PLAID:

Programming with Typestates and Permissions

Jonathan Aldrich

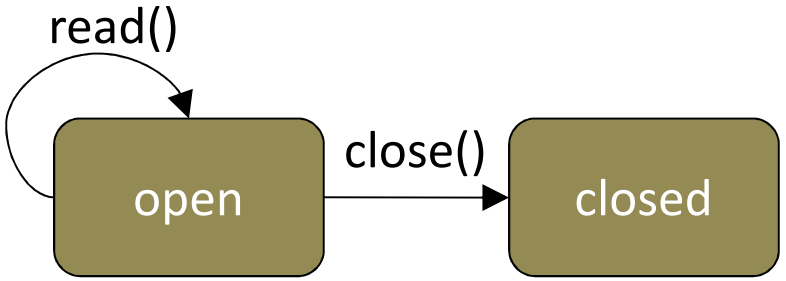
15-214

December 2013

Carnegie Mellon University
School of Computer Science

APIs Define Protocols

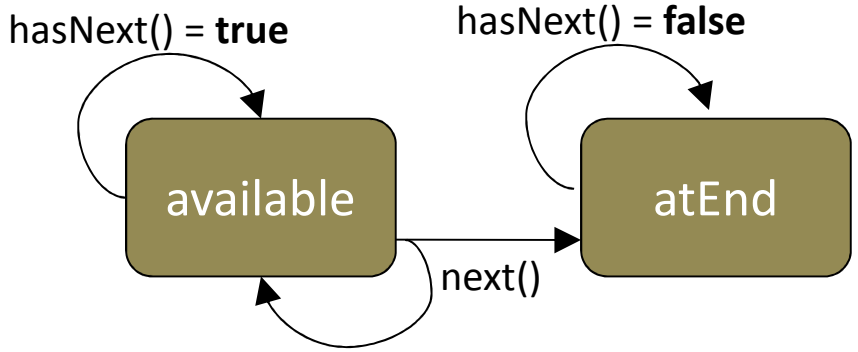
- APIs often define **object protocols**
- Protocols restrict possible orderings of method calls
  - Violations result in error or undefined behavior




```
package java.io;

class FileReader {
    int read() { … }
    …
    /** Closes the stream and releases any system resources associated with it.
    Once the stream has been closed, further read(), ready(), mark(), reset(), or
    skip() invocations will throw an IOException. Closing a previously closed stream
    has no effect. **/
    void close() { … }
}
```

APIs Define Protocols

- Another protocol: Iterator




```
package java.util;

interface Iterator<E> {
    /** Returns true if the iteration has more elements. **/
    boolean hasNext();

    /** Returns the next element in the iteration.  Throws NoSuchElementException if
    the iteration has no more elements. **/
    E next();

    /** Removes from the underlying collection the last element returned by the
    iterator. This method can be called only once per call to next. Throws
    IllegalStateException if the next method has not yet been called, or
    the remove method has already been called after the last call to
    the next method.**/
    void remove();
}
```

Discussion: what does the state machine look like with remove?

Outline and Research Questions

- **How common are protocols?**
- Do protocols cause problems in practice?
- Can we integrate protocols more directly into programming?
- Does such a programming model have benefits?

- Other current and future research

Empirical Study: Protocols in Java

- Object Protocol [Beckman, Kim, & Aldrich, ECOOP 2011]
  - Finite set of abstract states, among which an object will transition
  - Clients must be aware of the current state to use an object correctly
- Question: how commonly are protocols defined & used?
  - Corpus study on 2 million LOC: Java standard library, open source
- Results
  - 7% of all types define object protocols
    - c.f. 2.5% of types define type parameters using Java Generics
  - 13% of all classes act as object protocol clients
  - 25% of these protocols are in classes designed for concurrent use

Outline and Research Questions

- How common are protocols?
- **Do protocols cause problems in practice?**
- Can we integrate protocols more directly into programming?
- Does such a programming model have benefits?

