Corpus Studies & Formative Studies for PL Design

Jonathan Aldrich

17-396/17-696/17-960:
Language Design and Prototyping

Overview

- Goal: Learn about a problem you want to solve with your PL
  - Start with a target situation that illustrates the problem
    - A library imposes ordering constraints on calls to its functions/methods
    - Programmers are forced to use types that don't describe exactly what they want
- Questions you can answer:
  - How frequently does the target situation show up?
    - A proxy for importance
  - Find examples of the target situation
    - Can drive language design
  - Characterize/categorize the target situation
  - Does the target situation cause problems?
    - Sometimes can infer from characteristics of the codebase
    - Sometimes want to study programmers

Strategy

- Search open source code, Q&A forums, etc. for patterns
  - Set up rigorous criteria for what you are looking for
    - Connect it to your problem
  - Be creative about sources
    - GitHub super common and easy – lots of data exposed
    - Many alternatives – e.g. we got a lot of mileage out of StackExchange
  - Use automation to collect data at scale
  - Often further manual processing –in PL, the detailed context matters
  - Consider follow-up studies to evaluate actual impact with users

Protocol Programming in

Jonathan Aldrich

17-396/17-696/17-960:
Language Design and Prototyping

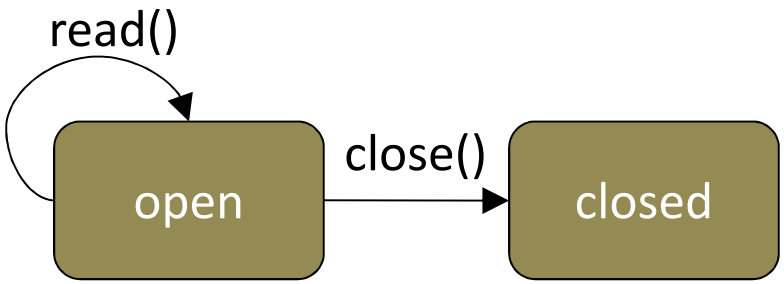
Carnegie Mellon University
School of Computer Science

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() { … }
}
```



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

Object Protocols in the Wild

- How commonly are object protocols defined and used? What are they like?
  - One way to answer: empirical study

- Hypotheses
  - Protocols are defined and used in common libraries and applications with significant frequency
  - Familiar protocols (Iterators, Streams) are most commonly used, but many other kinds of protocols are defined
  - There are a small number of categories of protocols

Protocol Definition

- A type defines an **object protocol** if:
  - the concrete state of objects of that type can be abstracted into a finite number of abstract states,
  - clients must be aware of those states in order to use that type correctly,
  - and object instances dynamically transition among those states

- Aspects of definition:
  - Abstract and finite
  - Observable
  - Important for correct use
  - Run time transitions

- We will also be interested in *type qualifiers*, i.e. states that are set at initialization time
  - Missing third part of definition

Results: Commonality

- At least 7.2% of types define protocols
  - Not a majority—but more common, for example, than generics (2.5%)
  - Our methodology misses some—for example, objects that pass on protocols from their fields account for about 2% more
- At least 13.3% of classes use protocols
- Most commonly used protocols include iterators, streams
  - But also setting the cause of an exception, setting XML attributes
- There are many less common protocols
  - Security, Graphics, Networking, Configuration, Data structures, Parsing, …

Methodology

- Scanning tool
  - Identifies code that tests based on a field, and throws an exception
- Manual examination
  - Test candidates from tool against protocol definition
  - Categorize candidates into group
- Compute usage metrics
  - Automated analysis
- Subjects of study
  - Large, diverse, open-source libraries, applications, and frameworks
  - 1.9 million lines of code
  - Java standard library, Eclipse, Azureus, ant, antlr, freecol, …

Results: Protocol Categories

- 98% of protocols fit into one of 7 categories
  - **Initialization** before use – e.g. init(), open(), connect()
  - **Deactivation** – e.g. close()
  - **Type qualifier** – disables certain methods for the lifetime of an object, e.g. immutable collections are missing mutator methods
  - **Preparation** – e.g. call mark() before reset() on a stream
  - **Boundary** check – e.g. hasNext()
  - **Non-redundancy** – can only call a method once, e.g. setCause()
  - **Mode** – domain-specific modes enable/disable certain operations

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]

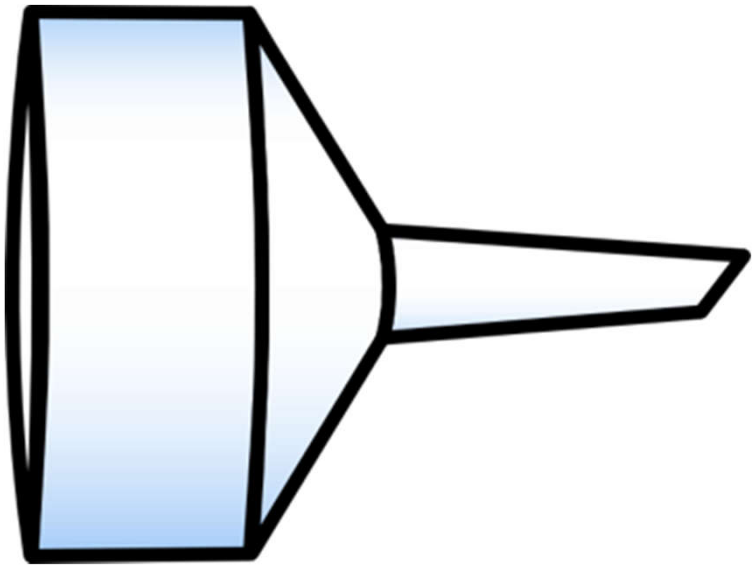
- Preliminary evidence: security issues
  - Georgiev et al. The most dangerous code in the world: validating SSL certificates in non-browser software. ACM CCS '12.
