Squashing Bugs and Empowering Programmers with User-Centered Programming Language Design

MICHAEL COBLENZ

BUT FIRST: FEEDBACK

IT'S TOO HARD TO WRITE SAFE SOFTWARE

Home » News & Events » Blogs » Tech@FTC » FTC warns companies to remediate Log4j security vulnerability

FTC warns companies to remediate Log4j security vulnerability

By: This blog is a collaboration between CTO and DP
2022 9:19AM

22 September 2021

Integer Overflow resulting in OOB Access in CoreGraphics as used by iMessage (ForcedEntry)

#Binary

Original Post:
ero-Click iPhone Exploit

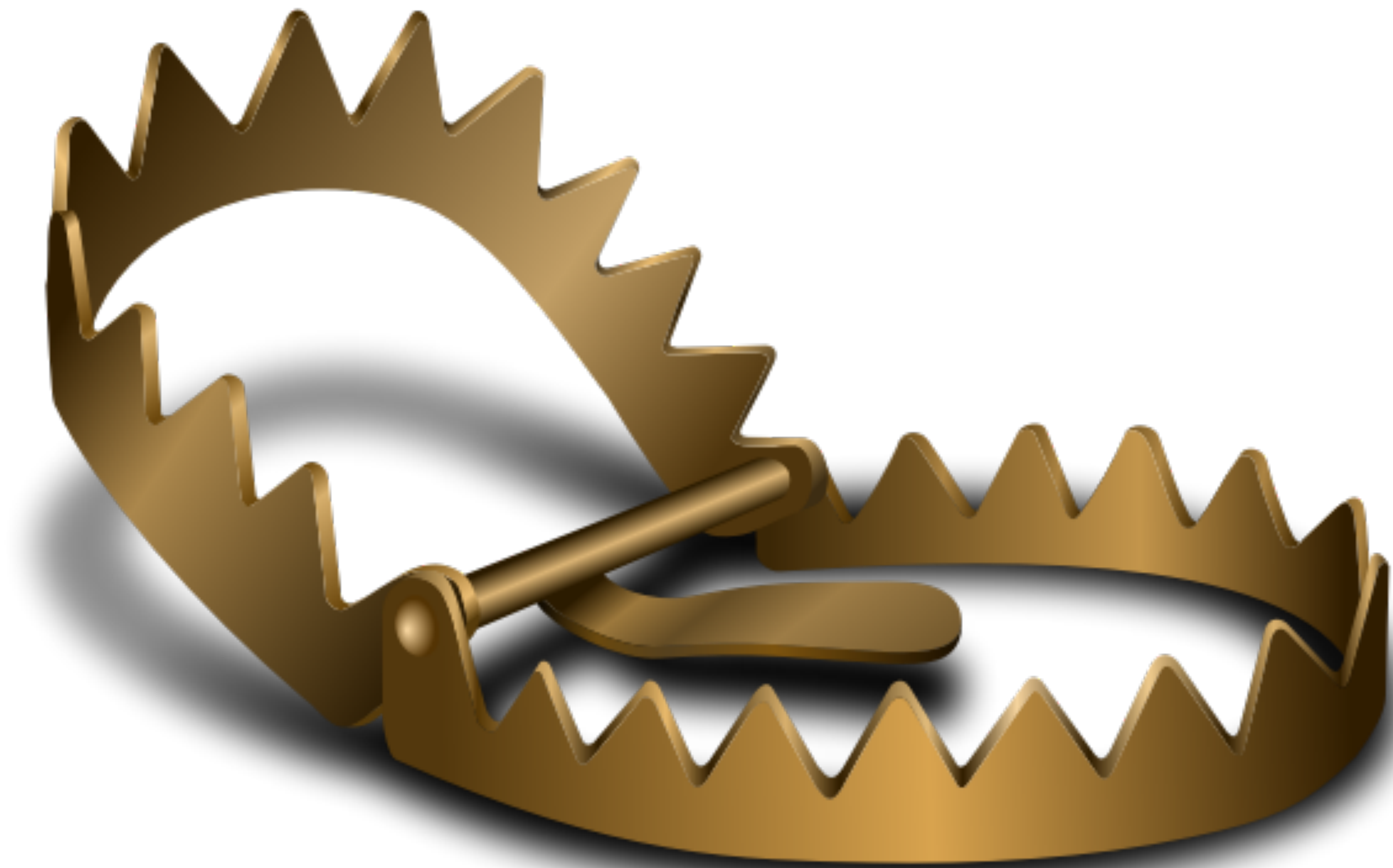
TECH • AMAZON

Amazon says software problem was at root of huge Internet outage this week

BY **SPENCER SOPER**, **JACK GILLUM**, AND **BLOOMBERG**
December 10, 2021 8:04 PM EST

…"Basically, a bad piece of code was executed automatically and it caused a snowball effect," Forrester analyst Brent Ellis said.

PROGRAMMING LANGUAGES LAY TRAPS



TRAPS PROGRAMMING LANGUAGES LAY

- In an object-oriented language: call `foo() on x.`
  - Not if `x`'s type doesn't implement `foo()`
  - Not if `x` is `null`
  - Not if `x` is a reference to freed memory
- Call `person.setAddress(a)`?
  - Is `person` used as a key in a hash table?

NO FREE LUNCH

- Goal: make serious errors less common
  - e.g. security
- Also want to keep the cost low
- Let's talk about an everyday safety problem…

Option A

- Easy to dispense
- Easy to replace roll
- …oops, the roll fell off.

https://www.hansgrohe-usa.com/articledetail-logis-toilet-paper-holder-40526000

Option B

- Can't steal rolls or paper
- Replacing roll is a privileged operation
- VERY ANNOYING to dispense

https://accessories.toiletpartitions.com/product/bradley-5402-dual-roll-toilet-paper-dispenser/

IMPROVING PRODUCTIVITY THROUGH USABILITY

- My goal: enable designers to create safer programming languages with *reasonable, known* user costs
  - Choose *relevant* guarantees
  - Minimize user cost to obtain those guarantees
- Evaluate with users; provide evidence of:
  - Fewer bugs
  - Less time
  - More likely to complete task correctly within time window
  - Less training required

PLIERS: PROGRAMMING LANGUAGE ITERATIVE EVALUATION AND REFINEMENT SYSTEM

- A systematic method for making languages effective for programmers
    - while retaining key safety properties
- Mantra of HCI: **the user is not like me**
    - (similar to the *expert blind spot* in teaching)
- Involve **users** throughout design process
- Soon, I'll show how I leveraged PLIERS in specific designs
- Five steps

PLIERS: PROGRAMMING LANGUAGE ITERATIVE EVALUATION AND REFINEMENT SYSTEM

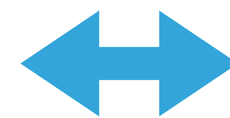
Need-finding

- Interviews
- Corpus studies
- Contextual inquiry

Conception

Formal:

- Core calculus
- Identify properties
- Proof sketches



User-Centered:

