First-Class State Change in

PLAID

Karl Naden
with Joshua Sunshine, Sven Stork
Jonathan Aldrich, and Éric Tanter
OOPSLA 10/27/2011
• Things all around us are changing state
  • Butterfly: Egg ➔ Caterpillar ➔ Chrysalis ➔ Imago
• Things all around us are changing state
  • Butterfly: Egg ➔ Caterpillar ➔ Chrysalis ➔ Imago
  • You: Awake ↔ Asleep
• Things all around us are changing state
  • Butterfly: Egg ➔ Caterpillar ➔ Chrysalis ➔ Imago
  • You: Awake ↔ Asleep
• In Programming, too
  • File: Open ↔ Closed
• Things all around us are changing state
  • Butterfly: Egg ➔ Caterpillar ➔ Chrysalis ➔ Imago
  • You: Awake ↔ Asleep
• In Programming, too
  • File: Open ↔ Closed
  • Java Exception: Cause Not Set ➔ Cause Set
• Things all around us are changing state
  • Butterfly: Egg ➔ Caterpillar ➔ Chrysalis ➔ Imago
  • You: Awake ↔→ Asleep
• In Programming, too
  • File: Open ↔→ Closed
  • Java Exception: Cause Not Set ➔ Cause Set
• Different abilities depending on the state
  • Butterfly can only fly as an Imago
  • read() only available in the Open state of File
• Things all around us are changing state
  • Butterfly: Egg $\rightarrow$ Caterpillar $\rightarrow$ Chrysalis $\rightarrow$ Imago
  • You: Awake $\leftrightarrow$ Asleep

• In Programming, too
  • File: Open $\leftrightarrow$ Closed
  • Java Exception: Cause Not Set $\rightarrow$ Cause Set

• Different abilities depending on the state
  • Butterfly can only fly as an Imago
  • read() only available in the Open state of File
  • Error when the current state does not support the action
• An **Object Protocol** [Strom, Yemini ’86] dictates an **order** on method calls:
  • Has a **finite number** of **abstract states** in which different method calls are valid;
  • Specifies **transitions** between abstract states that occur as a part of some method calls.
• An **Object Protocol** [Strom, Yemini '86] dictates an **order** on method calls:
  • Has a **finite number** of **abstract states** in which different method calls are valid;
  • Specifies **transitions** between abstract states that occur as a part of some method calls.

• **State Charts** [Harel '87]: **File**
  • States: File, Open, Closed
  • Methods: read(), open(), closed()
  • Transitions: close(), open()
• States exists
  • In documentation
    /** @throws IllegalStateException if task was already scheduled or
cancelled, timer was cancelled, or timer thread terminated.
*/
    private void sched(TimerTask task, long time, long period) { … }
• States exists
  • In documentation
    /** @throws IllegalStateException if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated. */
    private void sched(TimerTask task, long time, long period) { … }
  • Encoded as lower-level constructs
    if (task.state != TimerTask.VIRGIN) throw new IllegalStateException(…);
• States exists
  • In documentation
    ```java
    /** @throws IllegalStateException if task was already scheduled or
cancelled, timer was cancelled, or timer thread terminated. */
    private void sched(TimerTask task, long time, long period) { ... }
    ```
  • Encoded as lower-level constructs
    ```java
    if (task.state != TimerTask.VIRGIN) throw new IllegalStateException(...);
    ```
• Problems
  • States from design are obfuscated
    • Code difficult to understand
  • If checks forgotten, results difficult to debug, e.g.
    • Non-specific `NullPointerException`
    • Data corruption (overwritten `TimerTask`)
• Common [Beckman ‘11]
  • 7% of Java classes define object protocols
    • 3x as many as define generics
  • 13% use them
• Common [Beckman ’11]
  • 7% of Java classes define object protocols
    • 3x as many as define generics
    • 13% use them
• Complex:
• Plaid programming language
  – First class states and transitions
• **Plaid** programming language
  – First class states and transitions

• Overview
  – Syntax and semantics of states and transitions
  – Trait-based reuse
Two Encodings of File

Open
read()
close()
open()
Closed
1) States

class File {

    private FileResource filePtr = null;

}
1) States

```java
class File {
    private FileResource filePtr = null;

    // substate of file determined by null-ness of filePtr field
}
```
class File {

    private FileResource filePtr = null;

    state File {
    }

    state Open case of File {
    }

    state Closed case of File {
    }

    1) States

    state File {
    }

    }
Two Encodings of File

class File {

  private FileResource filePtr = null;
  public int read() {…}

  public void close() {…}
  public void open() {…}

}

state File {

}

state Open case of File {

}

state Closed case of File {

}
class File {

    private FileResource filePtr = null;
    public int read() {...}

    public void close() {...}
    public void open() {...}

    state File {
    
    }

    state Open case of File {

        method read() {...}
        method close() {...}
    
    }

    state Closed case of File {

        method open() {...}
    
    }

    2) Methods

}
class File {

    private FileResource filePtr = null;
    public int read() {
        if (filePtr == null)
            throw new IOException
        else ...
    }
    public void close() {...}
    public void open() {
        if (filePtr == null) {...}
    }
}