- Other current and future research

Protocols Cause Problems

- Preliminary evidence: help forums
  - 75% of problems in one ASP.NET forum involved temporal constraints [Jaspan 2011]

- Preliminary evidence: security issues
  - Georgiev et al. The most dangerous code in the world: validating SSL certificates in non-browser software. ACM CCS '12.
    - "SSL certificate validation is completely broken in many security-critical applications and libraries…. The root causes of these vulnerabilities are badly designed APIs of SSL implementations."
    - Fix includes not forgetting to verify the hostname (a protocol issue)

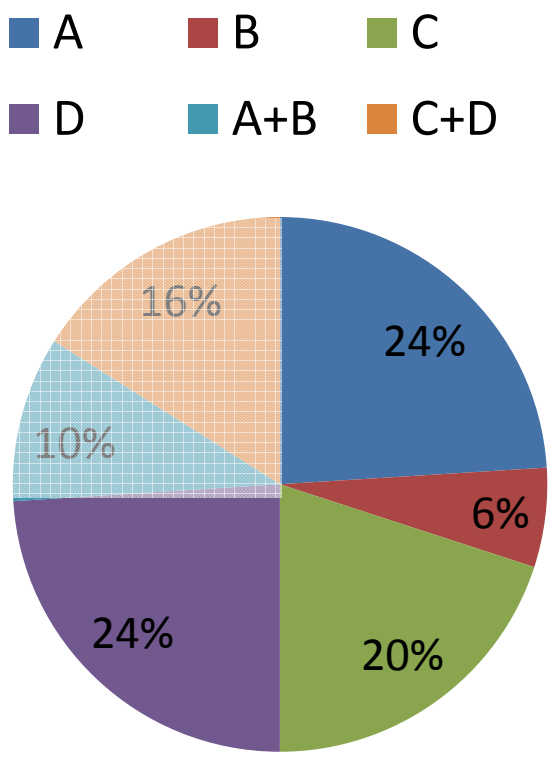
User Study: Programming with Protocols

- User Study [Sunshine & Aldrich, submitted]
  - Selected protocol-related tasks from StackOverflow forums
  - Watched developers perform the tasks in the lab
    - Think-aloud: developers say what they are thinking so we can gain insight into the barriers they encounter
  - Gathered transcripts, timings, and performed open coding of problems
- Results
  - 71% of time spent answering 4 kinds of protocol-related questions

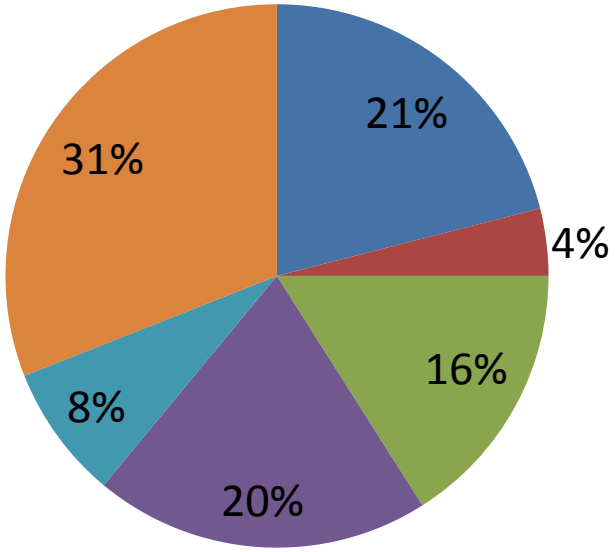
How long does it take to answer each question?

A) What abstract state is the object in?
C) In what state(s) can I do operation Z?

B) What are the capabilities of object in state X?
D) How do I transition from state X to state Y?



% of questions



% of time

PLAID

Outline and Research Questions

- How common are protocols?
- Do protocols cause problems in practice?
- **Can we integrate protocols more directly into programming?**
- Does such a programming model have benefits?

- Other current and future research

Typestate-Oriented Programming

A **new programming paradigm** in which:

programs are made up of dynamically created **objects**,

each object has a **typestate** that is **changeable**

and each typestate has an **interface**, **representation**, and **behavior**.

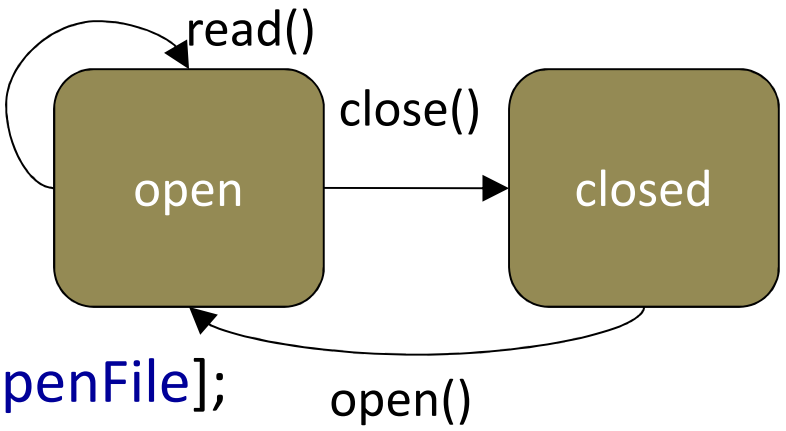
Typestate-oriented Programming is embodied in the language

