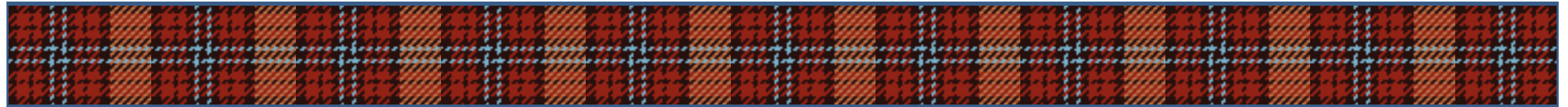
PLAID: Programming with Typestate and Permissions



Karl Naden

Carnegie Mellon University

OSCON

July 28, 2011

Carnegie Mellon University
School of Computer Science

Motivation

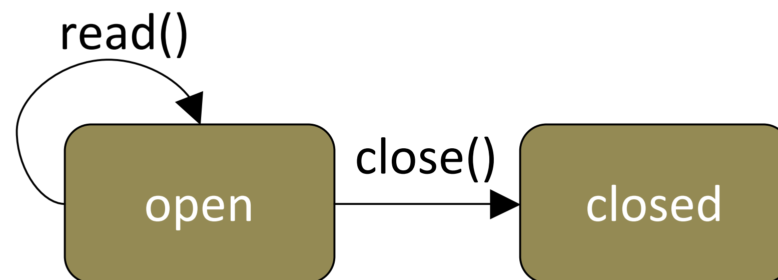
- **Principles**
  - Object-Oriented modeling provides a powerful metaphor for designing and understanding programs
  - Types aid understanding and correctness by providing checkable documentation

- **Goals for Plaid:**
  - Build an object oriented language with explicit support for objects that follow protocols
  - Develop a static type system to help programmers reason about protocols

PLAID

Object Protocols

- A protocol dictates an order on method calls to the object
- Example: FileReader
  - Sometimes you can read from the file, sometimes you cannot
  - Depends on whether the File is **Open** or **Closed**
- Modeled by **Abstract States**
  - read() method only available in the Open state
  - Calling close() **changes** the abstract state of the object to Closed
- Recognized in design tools
  - State Machines
  - UML State Charts