- Example programs
- Interpreter/compiler
- Natural programming elicitation

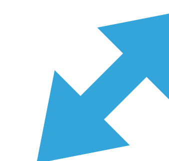
Risk Analysis

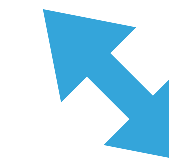
- Cognitive Dimensions of Notations
- Comparison with prior systems
- User research

Refinement

Evaluation:

- Usability studies
- Natural programming
- Performance testing
- Case studies





Theory:

- Core calculus
- Proofs of key properties

Prototype:

- Interpreter/compiler
- Programmer experience

Assessment

- Usability studies
- Randomized controlled trials (RCT)

TODAY

- **Obsidian:** a safer smart contract language [TOPLAS20, OOPSLA20, TOCHI21]

OBSIDIAN

https://en.wikipedia.org/wiki/File:Lipari-Obsidienne_(5).jpg

SMART CONTRACT SECURITY

- Blockchains: for users who do not trust each other
  - High-stakes applications ("smart contracts"). Hard to fix deployed bugs.
- Bugs: > $80 million stolen
  - via programming traps laid by Solidity
- Programming is hard. How can we design languages that are less error-prone?

[1] https://paritytech.io/the-multi-sig-hack-a-postmortem/

RIGOROUS DESIGN PROCESS

- *Need-finding:* studied domain
- *Conception:* identified safety properties, technical approach
- *Risk analysis*: relative to prior languages
- *Refinement*: language design, formal model, case studies
  - a *historical tour*
- *Assessment:* formal proof, RCT comparing Obsidian to Solidity

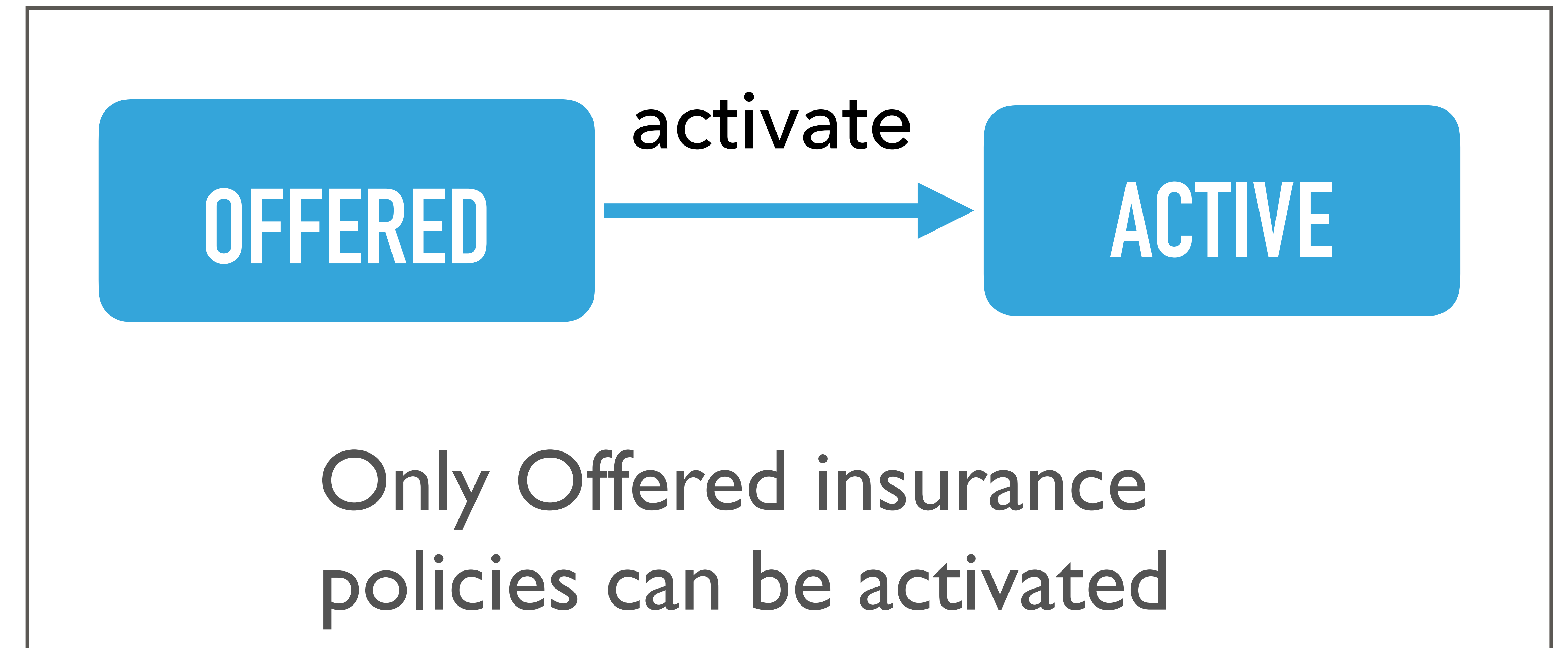
NEED FINDING AND CONCEPTION

- Blockchain applications frequently:
  - Manage important *assets*, such as virtual currencies
    - Some smart contract bugs have involved trapped/forgotten assets [Delmolino et al. 2015]
  - Support different operations depending on state
    - DAO hack exploited re-entrancy – invocation when contract is in an inconsistent state
  - Combining assets with typestate is new, and user studies had not evaluated the usability of typestate or assets

Linearity
[Wadler 1990, Girard 1987]

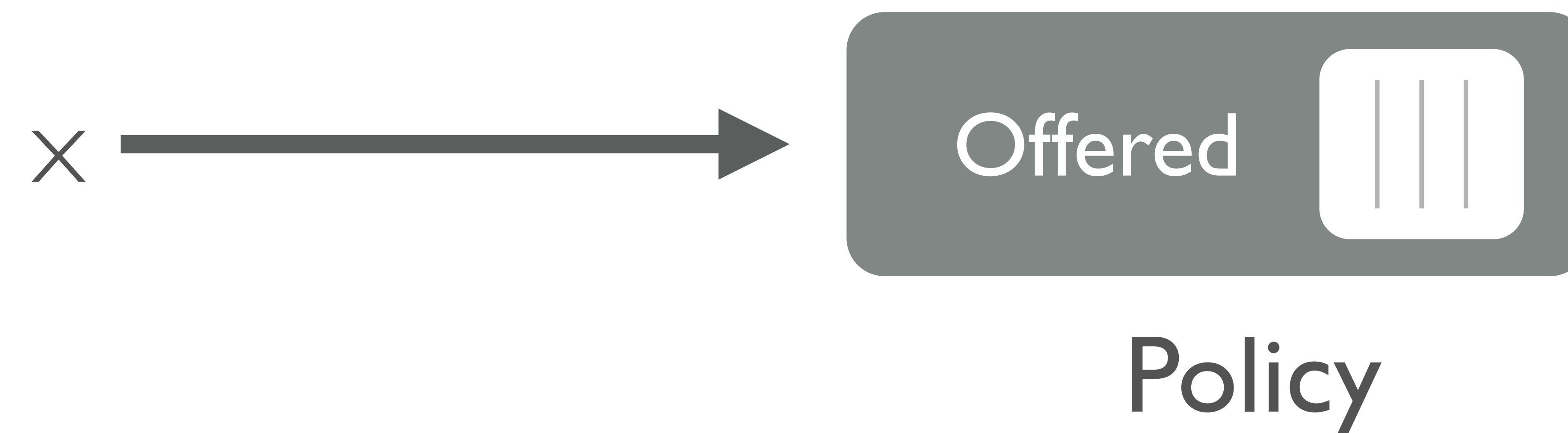
Typestate
[DeLine 2004]