    - "SSL certificate validation is completely broken in many security-critical applications and libraries…. The root causes of these vulnerabilities are badly designed APIs of SSL implementations."
    - Fix includes not forgetting to verify the hostname (a protocol issue)
  - Somorovsky et al. On Breaking SAML: Be Whoever You Want to Be. USENIX Security '12.
    - Again, libraries are insecure if not used correctly

Productivity and Protocols

- How do developers struggle with protocols?
  - What in particular is causing the struggle?
    - Do they understand the protocol concept?
    - Do they understand the error messages?
  - What kinds of protocols cause problems?
  - When struggling what resources do they look to?
  - How do programmers resolve the issue?

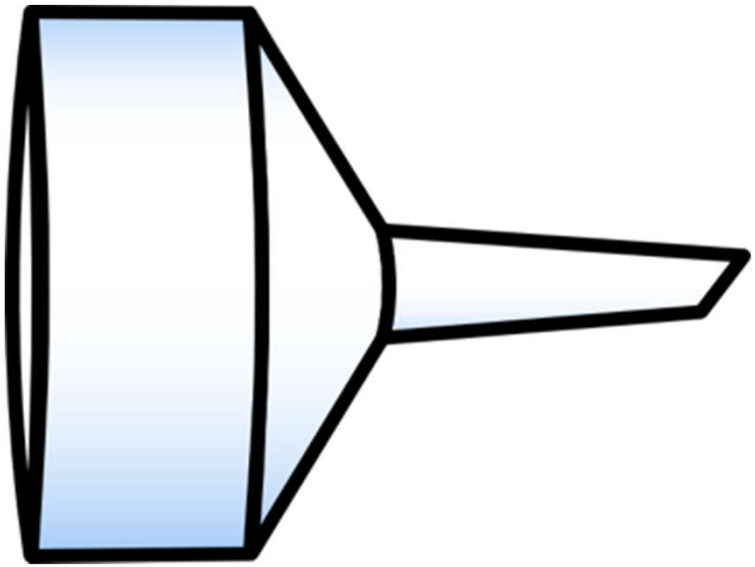
- Knowing how is critical to
  - further study
  - design assurance tools that are usable

Mining forums for protocol challenges



Mining forums for protocol challenges

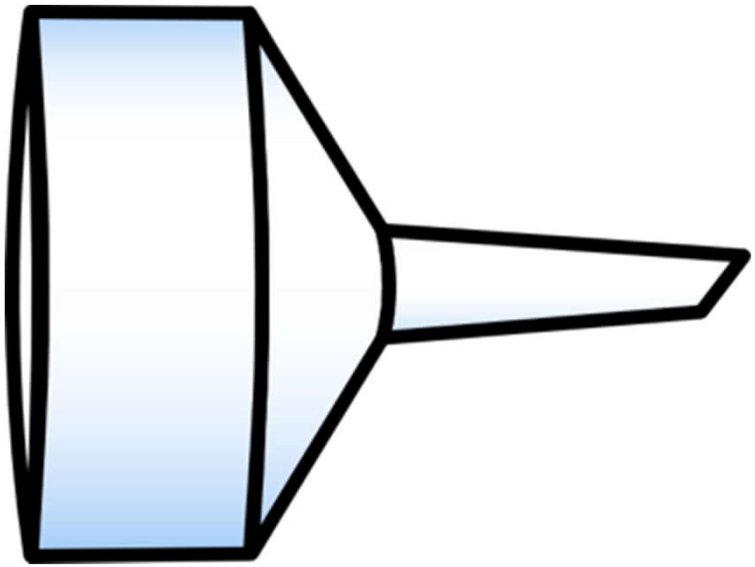
109 Java Standard Library classes and interfaces with protocols



Mining forums for protocol challenges

Discard extremely simple and familiar protocols (e.g. Iterator, Exception)

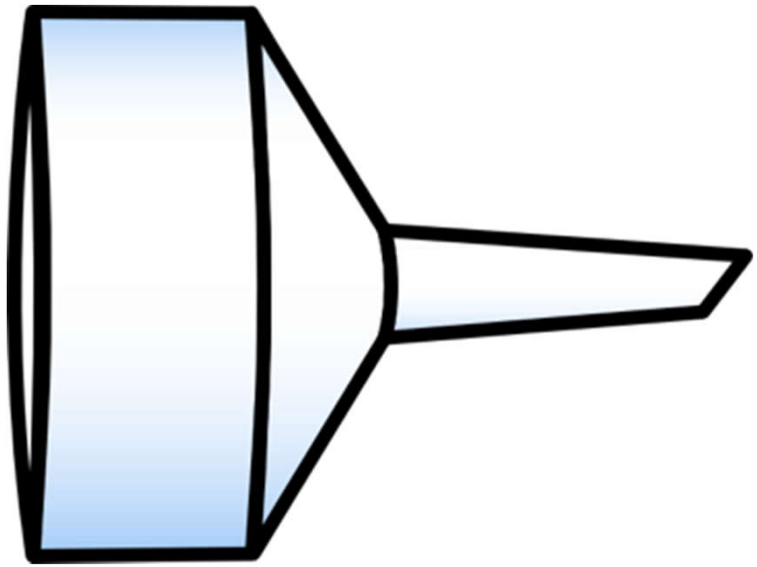