state File {

}

state Open case of File {

    method read() {...}
    method close() {...}
}

state Closed case of File {

    method open() {...}
}

2) Methods
3) Representation

class File {
    private String filename;
    private FileResource filePtr = null;
    public int read() {
        if (filePtr == null) {
            throw new IOException
        } else ...
    }
    public void close() {...}
    public void open()
    { if (filePtr == null) {...} }
}

state File {
    val filename;
}

state Open case of File {
    val filePtr;
    method read() {...}
    method close() {...}
}

state Closed case of File {
    method open() {...}
}
```java
class File {
    private String filename;
    private FileResource filePtr = null;
    public int read() {
        if (filePtr == null)
            throw new IOException
        else ...
    }
    public void close() {...; filePtr = null;}
    public void open() {
        if (filePtr == null) {
            filePtr = ...
        }
    }
}
```

4) Transitions

```java
state Open case of File {
    val filePtr;
    method read() {...}
    method close() {...}
}
state Closed case of File {
    method open() {...}
}
```
4) Transitions

```java
class File {
    private String filename;
    private FileResource filePtr = null;
    public int read() {
        if (filePtr == null)
            throw new IOException
        else ...
    }
    public void close() {...; filePrt = null;}
    public void open() {
        if (filePtr == null) { filePtr = ... }
    }
}
```

```java
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 { val filePtr = ... }
    }
}
```
• Plaid encoding advantages:

  – Design salient in the code
  – Succinct: fewer explicit checks
  – Errors handled safely and informatively
• **Goal**: flexibility of Traits [Ducasse ‘06] with benefits of protocols
  - Break protocols up into elemental protocols
  - Compose them
• ReadStream = Position + Reader
• ReadStream = Position + Reader
• ReadStream = Position + Reader

Protocols in two separate dimensions
• ReadStream = Position + Reader

```java
state Reading {
    method read() {
        /* read character */
        this.next();
    }
}
```
• ReadStream = Position + Reader

\[
\text{state NotEnd case of Position} \{ \\
\text{method next()} \{ \\
\text{/* move position forward */} \\
\text{if (/* at the end */) } \\
\text{this \rightarrow End } \\
\text{this.next();} \\
\} \\
\}
\]

\[
\text{state Reading} \{ \\
\text{method read()} \{ \\
\text{/* read character */} \\
\text{this.next();} \\
\} \\
\}
\]
val rs = new Reading with
   NotEnd { val coll = [1,2]; ... };
```java
val rs = new Reading with
    NotEnd { val coll = [1,2]; ... };
rs.read();
```

val rs = new Reading with
  NotEnd { val coll = [1,2]; ... };
rs.read(); //no state change
val rs = new Reading with
  NotEnd { val coll = [1,2]; … }
rs.read(); //no state change
rs.read();
```scala
val rs = new Reading with
  NotEnd { val coll = [1,2]; ... };
rs.read(); //no state change
rs.read(); // this \(\Rightarrow\) End

No change in the Reader dimension!
```
Solution:
State Members allow the incoming state to be determined dynamically
Solution:
State Members allow the incoming state to be determined dynamically.

```plaintext
state Position {
  val endState = End;
  /* ... */
}
```
Solution:
State Members allow the incoming state to be determined dynamically

```
state Position {
  val endState = End;
  /* ... */
}
```

```
state NotEnd case of Position {
  method next() {
    /* ... */
    if (/* at the end */) {
      this ← this.endState
    }
  }
}
```
val rs = new Reading with
NotEnd { val coll = [1,2]; ...;
    val endState = ReadEnd with End; };
rs.read(); //no state change
rs.read();
val rs = new Reading with
  NotEnd { val coll = [1,2]; ...;
  val endState = ReadEnd with End; }
rs.read(); //no state change
rs.read(); // this → ReadEnd with End

Both dimensions change together!
• ReadStream = Reader + Position

```scala
def read(): Unit = {
  // read character
  next()
}
```

```scala
val endState = ReadEnd
with End;
```

```scala
state Reader {  
  method read() {  
    /* read character */  
    this.next();  
  }  
}
```

```scala
state ReadStream = Position {  
  val endState = ReadEnd with End;
} with Reader
```
Reuse as a WriteStream = Writer + Position

```scala
state Writing {
  method write() {
    /* write character */
    this.next();
  }
}

state WriteStream = Position {
  val endState = WriteEnd with
    End;
} with Writer
```
• Or as a ReadWriteStream = Reader + Writer + Position

\[
\text{state ReadWriteStream} = \text{Position} \{ \\
\text{val endState} = \text{ReadEnd with} \\
\text{WriteEnd with} \\
\text{End;}
\} \text{ with Reader with Writer;}
\]
• Object Protocols in Plaid are
  – More concise, understandable, and safer
  – Reusable and composable
• Object Protocols in Plaid are
  – More concise, understandable, and safer
  – Reusable and composable

• Open up new possibilities
  – Visualization tools
  – More helpful error messages
  – Static checking
• Object Protocols in Plaid are
  – More concise, understandable, and safer
  – Reusable and composable

• Open up new possibilities
  – Visualization tools
  – More helpful error messages
  – Static checking

PlaID Demo, 2pm Galleria III