WITHOUT TYPESTATE (e.g., JAVA)

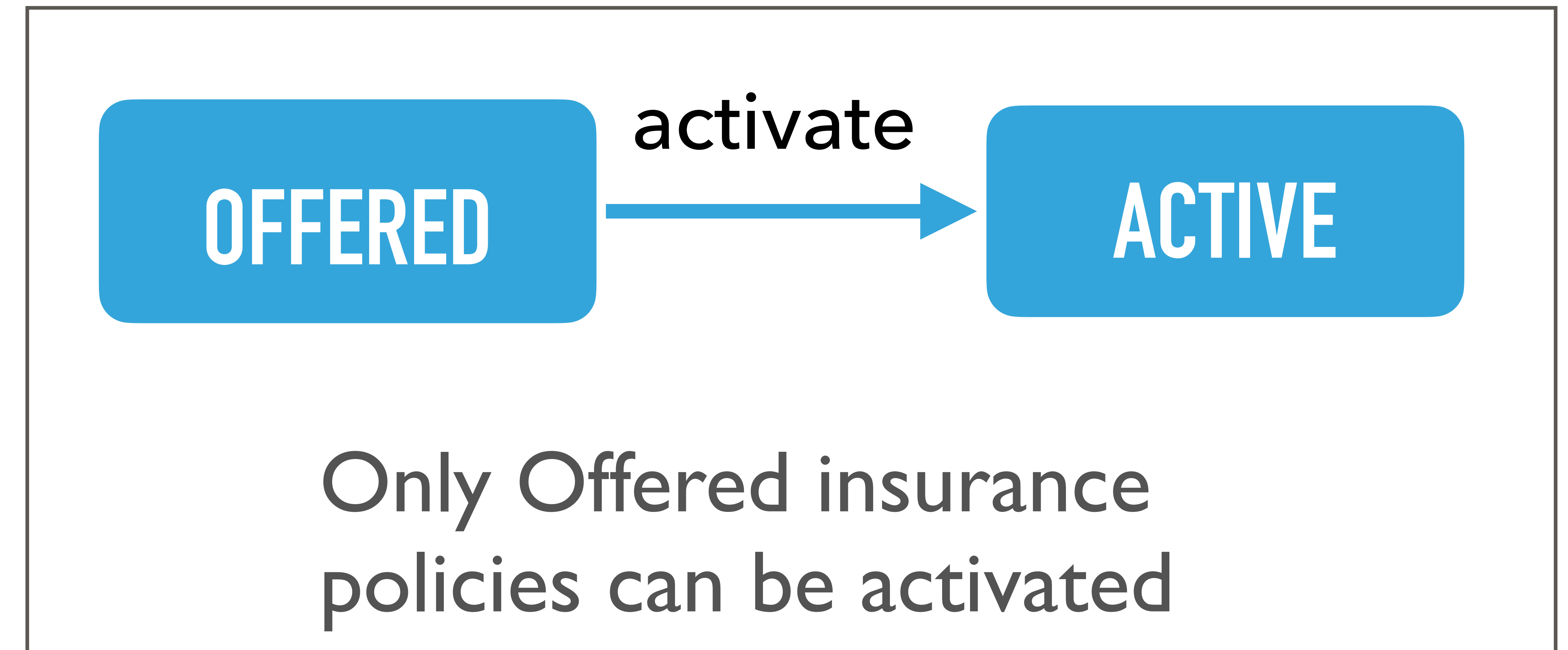


- Type *lacks* state information

```
buy(Policy x) {
  x.activate();
}
```

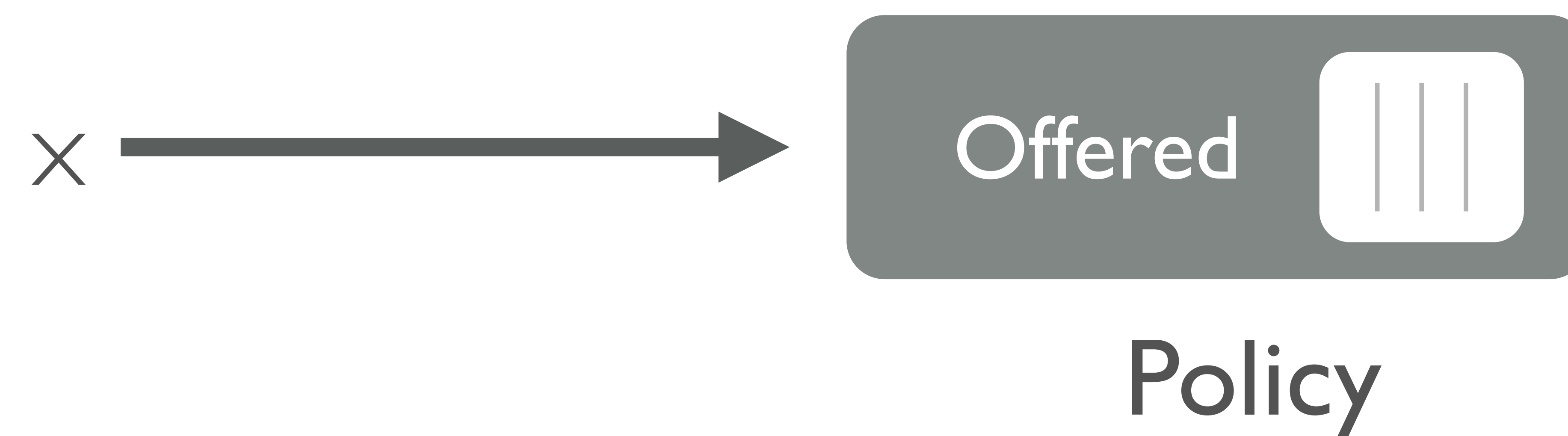


WITH TYPESTATE (OBSIDIAN)



- Type *includes* state information

```
buy(Policy@Offered x) {
  x.activate();
}
```



WITHOUT ASSETS (e.g., JAVA)

```
spend(Token t) {
    transferToken(t, alice);
    transferToken(t, bob);
}
transferToken(Token t, Person p) {
    …
}
```

Compiler says OK!

WITHOUT ASSETS (e.g., JAVA)

```
spend(Token t) {
    transferToken(t, alice);
    transferToken(t, bob);
}
```

Compiler says OK!