109 Java Standard Library classes and interfaces with protocols



Mining forums for protocol challenges

Discard extremely simple and familiar protocols (e.g. Iterator, Exception)

109 Java Standard Library classes and interfaces with protocols



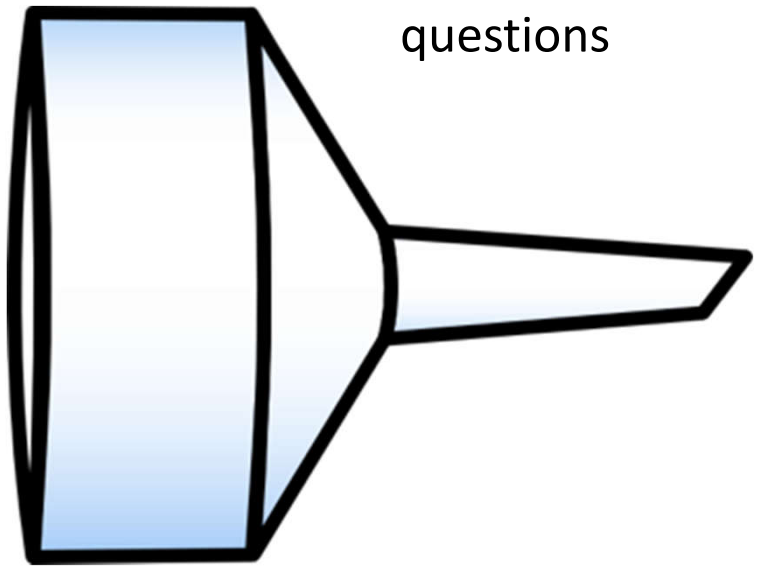
69 classes and interfaces

Mining forums for protocol challenges

Discard extremely simple and familiar protocols (e.g. Iterator, Exception)

Remove classes with fewer than 50 StackOverflow questions

109 Java Standard Library classes and interfaces with protocols



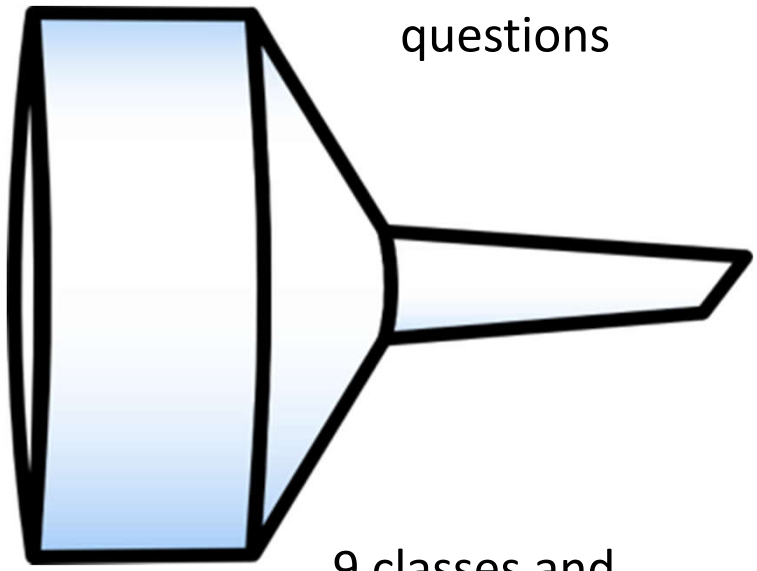
69 classes and interfaces

Mining forums for protocol challenges

Discard extremely simple and familiar protocols (e.g. Iterator, Exception)

Remove classes with fewer than 50 StackOverflow questions

109 Java Standard Library classes and interfaces with protocols



69 classes and interfaces

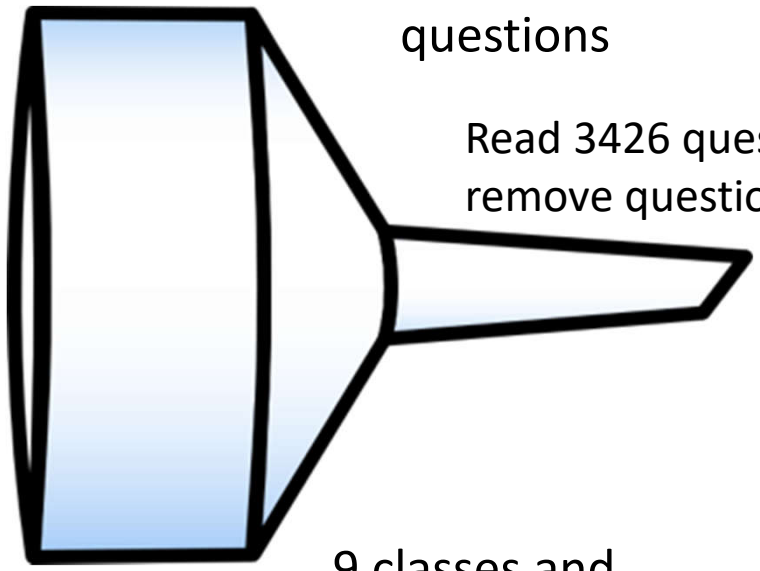
9 classes and interfaces

Mining forums for protocol challenges

Discard extremely simple and familiar protocols (e.g. Iterator, Exception)

Remove classes with fewer than 50 StackOverflow questions

109 Java Standard Library classes and interfaces with protocols



Read 3426 questions about 9 classes, and remove questions unrelated to a protocol

69 classes and interfaces