PLAID

Object Protocols in Java

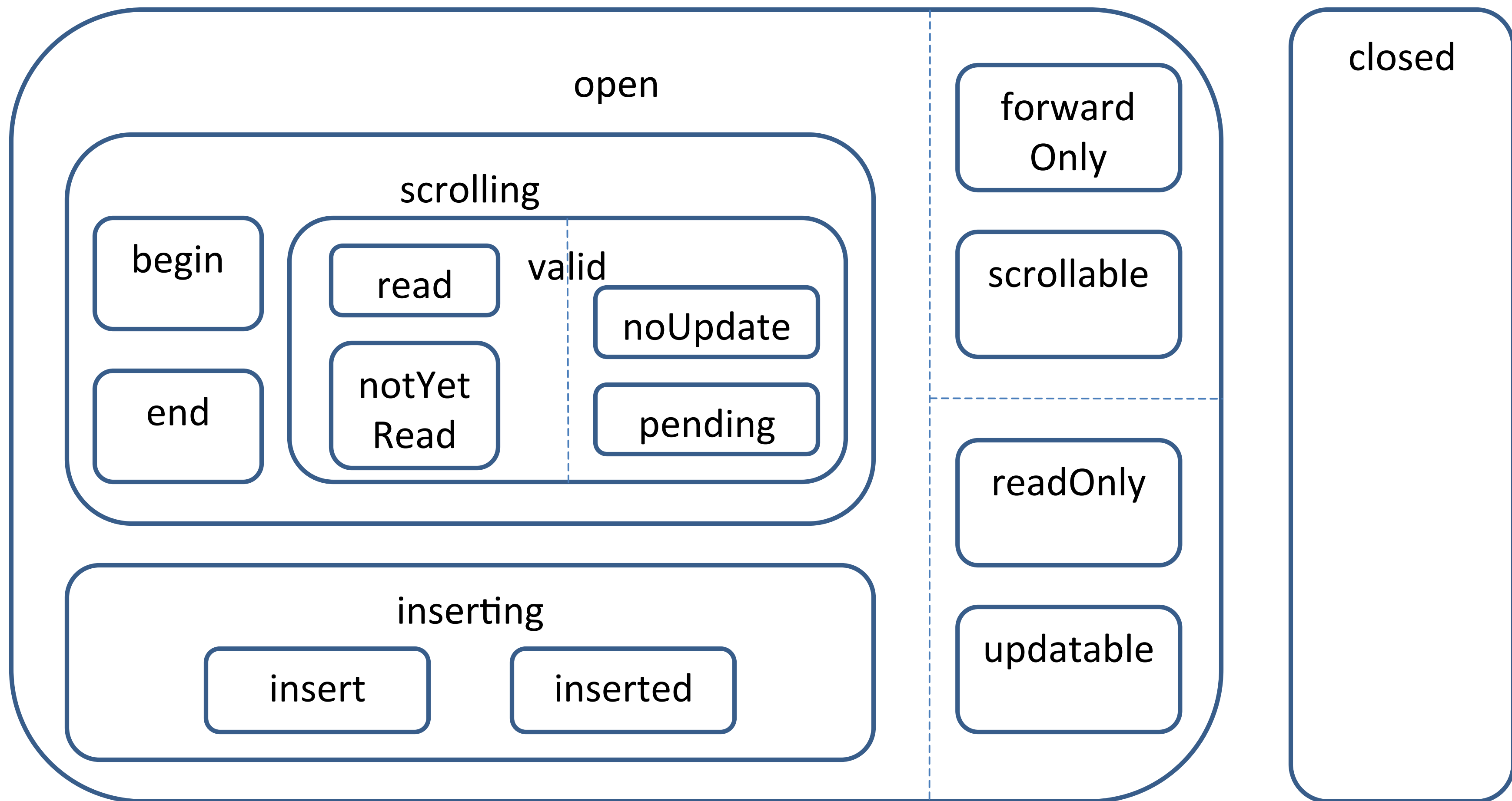
```
package java.io;
class FileReader {
    int read() { … }
    …
    /** Closes the stream and releases any system resources associated with it.
    Once the stream has been closed, further read(), ready(), mark(), reset(), or
    skip() invocations will throw an IOException. Closing a previously closed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

- Why is this a poor model?
  - Interface does not change
  - The abstract state is not explicit
  - The protocol is encoded as textual comments

PLAID

ResultSet State Space

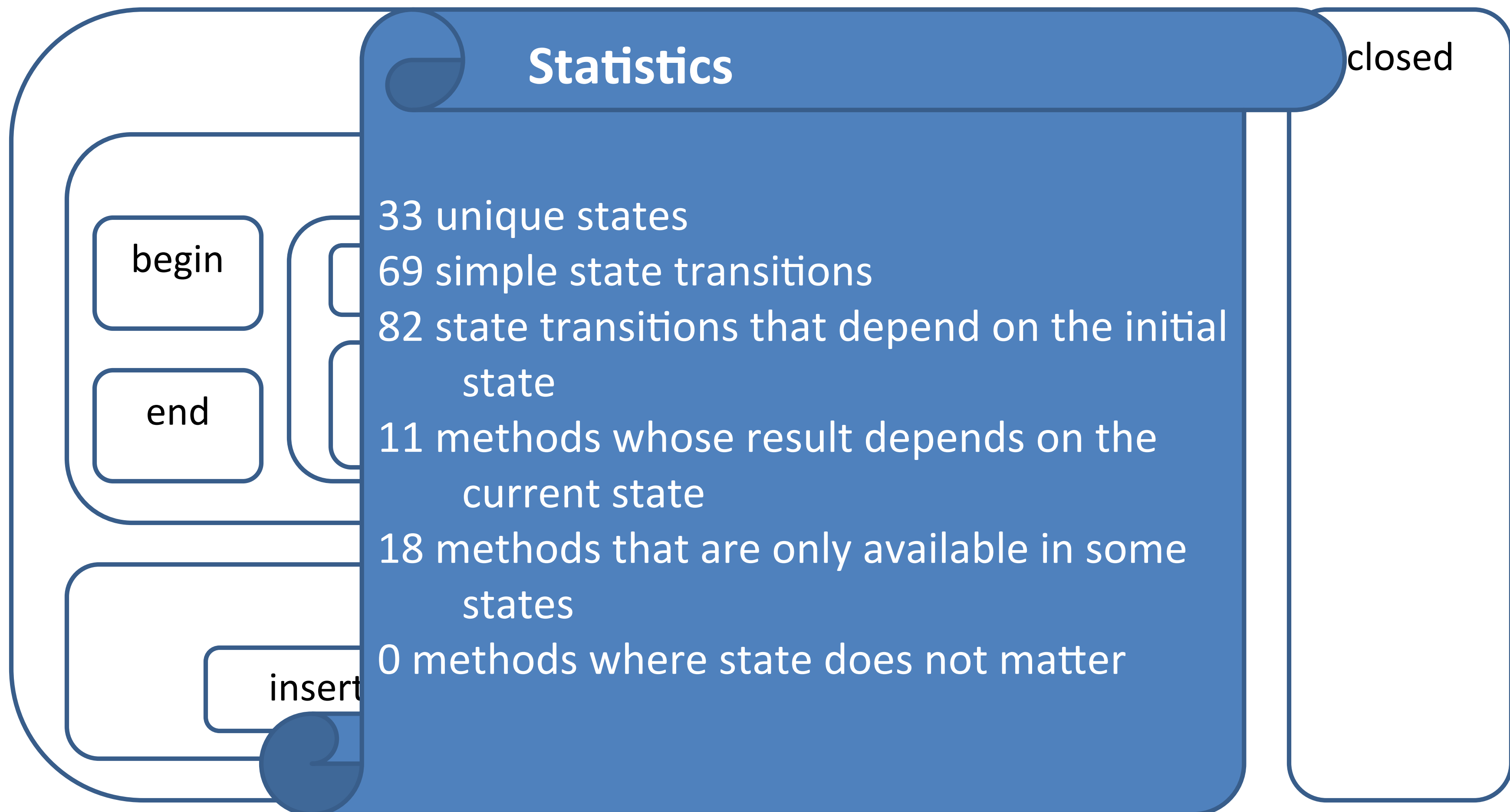
Java Database Connectivity (JDBC) Library State Space



PLAID

ResultSet State Space

Java Database Connectivity (JDBC) Library State Space



PLAID

Why Do We Need Plaid?

1. Modeling important for complex protocols