*Plaid (rhymes with “dad”) is a pattern of Scottish origin, composed of multicolored crosscutting threads

Typestate-Oriented Programming

```
state File {
    val String filename;
}
state ClosedFile = File with {
    method void open() [ClosedFile>>OpenFile];
}
state OpenFile = File with {
    private val CFile fileResource;

    method int read();
    method void close() [OpenFile>>ClosedFile];
}
```

State transition



Different representation

New methods

Implementing Typestate Changes

```
method void open() [ClosedFile>>OpenFile] {
    this <- OpenFile {
        fileResource = fopen(filename);
    }
}
```

Typestate change primitive – like Smalltalk *become*

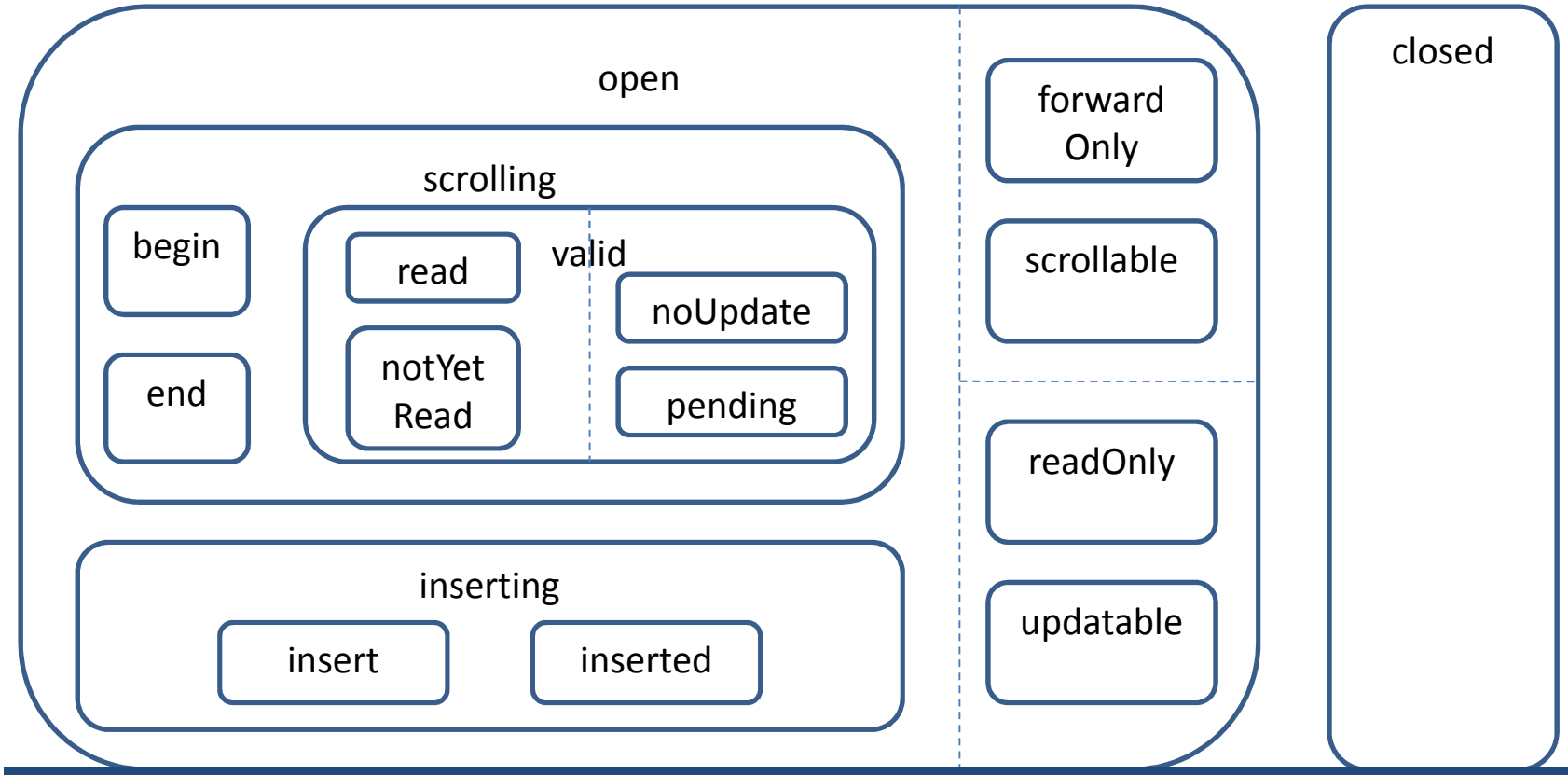
Values must be specified for each new field