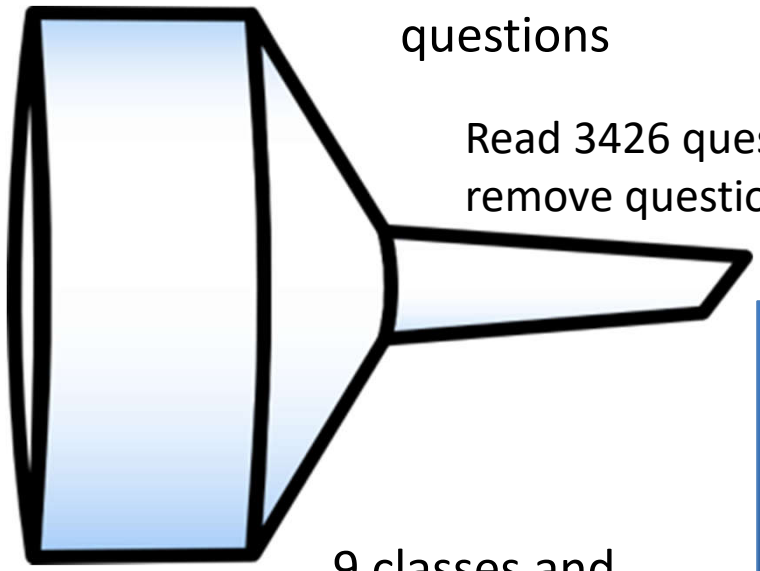
9 classes and interfaces

Mining forums for protocol challenges

Discard extremely simple and familiar protocols (e.g. Iterator, Exception)

Remove classes with fewer than 50 StackOverflow questions

109 Java Standard Library classes and interfaces with protocols



Read 3426 questions about 9 classes, and remove questions unrelated to a protocol

Socket
ResultSet
Timer
URLConnection

69 classes and interfaces

9 classes and interfaces

Observational study of protocols

- Participants: 6 experienced professional programmers
  - Work experience: minimum of 3.5 years, median 11 years
  - Worked with object-oriented languages and frameworks
- Tasks:
  - Based on questions found in forum mining
  - Greenfield programming and debugging
  - Resources: Eclipse, JavaDoc, code, browser
- Methodology:
  - Think-aloud laboratory study
  - Screens and speech recorded

Analysis

1. Transcribed participant think-aloud
2. Examine transcript for questions
3. Watch screen recordings to see how participants answer questions and approximately how long it took them
4. Performed open-coding on questions looking for similar questions that repeat

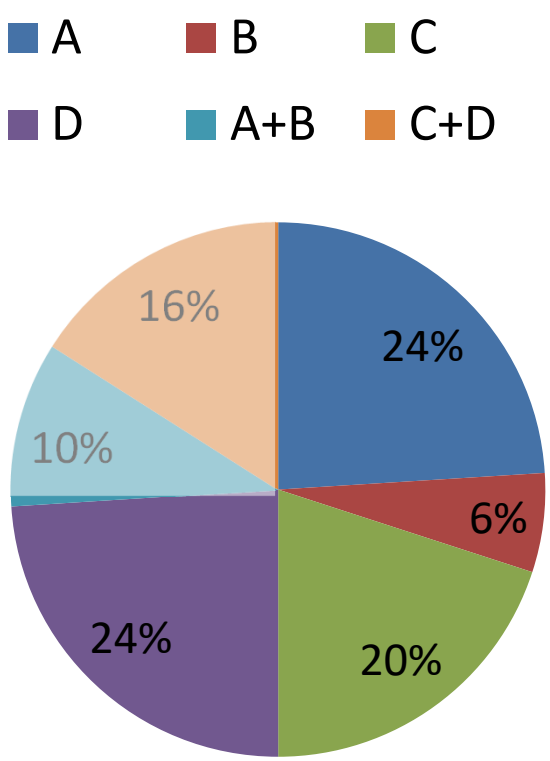
What questions were developers asking?