Opportunity: references can encode TWO kinds of information:

- Information about referenced object (Token)
- Information about the *relationship* with the referenced object

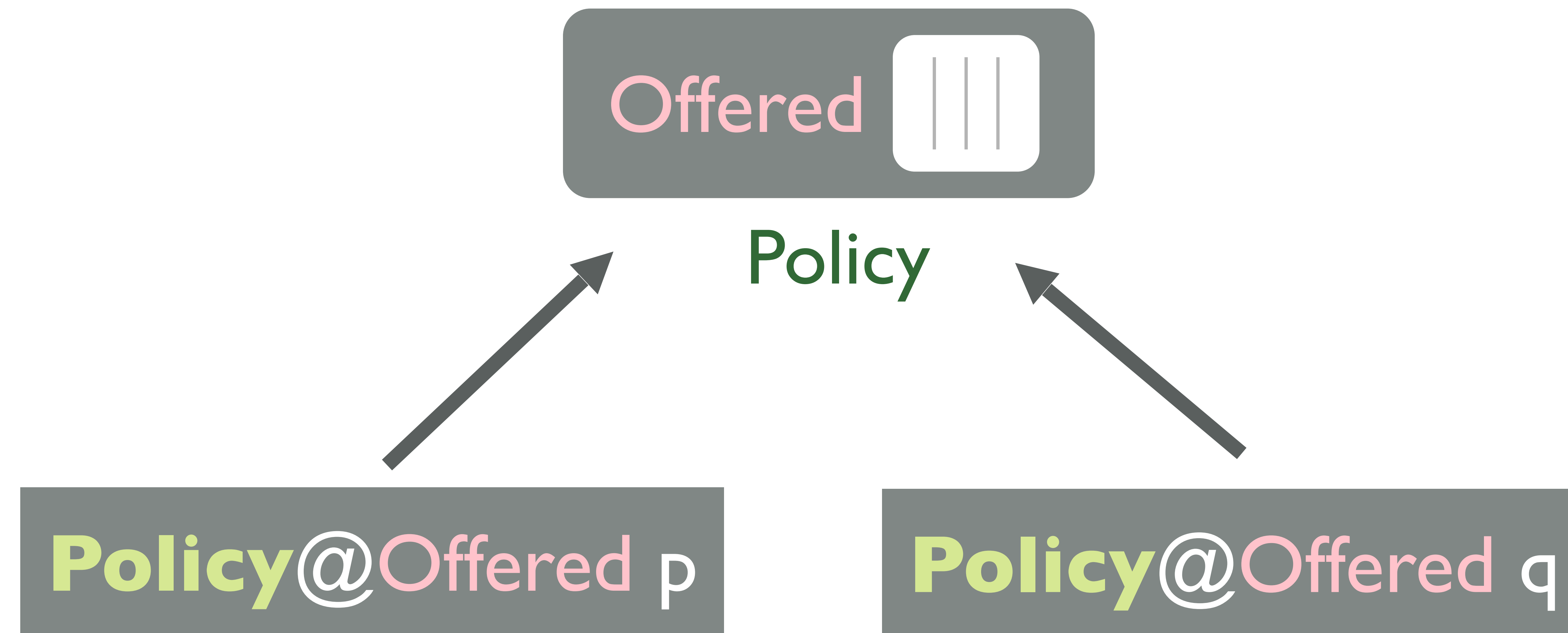
WITH ASSETS (OBSIDIAN)

```
spend(Token@Owned >> Unowned t) {
    transferToken(t, alice);
    transferToken(t, bob);
}

transferToken(Token@Owned >> Unowned t) {
    …
}
```

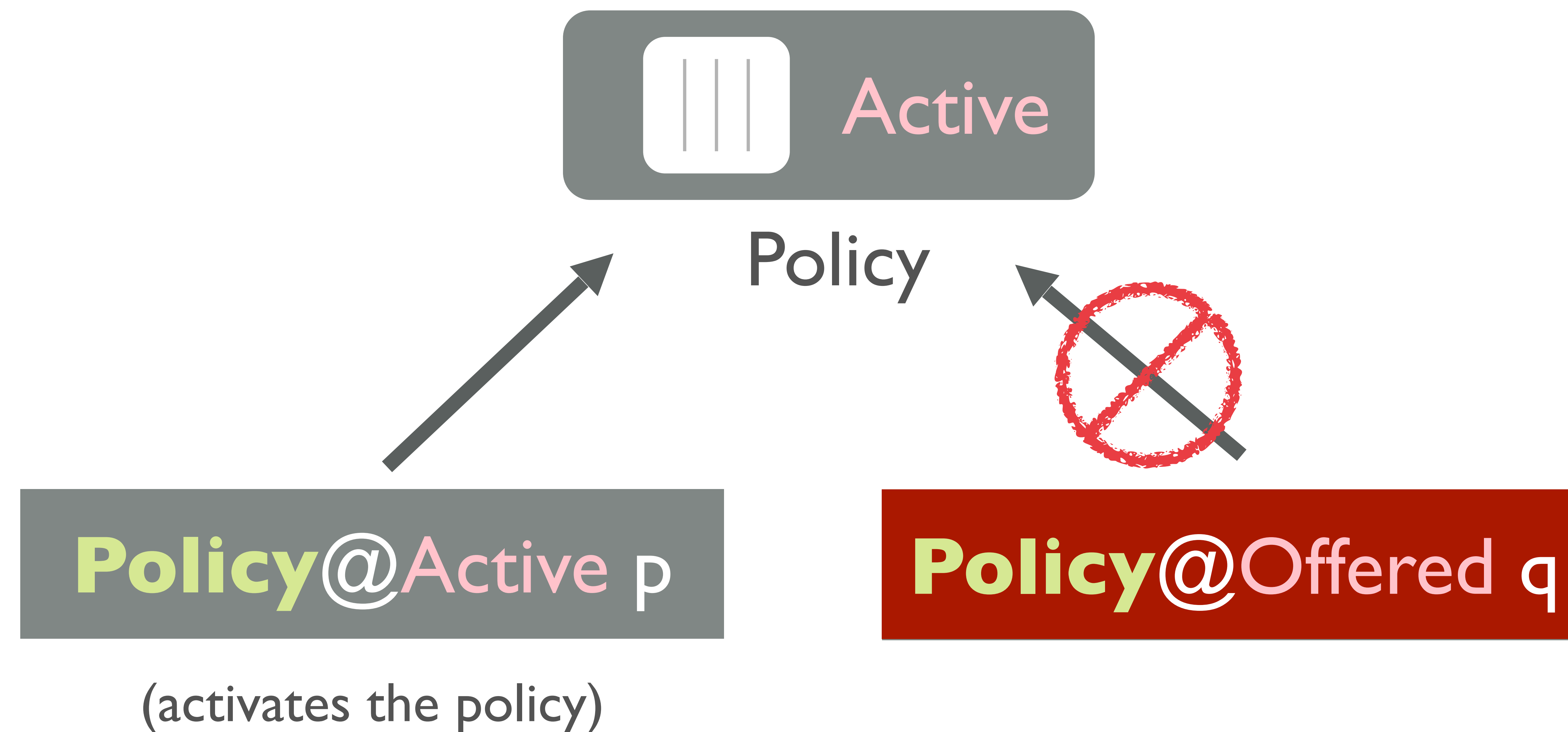
Compiler says ERROR!