:

Why Typestate in the Language?

- The world has state – so should programming languages
  - egg -> caterpillar -> butterfly; sleep -> work -> eat -> play; hungry <-> full
- Language influences thought [Sapir '29, Whorf '56, Boroditsky '09]
  - Language support encourages engineers to **think** about states
    - Better designs, better documentation, more effective reuse
- Improved library specification and verification
  - Typestates define when you can call read()
  - Make constraints that are only implicit today, explicit
- Expressive modeling
  - If a field is not needed, it does not exist
  - Methods can be overridden for each state
- Simpler reasoning
  - Without state: fileResource non-**null** if File is open, **null** if closed
  - With state: fileResource always non-**null**
    - But only exists in the FileOpen state

Typestate Expressiveness



- Research questions
  - Can we express the structure of real state machines expressed in UML?
  - Can we break protocols into component parts and reuse them?
  - Can we provide better error messages when something goes wrong?
- [Sunshine et al., OOPSLA 2011]

Checking Typestate

```
method void openHelper(ClosedFile>>OpenFile aFile) {
    aFile.open();
}

method int readFromFile(ClosedFile f) {
    openHelper(f);
    val x = computeBase() + f.read();
    f.close();
    return x;
}
```

This method transitions the argument from ClosedFile to OpenFile

Must leave in the ClosedFile state

Use the type of openHelper

f is open so read is OK

Correct postcondition; f is in ClosedFile

Question: How do we know computeBase doesn't affect the file (thorugh an alias)?

Typestate Permissions

- **unique** OpenFile
  - File is open; no aliases exist
  - Default for mutable objects
- **immutable** OpenFile
  - Cannot change the File
    - Cannot close it
    - Cannot write to it, or change the pos
  - Aliases may exist but do not matter
  - Default for immutable objects
- **shared** OpenFile@NotEOF [
  - File is aliased
  - File is currently not at EOF
    - Any function call could change that, due to alia
  - It is forbidden to close the File
    - OpenFile is a *guaranteed* state that must be respected by all operations through all aliases
- **full** – like **shared** but is the exclusive writer
- **pure** – like **shared** but cannot write

pure resource-based programming

pure functional programming

le

ClosedFile

OpenFile

NotEOF

EOF

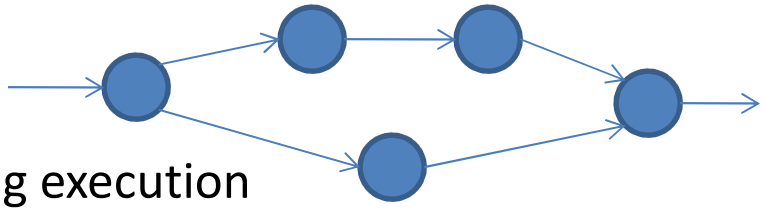
shared OpenFile@OpenFile is (almost) traditional object-oriented programming

Key **innovations** vs. prior work (c.f. Fugue, Boyland, Haskell monads, separation logic, etc.)