Participants' time was dominated by working on four categories of search problems:

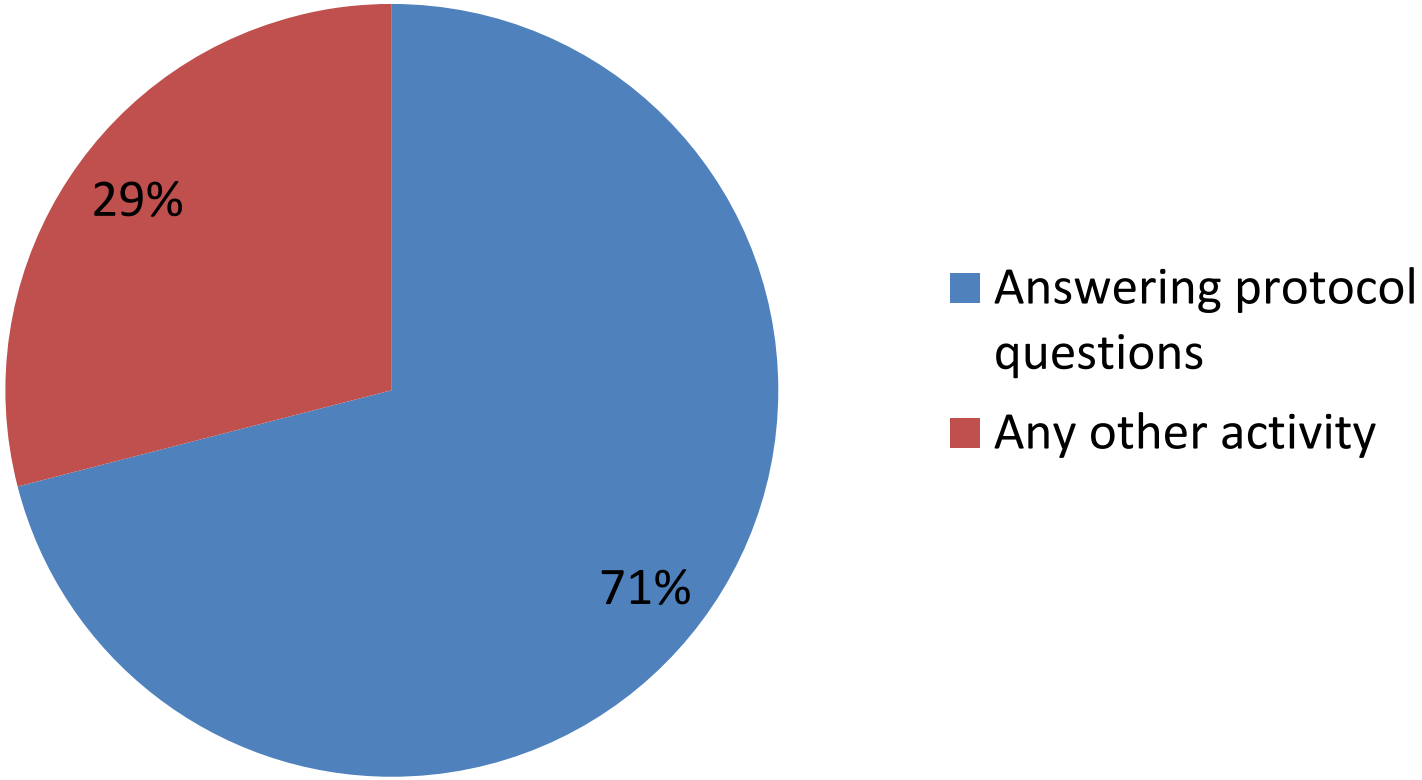
Example instances	Category
Is the TimerTask scheduled? Is [the ResultSet] x scannable?	What abstract state is the object in?
Can I schedule a scheduled TimerTask? What can I do on the insert row?	What are the capabilities of object in state X?
When can I call doInput? Which ResultSets can I update?	In what state(s) can I do operation Z?
How do I move from the insert row to the current row? Which method schedules a TimerTask?	How do I transition from state X to state Y?

Which questions were most frequent?

- A) What abstract state is the object in?
- B) What are the capabilities of object in state X?
- C) In what state(s) can I do operation Z?
- D) How do I transition from state X to state Y?

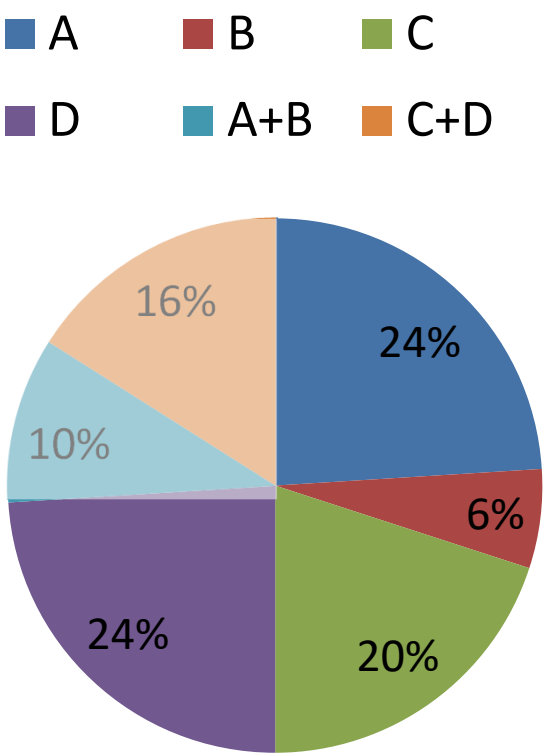


How much time was spent answering questions?

