Typestate-Oriented Programming

Jonathan Aldrich

Joshua Sunshine

Darpan Saini

Zachary Sparks

Carnegie Mellon

isr institute for SOFTWARE RESEARCH

Object-Oriented Modeling

- History: Simula 67 was created to facilitate modeling
- Object-orientation still works today because of its modeling power
  - Objects - model real-world or conceptual entities
  - Fields - model object properties and changes to those properties over time
  - Methods - model actions that can be performed on objects
  - Subtyping - models commonality and variation between objects

- Models of state change are very limited. What about:
  - New **properties** that did not exist before?
  - New **actions** that can be performed?
  - Conceptual **variations** in an object's interface over time?

State Change Is Ubiquitous

In the world

- Egg, caterpillar or butterfly?
- Working, sleeping, eating, or playing?
- Hungry or full?
  - The OOPSLA Ice Cream Social is not far off!

In software systems

- Streams: open, EOF, or closed?
- Iterators: has next or not?
- Collections: empty or not?
- Exceptions: cause set or not?

Design: UML Statecharts

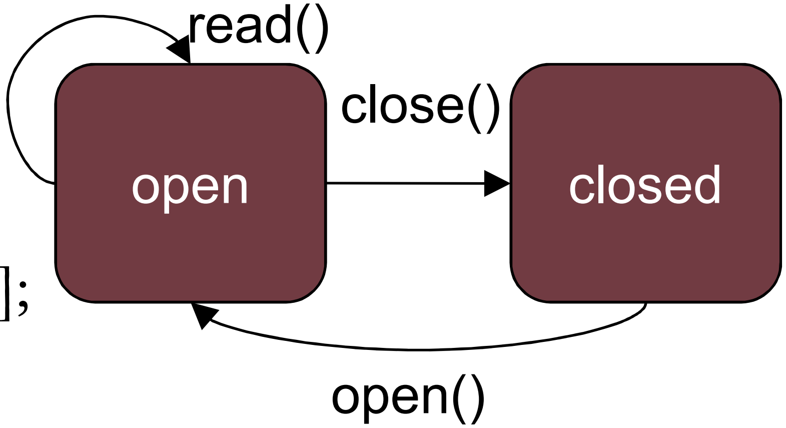
If state is ubiquitous,
perhaps languages should support it!

We build on **Typestate**, a type-based approach for tracking states

Typestate-Oriented Programming

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

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

State transition



Different representation

New methods

Typestate-Oriented Programming

- Definition: A **programming paradigm** in which:

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

  - Compare: embedded system CASE tools

  each object has a **typestate** that is **changeable** and **statically trackable**,

  - Compare: plain OO classes
  - Compare: dynamically typed state proposals (actors, roles, modes, …) or the State design pattern

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

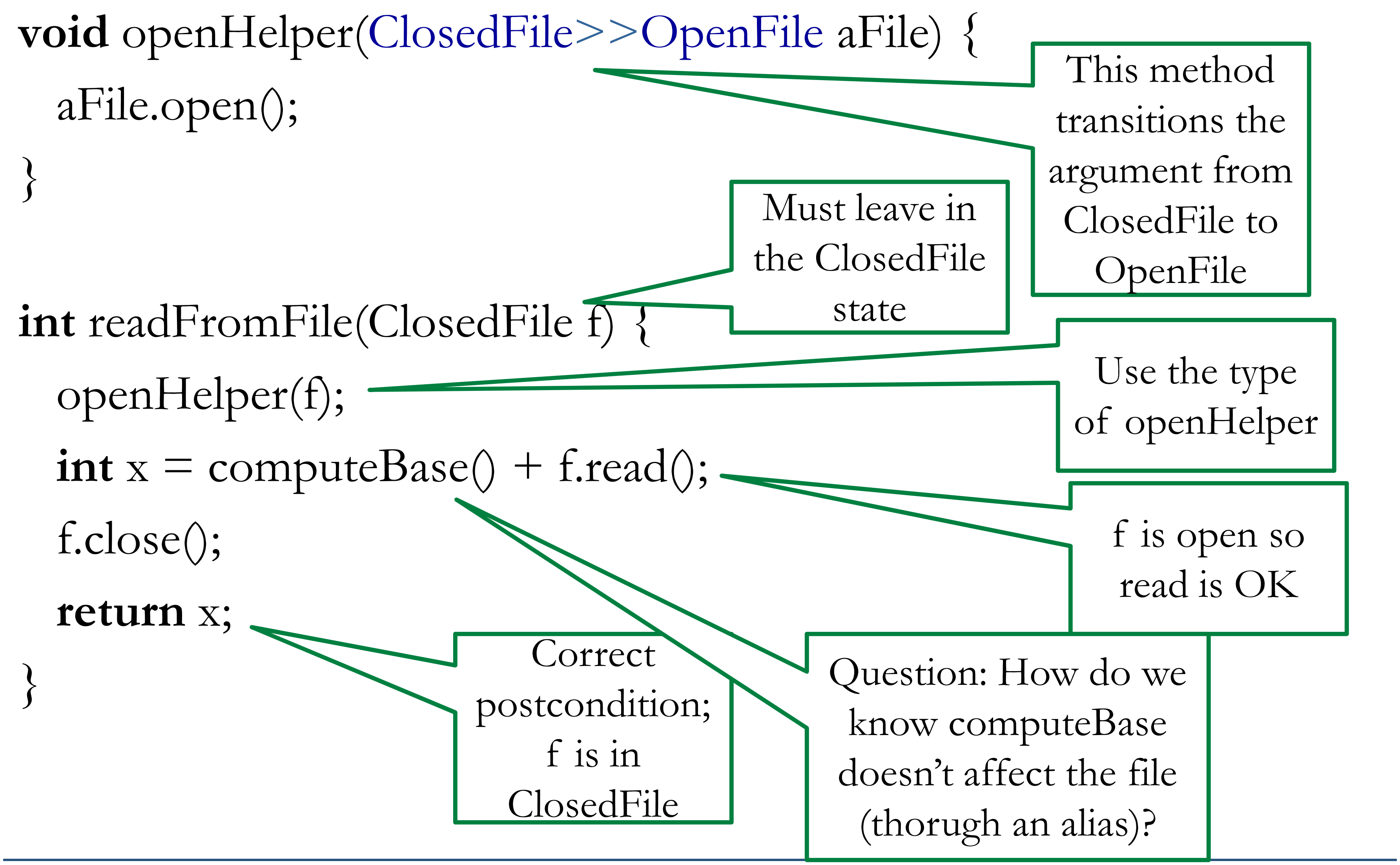
  - Compare: typestate analysis on top of OO