   ResultSet, Frameworks (ASP.NET, Swing, etc.), Java iterators

   - Explicit questions about states
   - Errors that refer to the design
   - Types for documentation and verification of definition and use

2. Object Protocols are prevalent:

   - Study of protocols in Java Standard Library and open source projects
   - At least 7% of Java classes **define** a protocol
     (compare to 2.5% of classes which are generic)
   - At least 13% of Java classes **use** a protocol

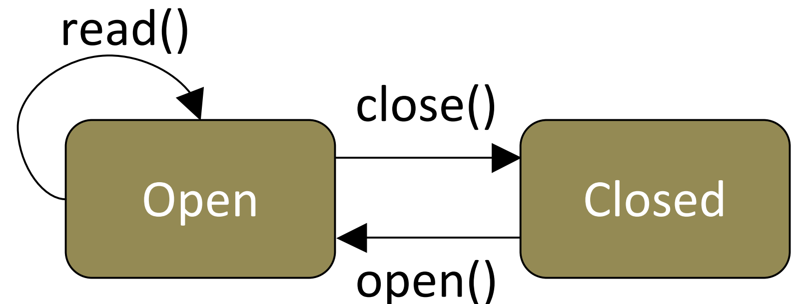
Typestate

- Idea: break object definition up into **Typestates**
- Like Types:
  - Provide and **Interface**
  - Define **Behavior** via methods
  - Specify **Representation** via fields
- But not fixed:
  - An object's typestate can change over its lifetime

PLAID

Dynamic Typestates in Plaid

- **state** = class
- **case of** = extends