TECHNICAL CHALLENGE: TYPESTATE AND ALIASING



Two fields think they know what state the policy is in…

TECHNICAL CHALLENGE: TYPESTATE AND ALIASING



If there is a typestate-specifying reference, then no other reference can change typestate.

OBSIDIAN BY EXAMPLE: PARAMETRIC INSURANCE

- Real client: World Bank
- Automated insurance policies via blockchain
  - Great in countries without trusted insurance companies
- Insure crops against *July rain < 2 in.*
- Insurance pays out if weather meets criteria

OBSIDIAN

```
contract Policy {
  state Offered {
    int cost;
    int expirationTime;
  }

  state Active;
  state Expired;


  Policy@Offered(int c, int expiration) {
    ->Offered(cost = c, expirationTime = expiration);
  }

  transaction activate(Policy@Offered >> Active this) {
    ->Active;
  }
```

cost, expirationTime always in scope; must be repeatedly reinitialized

Obsidian is shorter overall

SOLIDITY

```
contract Policy {
  enum States {Offered, Active, Expired}
  States public currentState;
  uint public cost;
  uint public expirationTime;

  constructor (uint _cost, uint _expirationTime) public {
    cost = _cost;
    expirationTime = _expirationTime;
    currentState = States.Offered;
  }

  function activate() public {
    require(currentState == States.Offered,
        "Can't␣activate␣Policy␣not␣in␣Offered␣state.");
    currentState = States.Active;
    cost = 0;
    expirationTime = 0;
  }
```

Explicit, verbose state checks (easy to omit)

ITERATING ON THE DESIGN

PLIERS: PROGRAMMING LANGUAGE ITERATIVE EVALUATION AND REFINEMENT SYSTEM

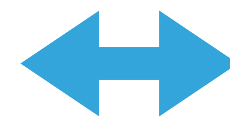
Need-finding

- Interviews
- Corpus studies
- Contextual inquiry

Conception

Formal:

- Core calculus
- Identify properties
- Proof sketches



User-Centered:

- Example programs
- Interpreter/compiler
- Natural programming elicitation

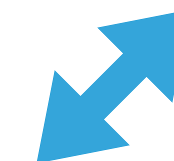
Risk Analysis

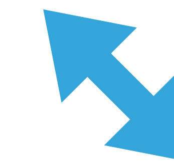
- Cognitive Dimensions of Notations
- Comparison with prior systems
- User research

Refinement

Evaluation:

- Usability studies
- Natural programming
- Performance testing
- Case studies





Theory:

- Core calculus
- Proofs of key properties

Prototype:

- Interpreter/compiler
- Programmer experience

Assessment

- Usability studies
- Randomized controlled trials (RCT)

FORMATIVE STUDY

- How can we evaluate the design proposal without implementing the whole language?

WIZARD OF OZ

- Traditional Wizard of Oz: experimenter remote-controls a prototype (to avoid implementing the “real” system)
- Users had a text editor; an experimenter simulated error messages

MOCK ENVIRONMENTS

- Goal: avoid teaching participants a whole new language
  - Teaching is high-cost
  - Effects of independent variable are conflated with other effects of new language
- Approach: back-port language changes to a familiar language

INITIAL PLAN (BASED ON PRIOR WORK)

- Example Annotation: **`owned`** `Bond@Sold`

  Permission Typestate

- Benefit: orthogonal permission and typestate specifications

ORTHOGONAL OWNERSHIP AND TYPESTATE (PRESENTED IN JAVA)

```
@Owned Prescription.FULL
```

- Lab study (Java annotations)
  - Asked participants to fix a typestate- and ownership-related bug
  - Allowing duplicate prescription refills
- Result: users had serious difficulties
  - "I haven't seen…types that complex in an actual language…enforced at compile time."
  - Participants thought about ownership *dynamically*, not statically
  - Expanded tutorial and practice did not seem to help
- Reflected a collection of different problems

ASSIGNMENT AND PARAMETER-PASSING

- Initial design: @Owned means acquires ownership

```
transaction deposit (Token@Owned t) {
  …
}
```

- User study results (N=6): this is confusing
  - *"when I [annotate this constructor type* **@Owned***], I'm not sure if I'm making a variable owned or I'm transferring ownership."*

- Revised design: make change in ownership explicit

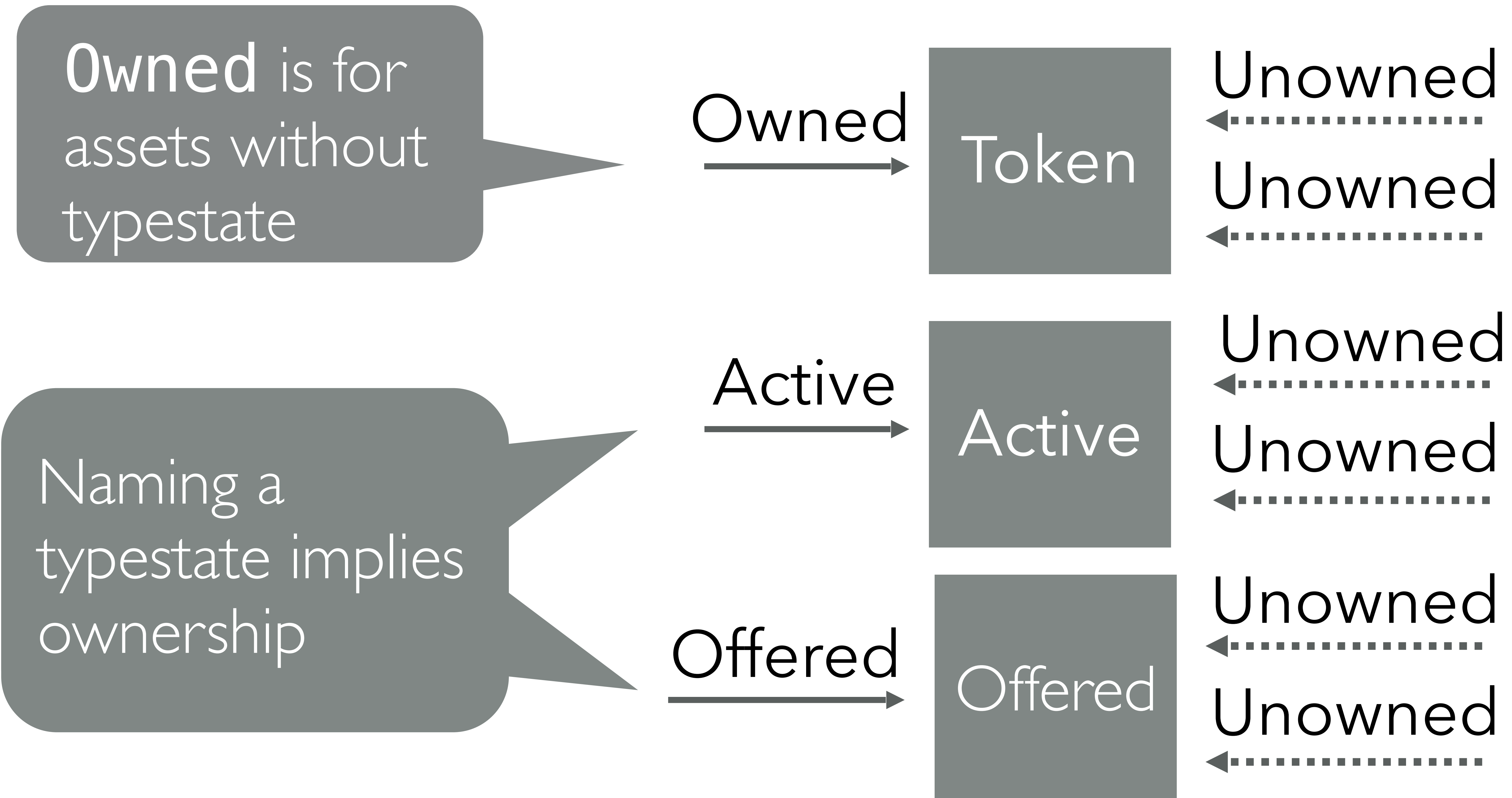
```
transaction deposit (Token@Owned >> Unowned t) {
  …
}
```