Permission Splitting

- Permissions may not be duplicated
  - No aliases to a unique object!
- Splitting that follows permission semantics is allowed, however
  - **unique → full**
  - **unique → shared**
  - **unique → immutable**
  - **shared → shared, shared**
  - **immutable → immutable, immutable**
  - **X → X, pure** *// for any non-**unique** permission X*
- Research challenges
  - Practical permission accounting [POPL '12]
  - Adding dynamic checks / casts [ECOOP '11]

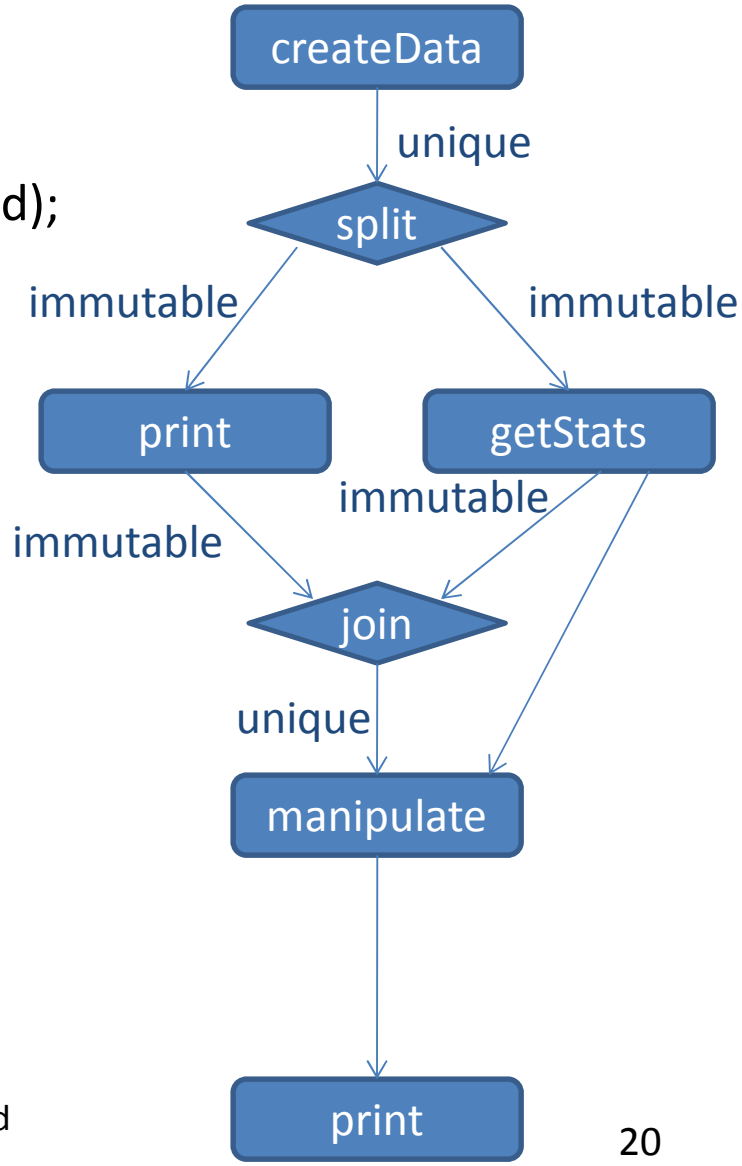
ÆMINIUM: Explicit Dependencies for Concurrency



- Concurrency is a major challenge
  - Avoiding race conditions, understanding execution
- Inspiration: functional programming is "naturally concurrent"
  - Up to data dependencies in program
- Idea: use permissions to construct dataflow graph
  - Easier to track dependencies than all possible concurrent executions
  - Functional programming passes data explicitly to show dependencies
  - For stateful programs, we **pass permissions explicitly** instead
- Consequence: stateful programs can be naturally concurrent
  - Furthermore, we can provide strong reasoning about correctness