- In our model interface, representation, and behavior change with an object's typestate, but object identity does not
  - Related: class change proposals (e.g. Fickle)

Why Put Typestate in the Language?

- Language influences thought [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

Checking Typestate



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

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

This method transitions the argument from ClosedFile to OpenFile

Must leave in the ClosedFile state

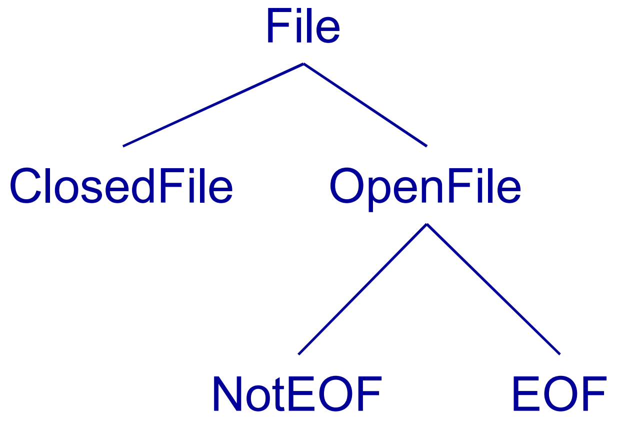
Use the type of openHelper

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
- **immutable** OpenFile
  - Cannot change the File
    - Cannot close it
    - Cannot write to it, or change the position
  - Aliases may exist but do not matter
- **shared** OpenFile@NotEOF
  - File is aliased
  - File is currently not at EOF
    - Any function call could change that, due to aliasing
  - It is forbidden to close the File
    - OpenFile is a *guaranteed* state that must be respected by all operations through all aliases
- **none** – no permission

Implementing Typestate Changes

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

Typestate change primitive

Values must be specified for each new field

:

Parametric Polymorphism

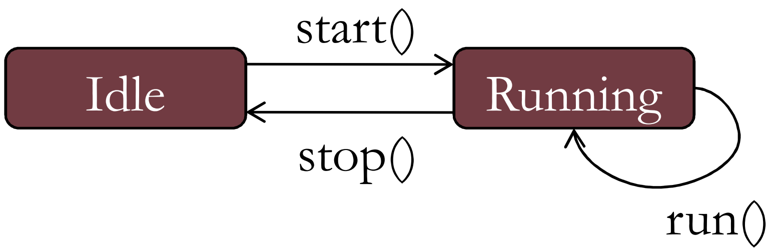
```
state Collection {
  type TElem;

  void add(TElem>>none e);

  TElem removeAny();


}
```

Type parameter must now include state and permission

Adding an element to the collection removes the client's permission to it (e.g. to ensure unique objects are unaliased)

If we want to get an element, we must remove it from the collection (to avoid aliasing).

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{}
      }
    }
  }
}
```

Current Work: Typestate-Oriented Programming

PLAID is a new typestate-oriented programming language

Features:

- Java-like syntax, as presented in this talk
- Permissions describe aliasing on all objects
- Concurrency-by-default execution model
    - See “Concurrency By Default” Onward! ’09 companion paper
- Gradual types
- Advanced modularity constructs (e.g. abstract types)
- Composition mechanism similar to traits (replaces inheritance)

Typestate-Oriented Programming

- Objects change their state
  - But until now, there's been no language support for state change
- Typestate-oriented programming makes states explicit
  - helps document, check and implement state changes
- Potential benefits
  - Communication, clarity, correctness, reuse
- PLAID
  - New typestate-oriented programming language

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