TYPE DECLARATIONS

- Initial design (from prior work): always specify typestate when declaring variable
- User study results (N=5): Too confusing!
  - ```
    Policy@Offered p = new Policy();
    p.activate();
    ```
- Revised design
  - Specify initial typestate, and any transition, in method signature (modularity)
  - No typestate on variables; support static typestate assertions

```
[s1 @ Offered];
```
```

REVISED DESIGN COMBINES TYPESTATE AND OWNERSHIP



~~`owned Policy@Offered`~~

Example types:

- `Token@Owned`
- `Policy@Offered`

- Simplifies design
- Avoids reasoning about ownership separate from typestate
- Aligns notation with abstraction

SILICA: A SOLID FOUNDATION FOR OBSIDIAN

Expression typing

$$\Gamma; \Delta \dashv e : T \vdash \Delta'$$

Splitting permissions

$$T_1 \Rrightarrow T_2 / T_3$$

Evaluation

$$\Sigma, e \rightarrow \Sigma', e'$$

Theorems proved

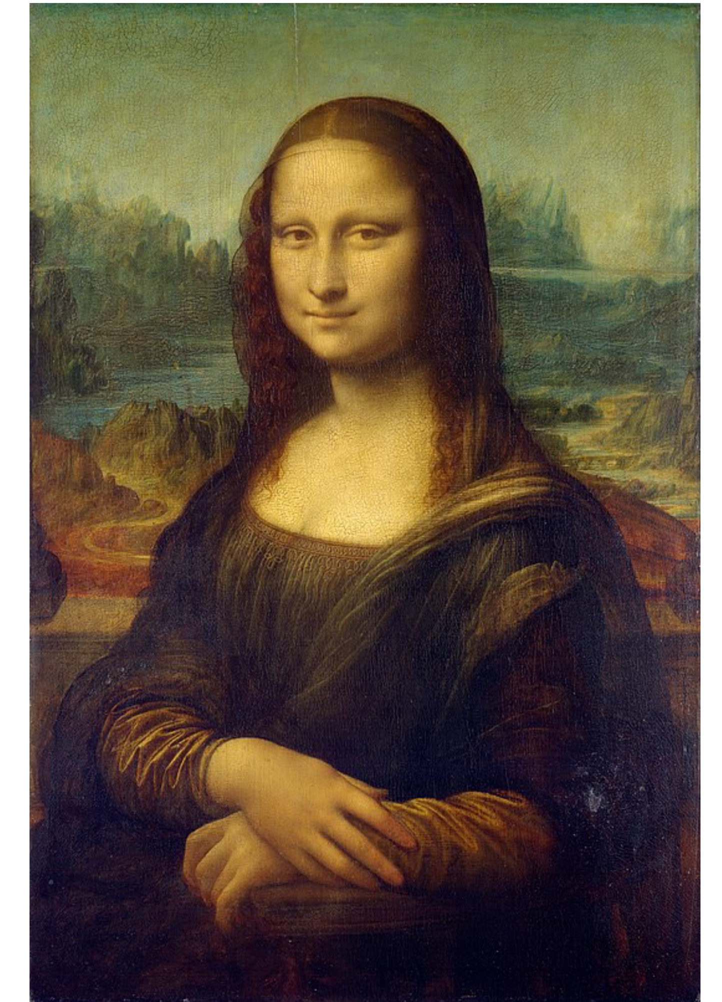
- Progress
- Preservation
- Asset retention

RANDOMIZED CONTROLLED TRIAL

- Is Obsidian *better* than Solidity?
- Recruited 20 Java programmers
- Tutorial: about 90 minutes