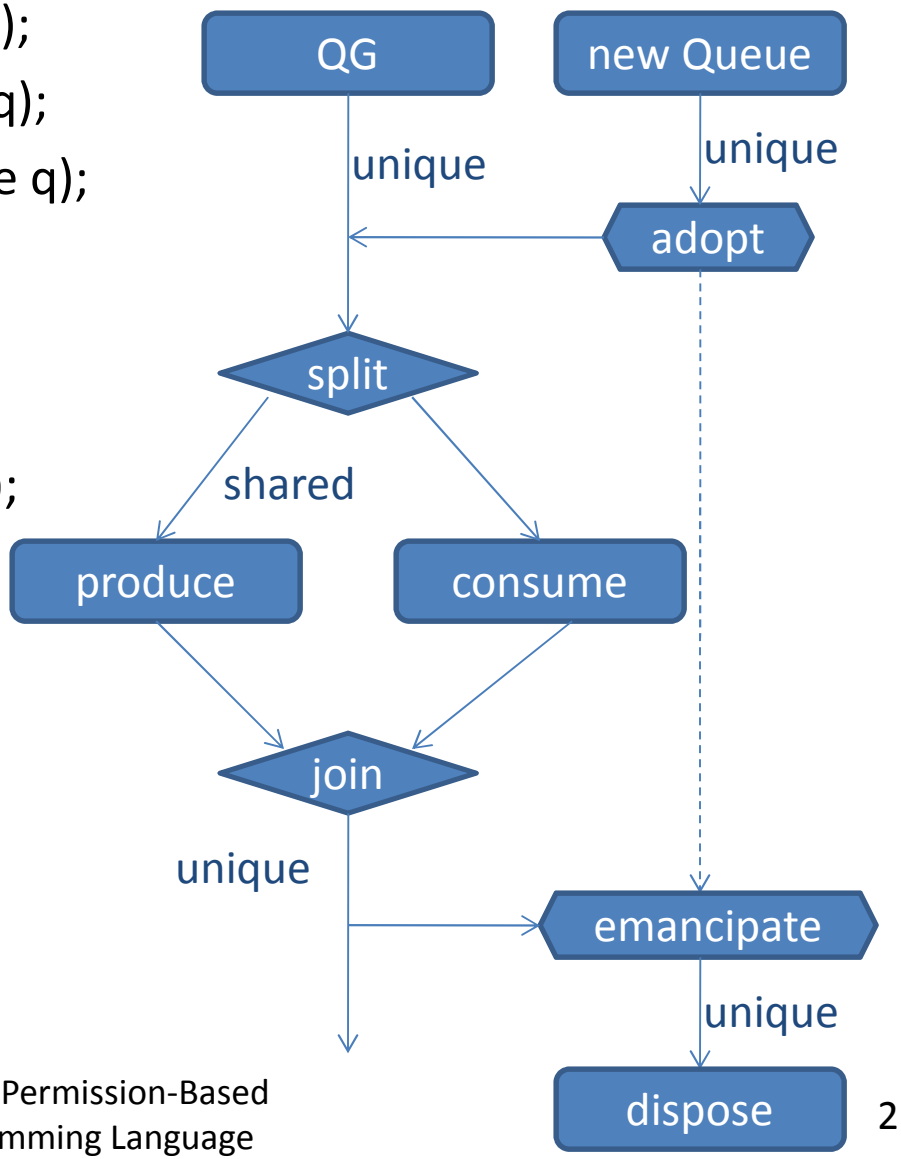
Features: Sharing and Dependencies

```
method unique Data createData();
method void print(immutable Data d);
method unique Stats getStats(immutable Data d);
method void manipulate(unique Data d,
                       immutable Stats s);

val d = createData();
print(d);
val s = getStats(d);
manipulate(d, s);
print(d);
```



Features: Sharing and Dependencies

```
method void produce(‘QG Queue q);
method void consume(‘QG Queue q);
method void dispose(unique Queue q);

group QG;
val QG Queue q = new Queue;
split QG: produce(q) || consume(q);
q.dispose();
```



Outline and Research Questions

- How common are protocols?
- Do protocols cause problems in practice?
- Can we integrate protocols more directly into programming?
- **Does such a programming model have benefits?**

User Experiment: Protocol Documentation Benefits

- Can a state-based programming language help programmers?
  - Multiple possible mechanisms: better documentation, typechecker catches more errors, better run-time error messages
- More focused question
  - Can state-based documentation help programmers complete state-related tasks faster?
- Controlled Laboratory Experiment
  - Similar tasks to the qualitative study described earlier, done in Java
  - Subjects given standard Javadoc, or "Plaiddoc" with state info
    - Hard to test Plaid directly
    - But if we had Plaid, we could generate better state documentation
    - So let's test that
- Results: Plaiddoc participants were dramatically faster
  - Factor of 2x for state-related tasks, p=0.0003
  - No slowdown for non-state-related tasks
  - Also less likely to make errors

PLAID

The Plaid Language

- A holistic, permission-based approach to managing state
  - First-class abstractions for characterizing state change
  - Use permission flow to infer concurrent execution
  - Practical mix of static & dynamic checking
- Opens a new area of research
  - Languages based on changeable states and permissions
- Benefits
  - Productivity enhancements from improved documentation
  - Programs can more faithfully model the target domain
  - Permissions encode design constraints for static/dynamic checking
  - Naturally safe parallel execution model

http://www.plaid-lang.org/