```
state File {
    val filename;
}
```

```
state Open case of File {
    val filePtr;
    method read() {…}
    method close() { this <- Closed; }
}
```

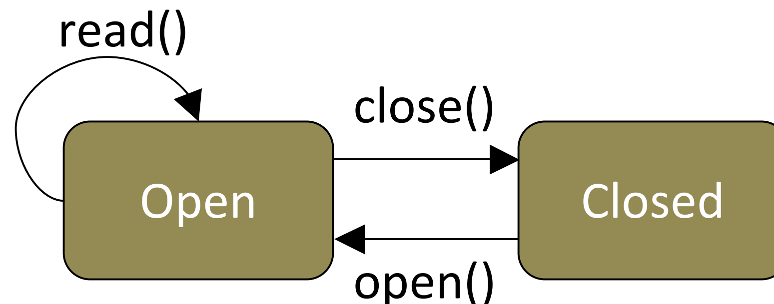
```
state Closed case of File {
    method open() { this <- Open; }
}
```

PLAID

Asking State Questions

- Query dynamic state with **match**
- Syntax notes
  - First class functions
  - Expression based syntax returns value of last expression in a block

```
val readChar = fn (file) => {
    match(file) {
        case Open { /* No Op*/ }
        case Closed { file.open(); }
    };
    file.read();
};
```



PLAID

Types for Typestate

- States define nominal types like Java classes
- Careful with transitions
  - Aliasing
- Solution: **Permissions**

```
val closeTwo =
  fn (openFile1, openFile2) => {
    openFile1.close();
    openFile2.close();
  };

val badFunction =
  fn (openFile) => {
    closeTwo(openFile, openFile);
  };
```

PLAID

Permissions

- Two purposes:
  1. Summarize level of aliasing for referenced object
  2. Determine what actions can be taken via the reference
- **unique** permission:
  1. No other usable (non-**none**) aliases exist
  2. No restrictions on use
- **immutable** permission:
  1. Any other usable (non-**none**) alias must be **immutable**
  2. Cannot change the state of the object
- **none** permission:
  - Arbitrary aliases
  - Unusable

PLAID

Annotating States

- All types include a permission and a nominal typestate
- **Change Types** for the receivers of methods
  - Starting permission and type
  - Ending permission and type

```
state Open case of File {
  val unique FileRef filePtr;
  method
    read()[immutable Open >>
           immutable Open] {…}
  method
    close()[unique Open >> unique Closed]
    { this <- Closed; }
}
```

```
state Closed case of File {
  method
    open()[unique Closed >> unique Open]
  { this <- Open; }
}
```

PLAID

Typing and Rejecting Programs

- Change Types for parameters too
- Permissions and Types tracked through body of function
  - closeTwo fulfills its contract
  - In badFunction, we need a **unique** when we only have a **none**

```
val closeTwo =
  fn (unique Open >> unique Closed f1,
      unique Open >> unique Closed f2)
  => {
    //f1:unique Open, f2: unique Open
    f1.close();
    //f1:unique Closed, f2: unique Open
    f2.close();
    //f1:unique Closed, f2: unique Closed
  };
val badFunction =
  fn (unique Open >> unique Closed f)
  => {
    //f:unique Open
    closeTwo(f,  //f:none Open
             f); //Error
  };
```

PLAID

Using immutable

- Sharing **immutable** objects
  - Same object can be passed into readTwo
- Regaining **unique**
  - Possible in some cases

```
val readTwo =
  fn (immutable Open>>immutable Open f1,
      immutable Open>>immutable Open f2)
  => { f1.read() + f2.read(); };

val badFunction =
   fn (unique Open >> unique Open f)
   => {
     //f:unique Open
     readTwo(f,  //f:immutable Open
             f); //f:immutable Open
     //f:unique Open
   };
```

PLAID

More coming to Plaid

- Other Permissions:
  - shared, full, pure, …
- Composition of states and protocols
- Flexibility in the typesystem
  - Permission casts
  - Gradual typing (types only where important)
- Static analyses using permissions
  - infer concurrency

PLAID

Challenges

- Keeping the type system manageable
  - Flexibility
  - Defaults
  - Aggressive tracking of permissions
  - Casting and other escape hatches
  - Gradual typing

- We need YOU to
  - Try the language
  - Tell us what works and what doesn't
  - contribute

PLAID

The Plaid Language

- Plaid models object protocols and provides
  - Language support for protocols and abstract states
  - A type system to help document and check protocols and their uses
- Try it
  - In your browser: http://www.plaid-lang.org/
  - Download: http://code.google.com/p/plaid-lang/
- Help Out
  - Contact us if you are interested in helping to move Plaid forward

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