AUCTION

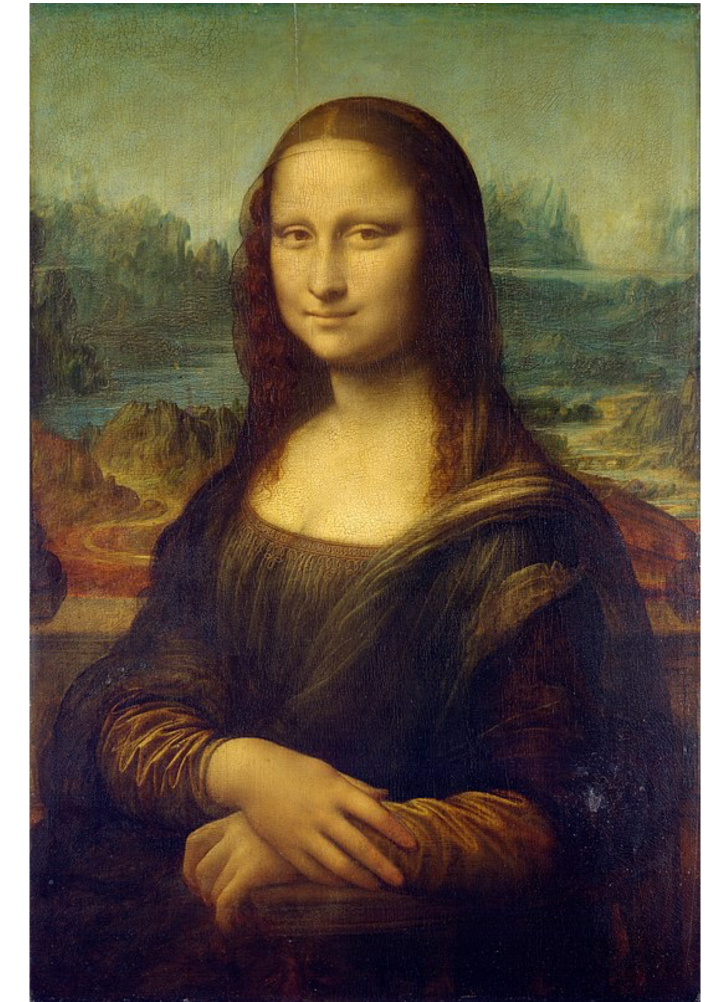
Escrow



An item to be auctioned

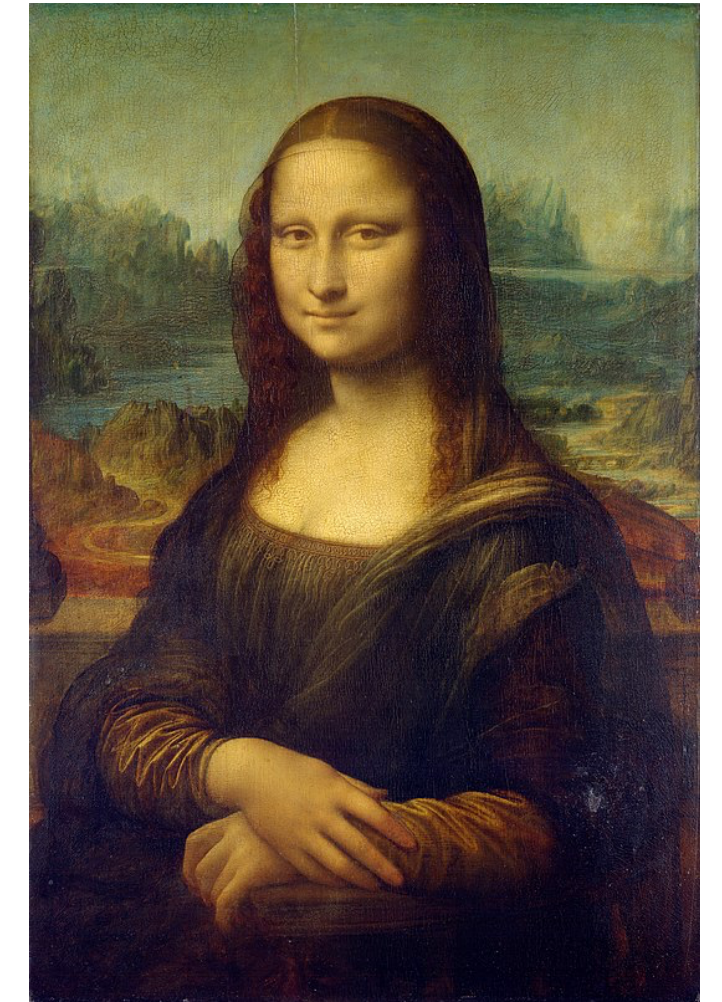
AUCTION

Escrow

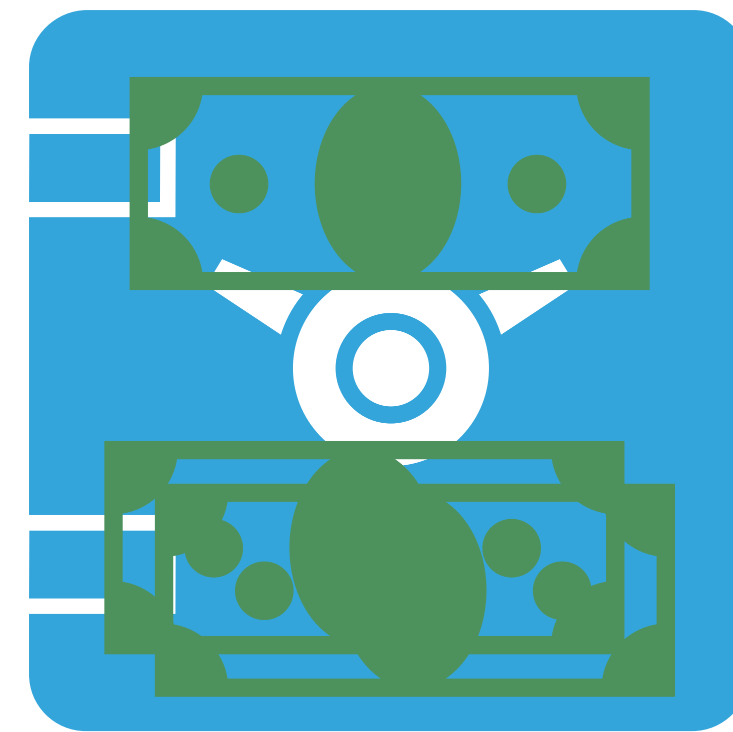


An item to be auctioned

AUCTION

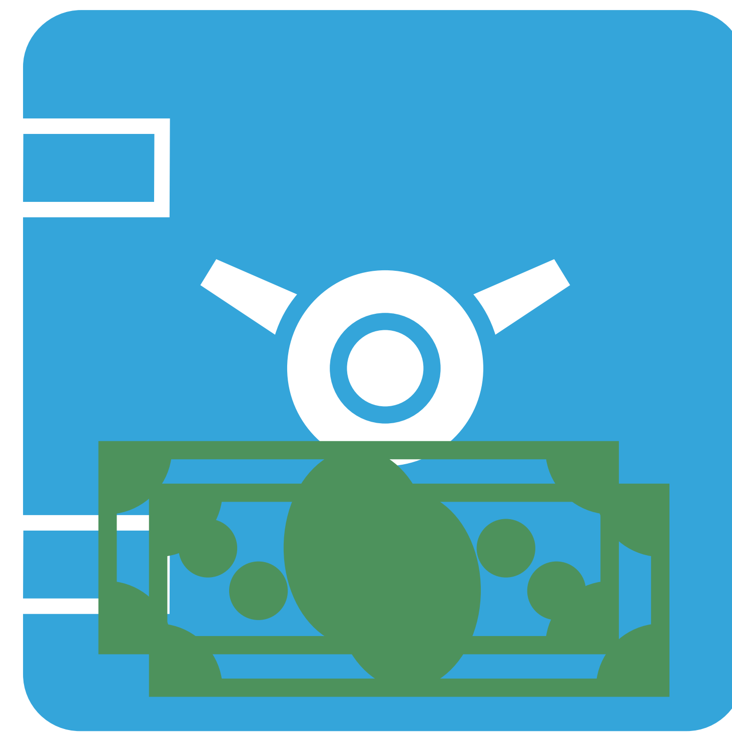


An item to be auctioned

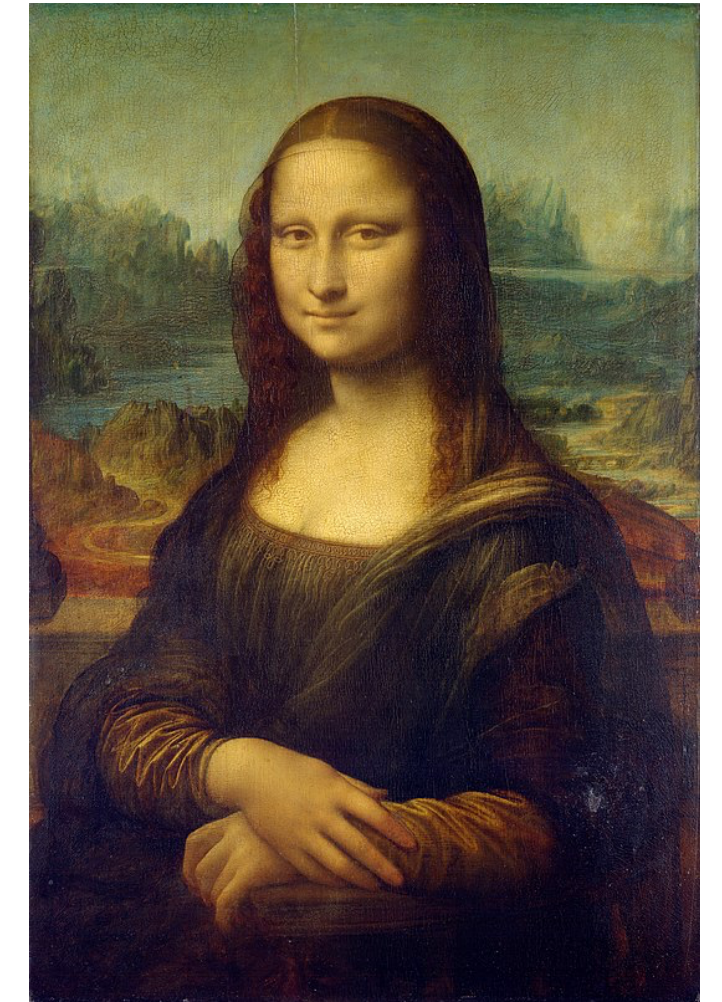


Escrow

AUCTION

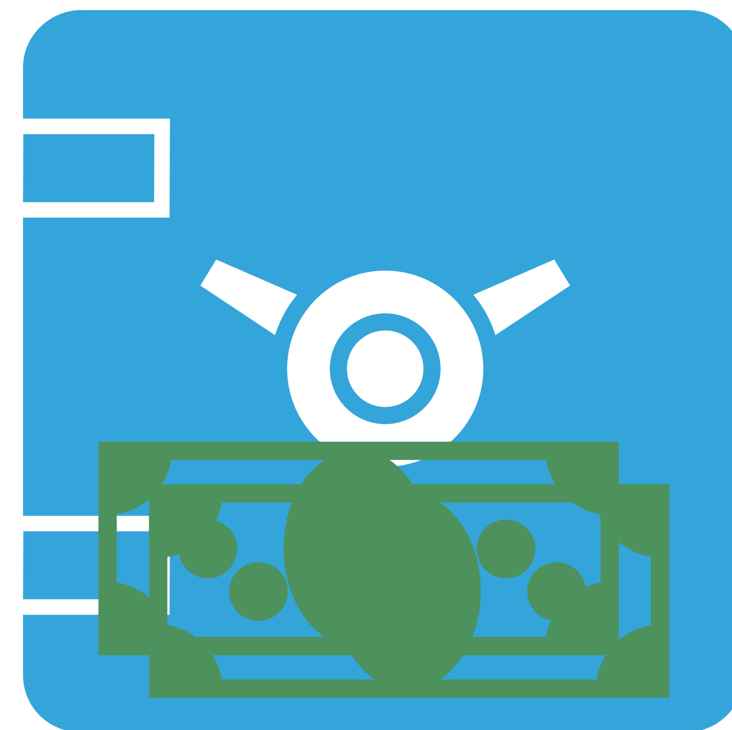


Escrow



An item to be auctioned

AUCTION



Escrow

AUCTION TASK

Condition: Solidity
Time limit: 30 minutes

```
contract Auction {
    address maxBidder; // the bidder who made the highest bid so far
    uint maxBidAmount;
    // 'payable' indicates that we can transfer money to this address
    address payable seller;

    // Allow withdrawing previous bid money for bids that were outbid
    mapping(address => uint) pendingReturns;

    enum State { Open, BidsMade, Closed }
    State state;

    constructor(address payable s) public {
        seller = s;
        state = State.Open;
    }
    …
```

implements *withdrawal pattern*

```
function bid() public payable {
       if (state == State.Open) {
           maxBidder = msg.sender;
           maxBidAmount = msg.value;
           state = State.BidsMade;
       }
       else {
           if (state == State.BidsMade) {
               //if the newBid is > than the current Bid
               if (msg.value > maxBidAmount) {
                   //1. TODO: fill this in.


               }
               else {
                   //2. TODO: return the newBid money to the bidder,
                   //   since the newBid wasn't high enough.
                   //   You may call any other functions as needed.

              }
           }
           else {
               revert ("Can only make a bid on an open auction.");
           }
       }
   }
```

```
function bid() public payable {
        if (state == State.Open) {
            maxBidder = msg.sender;
            maxBidAmount = msg.value;
            state = State.BidsMade;
        }
        else {
            if (state == State.BidsMade) {
                //if the newBid is > than the current Bid
                if (msg.value > maxBidAmount) {
                    //1. TODO: fill this in.
                    maxBidder = msg.sender;
                    maxBidAmount = msg.value;
                }
