command :share-key, :args [:dealer-2 :customer-8 :asset-17]}
 {:command :revoke-key, :args [:dealer-1 :customer-5 :asset-9]}
 {:command :revoke-key, :args [:customer-3 :customer-6 :asset-18]})
```

```
({:command :transfer, :args [:dealer-2 :customer-10 :asset-10]}
 {:command :share-key, :args [:dealer-1 :customer-7 :asset-11]}
 {:command :transfer, :args [:dealer-2 :customer-3 :asset-18]}
 {:command :revoke-key, :args [:dealer-1 :customer-7 :asset-11]}
 {:command :share-key, :args [:customer-10 :customer-4 :asset-10]}
 {:command :connect, :args [:dealer-1 :asset-11]}
 {:command :share-key, :args [:dealer-1 :customer-3 :asset-6]}
 {:command :share-key, :args [:dealer-1 :customer-5 :asset-9]}
 {:command :connect, :args [:dealer-1 :asset-3]}
 {:command :transfer, :args [:dealer-1 :customer-1 :asset-11]}
 {:command :transfer, :args [:dealer-2 :customer-6 :asset-16]}
 {:command :share-key, :args [:dealer-1 :customer-10 :asset-7]}
 {:command :share-key, :args [:dealer-2 :customer-8 :asset-15]}
 {:command :share-key, :args [:dealer-1 :customer-5 :asset-6]}
 {:command :transfer, :args [:dealer-2 :customer-4 :asset-6]}
 {:command :connect, :args [:dealer-1 :asset-8]}
 {:command :share-key, :args [:customer-3 :customer-6 :asset-18]}
 {:command :revoke-key, :args [:dealer-2 :customer-8 :asset-15]}
 {:command :share-key, :args [:dealer-1 :customer-8 :asset-20]}
 {:command :connect, :args [:dealer-2 :asset-14]}
 {:command :connect, :args [:dealer-2 :asset-17]}
 {:command :revoke-key, :args [:dealer-1 :customer-8 :asset-20]}
 {:command :share-key, :args [:dealer-2 :customer-8 :asset-17]}
 {:command :revoke-key, :args [:dealer-1 :customer-5 :asset-9]}
 {:command :revoke-key, :args [:customer-3 :customer-6 :asset-18]})
```

Initial State

Initial State

- Because the toy model puts everything into the state, we can take any unit test scenario already written and trivially derive an initial state to feed to the command generator

Initial State

- Because the toy model puts everything into the state, we can take any unit test scenario already written and trivially derive an initial state to feed to the command generator
- Jump start the generative test!

test.check

test.check

- We used test.check, an open source Clojure implementation of Quick Check with shrinking

test.check

- We used test.check, an open source Clojure implementation of Quick Check with shrinking
- We implemented a stateful generator with acceptable shrinking in ~130 LOC

Evaluator

Evaluator

- An evaluator for the commands is trivial since all functions are of the same form: $f(\mathrm{S}_0, \ldots) \rightarrow \mathrm{S}_1$

Evaluator

- An evaluator for the commands is trivial since all functions are of the same form: $f(S_0, \ldots) \rightarrow S_1$
- Evaluate the commands and record the final hash. Rerun the transactions (not the commands) from fresh toy state and verify hash is equal.

Stateful PBT Generator

Stateful PBT Generator

- Can create a “gold file”, a file of SMT hash to series of commands to generate that hash. Checked in CI when source code is pushed.

Stateful PBT Generator

- Can create a “gold file”, a file of SMT hash to series of commands to generate that hash. Checked in CI when source code is pushed.