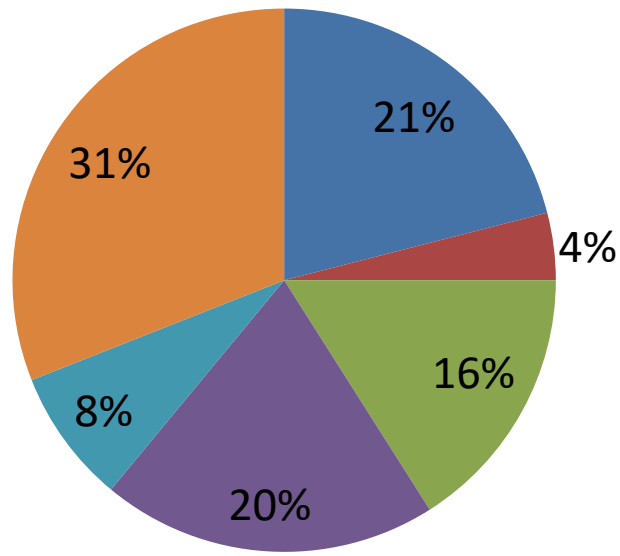


How long does it take to answer each question?

A) What abstract state is the object in?
B) What are the capabilities of object in state X?
C) In what state(s) can I do operation Z?
D) How do I transition from state X to state Y?



% of questions



% of time

Protocols Cause Problems

- Solid scientific evidence that protocols cause productivity issues
  - For a number of realistic tasks in a component-based development setting, developers spent >70% of task time answering protocol-related questions
- Grounding for follow-up experiment to evaluate assurance tools' impact on productivity
  - Relevant tasks and questions, and insights into barriers faced by developers
- Hypothesis: value of tools may be less about assurance and more about automatically-checked documentation
  - Challenges dominant paradigm among tool researchers

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**.

- compare: prior typestate work considered only changing interfaces [Strom and Yemeni, Deline and Fähndrich]

Typestate-oriented Programming is embodied in the language

PLAID

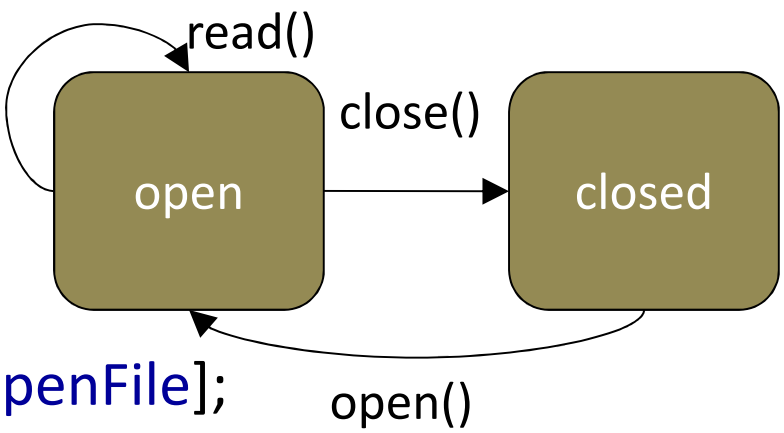
*Plaid (rhymes with "dad") is a pattern of Scottish origin, composed of multicolored crosscutting threads

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*

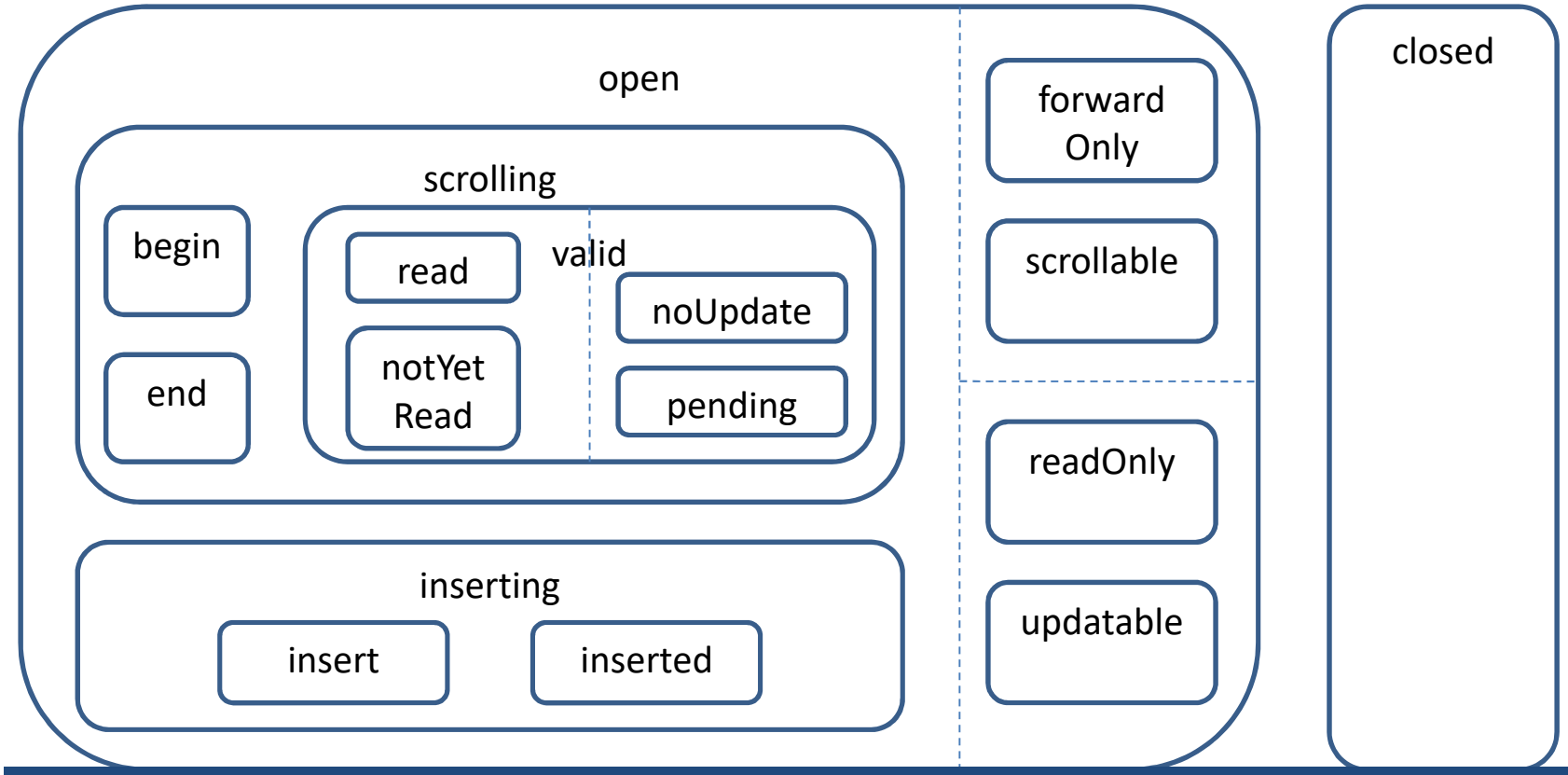
Values must be specified for each new field

:

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

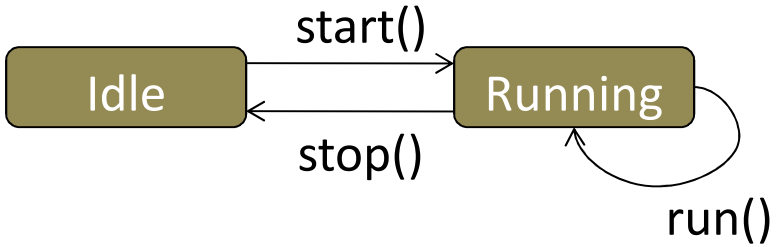
shared OpenFile@OpenFile is (almost) traditional object-oriented programming

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]

Example: Interactors




```
state Idle {
    void start() [Idle >> Running];
}
state Running {
    void stop() [Running >> Idle];
    void run(InputEvent e);
}
```

```
state MoveIdle extends Idle {
    GraphicalObject go;
    void start() [Idle >> Running] {
     this <- Running {
        void run(InputEvent e) {
           go.move(e.x,e.y);
        }
        void stop() [Running >> Idle] {
           this <- MoveIdle{}
        }
     }
    }
}
```

Typestate Checking Hypotheses

- Relatively simple permission mechanisms are sufficient to statically check typestate properties in most Plaid code