                else {
                    //2. TODO: return the newBid money to the bidder,
                    //   since the newBid wasn't high enough.
                    //   You may call any other functions as needed.
                    msg.sender.transfer(msg.value);
                }
            }
            else {
                revert ("Can only make a bid on an open auction.");
            }
        }
    }
```

Forgot refund!

Neglects withdrawal pattern

(P48)

RANDOMIZED CONTROLLED TRIAL

- Is Obsidian *better* than Solidity?

	Solidity (N=10)	Obsidian (N=10)
Finished in 30 mins	9	8
Correct answer	2	7

2 of these initially lost assets, but fixed after compiler errors

($p \approx 0.09$)

```
Variable 'maxBid' is an owning reference to an
asset, so it cannot be overwritten.
```

RESULTS: SUMMARY

- Considering all three tasks, mean numbers of successful task completions:
  - Solidity: 0.5 tasks
  - Obsidian: 1.5 tasks

OBSIDIAN CONCLUSIONS

- PLIERS grounded Obsidian's design in user data
- Obsidian enabled Java programmers to use typestate and assets after only 90 minutes of training
- Obsidian safety:
  - No invocations on contracts when they are in inappropriate states (e.g., DAO bug)
  - No asset loss bugs (as seen in lab studies)

CONCLUSION

IMPLICATIONS (FOR PL)

- Maximizing expressiveness may conflict with maximizing usability
  - More complex languages produce more varied kinds of failures
  - Asking users to understand the compiler is usually unreasonable
  - *First* optimize for usability, *then* for expressiveness
- Clarity over brevity
- The first design is rarely best
  - Usability studies should accompany every usability claim

IMPLICATIONS (FOR HCI)

- Expert tools benefit from usability studies too
- Methods require only a little adaptation
  - Think-aloud requires probing
  - Adapting questions to familiar contexts focuses studies, improves feasibility
- Reducing stress may be more important to users than saving time

ADOPTION

- Maybe people adopt languages for reasons *unrelated* to language quality.
  - Therefore, maybe we should be focusing on making good languages more adoptable.
- But: people DO create, modify, and adopt languages regularly.
- My focus: enable designers to create better languages.
  - Then, whatever languages get adopted will be better.