- Write property check to verify that a feature (or fix) can be enabled/disabled and always arrive at same hash

```
(defspec app-hash-playback-toggle 5
  (testing "App-hash test in the presence of feature toggling"
    (let [state      (base-scenario)
          init-state (-> state
                         big-step/state->gen-state
                         (update :features dissoc :abci.feature/short-retail-key-share))]
      (prop/for-all [commands (scen-gen/commands models/abci init-state 20 30)]
        (let [state1 (big-step/run state commands)
              state2 (ledger/playback (state/create tu/*conn*) (:txes state1))]
          (= (some-> state1 :smt :trie-root data-crypto/bytes->hex-str)
             (some-> state2 :smt :trie-root data-crypto/bytes->hex-str))))))
```

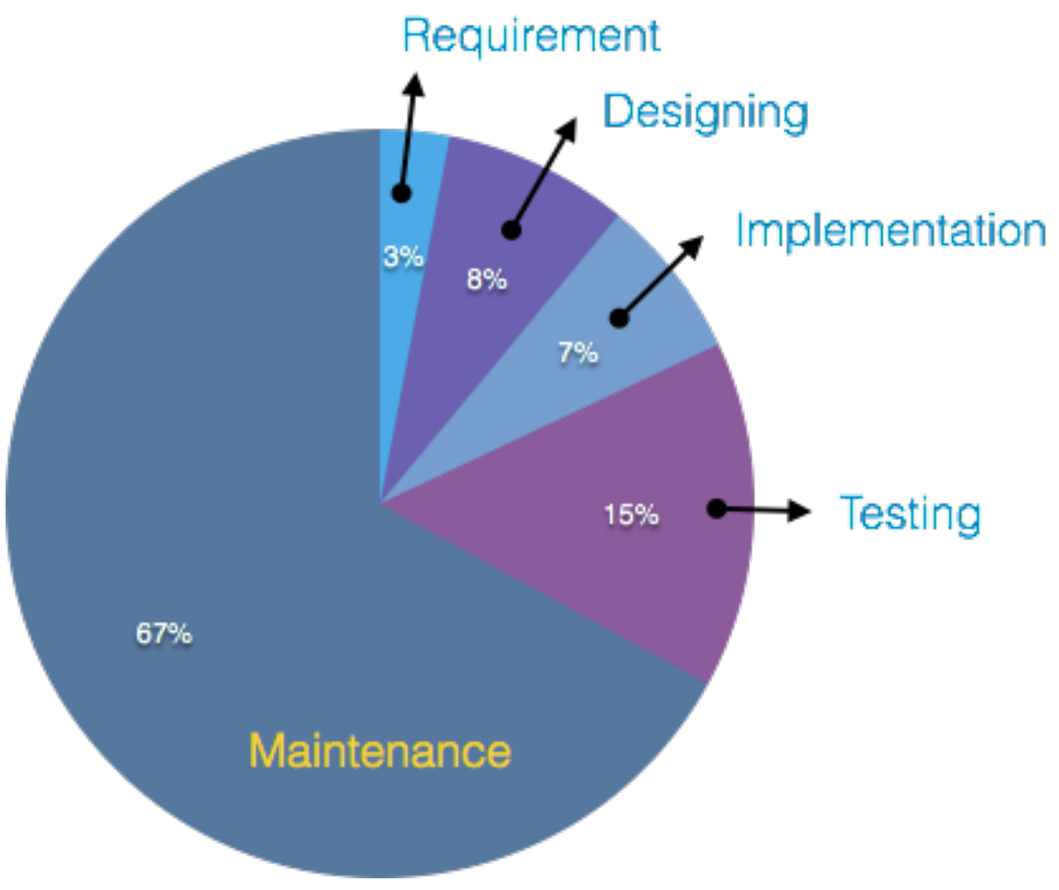
Takeaways

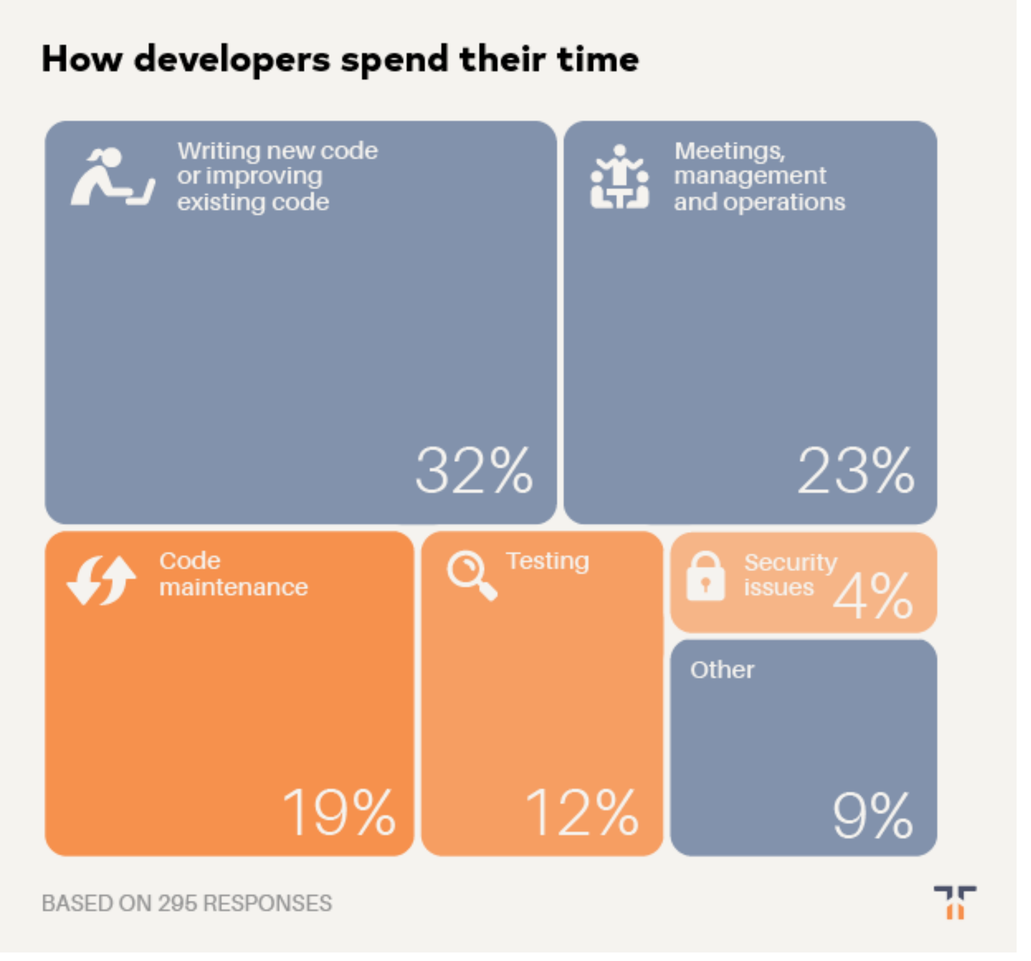
Takeaways

- Creating a toy model allowed us to write better unit tests - customer bug reports are easily captured in domain

Takeaways