  - (for the exceptions, see Gradual Types, below)
- Both permissions and typestates express important design constraints, helping developers correctly evolve software
- Permissions can help make automated verification tools more effective

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

User experiment

- Goal: Evaluate interventions designed to improve protocol programmability
- Tasks: answer questions that rely on solving instances of the search problems
- Participants: student programmers
- Method: mixed-mode controlled laboratory experiment

Precedent

- API Usability relativity unresearched
- Demonstrations of substantial performance improvements in speed of API use:
  - Constructor (up to 11x) faster than factory method [Ellis 2007]
  - Method on starting object faster than other object [Stylos 2009]
  - With eMoose directives faster than without [Dekel 2009]
- Search tasks can be much easier with diagrams and structured text than with unstructured text [Larkin 1987]
- Theses precedents suggest that well designed interventions could improve protocol programmability

Experiment design

- Between-subjects study – 20 participants
  - split between control and intervention conditions
- 3 Java library state search tasks
  - URLConnection, ResultSet, Timer
  - 4 state search questions each
- Output measures:
  - Task completion time
  - Correctness
- Validity
  - Observational study emphasizes external validity
  - Experiment focuses on internal validity
  - The external validity of the experiment is enhanced by the qualitative studies

Interventions

- Documentation
  - JavaDoc (control)

URLConnection
addRequestProperty
connect
getContent
getInputStream
setDoInput
…

Interventions

- Documentation
  - JavaDoc (control)
  - PlaidDoc

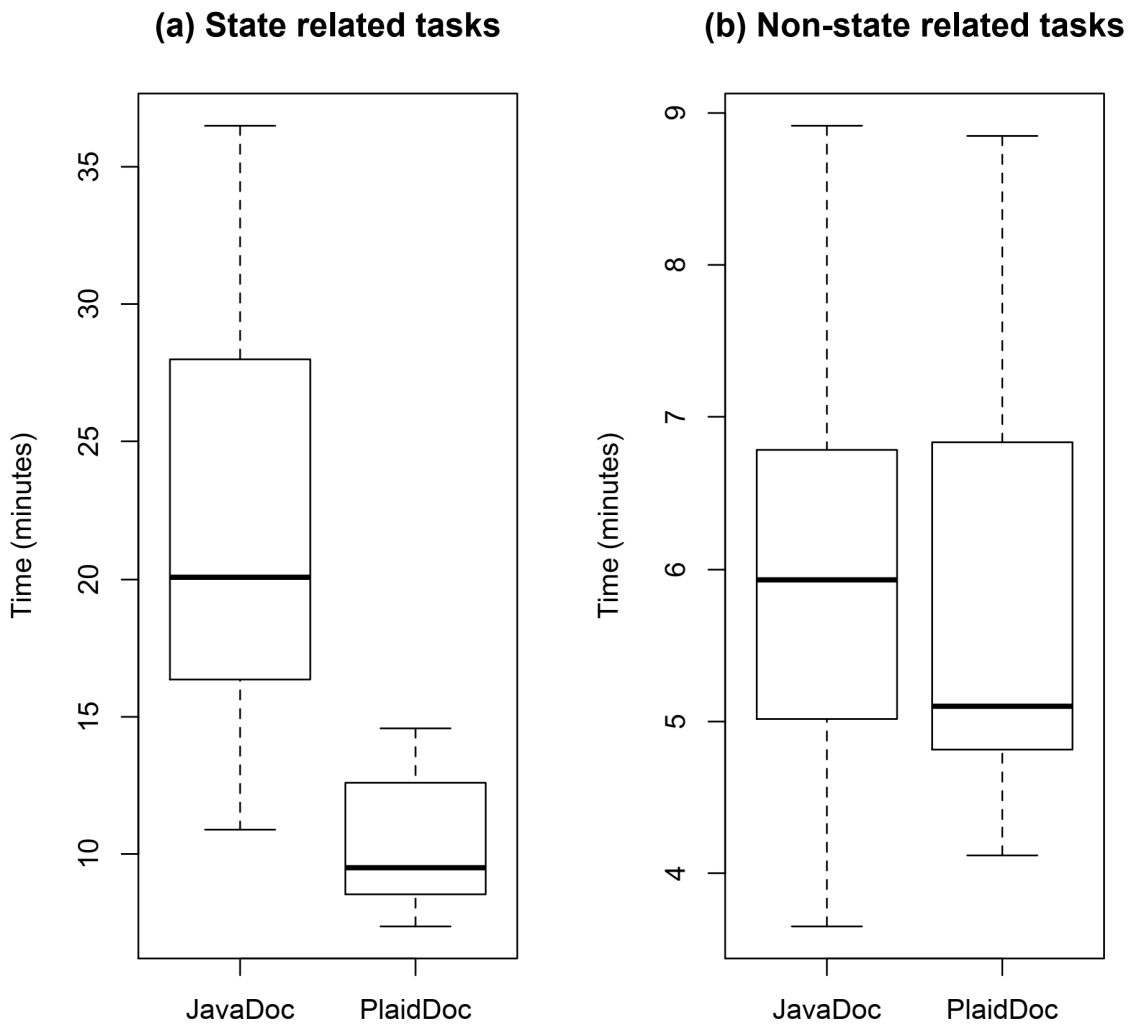
URLConnection
  addRequestProperty
  connect
  getContent
  getInputStream
  setDoInput
  …

URLConnection
  Disconnected
    addRequestProperty
    connect [-> *Connected*]
    setDoInput
  Connected
    getContent
    getInputStream

+ ASCII statechart

Results – Time On Task

- Time between end of question and final answer
- State task mean:
  - Plaiddoc: 10.3 min
  - Javadoc: 22.4 min
  - p<0.001
- Non-state mean:
  - Plaiddoc: 5.77 min
  - Javadoc: 5.95 min
  - p=0.802



Results – Correctness

- Participants confirmed "final answers"
- Sometimes wrong
  - Experiment proceeded regardless of correctness
  - Participants were not told if they were correct

Category	Plaiddoc	Javadoc
Correct	151	143
Incorrect	2	15
Timed-out	7	2

- Contingency table analysis, p=0.002

Other Studies We've Done

- Is Structural Subtyping Useful? An Empirical Study
  - Most object-oriented languages use nominal subtyping, but the research literature uses structural subtyping.
  - Are there benefits of structural subtyping in practice?
    - Methods that specify a specific nominal type but only use a subset of its protocol (client-side: suggests a narrower structural type would be useful)
    - How often is UnsupportedOperationException thrown? (implementation-side: suggests a narrower structural type would be useful)
- Inter-app Communication in Android: Developer Challenges
  - Documents typechecking kinds of issues related to messages sent between Android apps
  - Looked at what messages were sent/received by different apps, how many were common/unique, and how they changed over time

Bibliography

Papers Referenced in This Talk

- **An Empirical Study of Object Protocols in the Wild.** Nels E. Beckman, Duri Kim, and Jonathan Aldrich. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP '11), 2011.
- **Searching the State Space: A Qualitative Study of API Protocol Usability.** Joshua Sunshine, James Herbsleb and Jonathan Aldrich. Proc. International Conference on Program Comprehension (ICPC), 2015.
- **Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming.** Joshua Sunshine, James Herbsleb, and Jonathan Aldrich. Proc. European Conference on Object-Oriented Programming, 2014.
- **Is Structural Subtyping Useful? An Empirical Study.** Donna Malayeri and Jonathan Aldrich. In Proceedings of the European Symposium on Programming (ESOP '09), March 2009.
- **Inter-app Communication in Android: Developer Challenges.** Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. Proc. Mining Software Repositories (MSR), 2016.