- Creating a toy model allowed us to write better unit tests - customer bug reports are easily captured in domain
- Stateful PBT checks increased confidence about deploying fixes / new features to blockchain where backwards compatibility is a hard requirement


Requirement
Designing
Implementation
Testing
Maintenance
3%
8%
7%
15%
67%


How developers spend their time
Writing new code or improving existing code
32%
Meetings, management and operations
23%
Code maintenance
19%
Testing
12%
Security issues
4%
Other
9%
BASED ON 295 RESPONSES

Oct. 24, 1961 G. K. CHRISTIANSEN 3,005,282

TOY BUILDING BRICK

Filed July 28, 1958 2 Sheets-Sheet 1

FIG. 1.

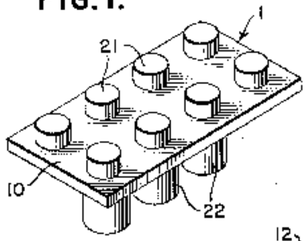


FIG. 2.

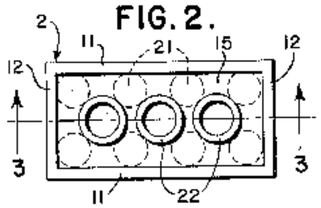


FIG. 3.

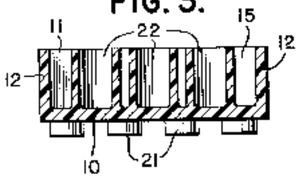


FIG. 4.

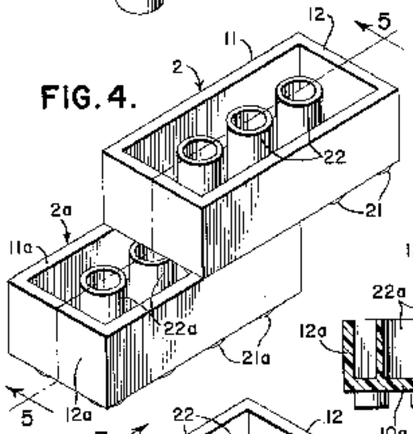


FIG. 5.

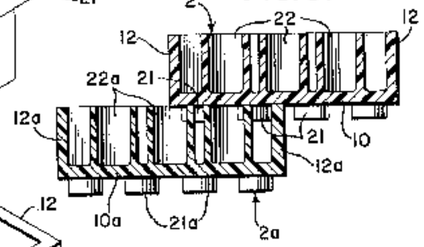
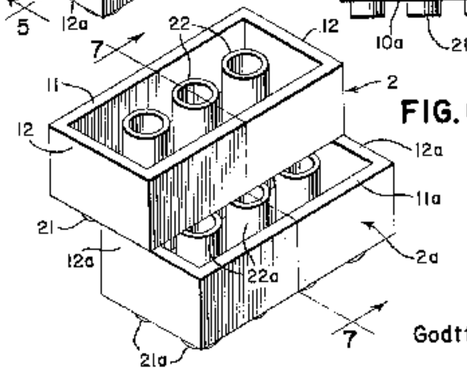


FIG. 6.



INVENTOR

Godtfred Kirk Christiansen

BY Stevens, Davis, Miller & Mosher
ATTORNEYS

Questions?