A Browser for Incremental Programming

Nathanael Schärli (SCG, University of Bern) and
Andrew P. Black (CSE, Oregon Health & Science University)
A Browser for Incremental Programming

Nathanael Schärli (SCG, University of Bern)
and
Andrew P. Black (CSE, Oregon Health & Science University)
A Browser for Incremental Programming

Nathanael Schärli (SCG, University of Bern) and Andrew P. Black (CSE, Oregon Health & Science University)
What is Incremental Programming?

• What we do in Smalltalk!
• One of the Extreme Programming (XP) practices
• Characterized by some patterns of work that should be familiar to you:
Incremental work patterns

- Programming with limited knowledge
- Working in multiple contexts
- Refactoring
- Testing
- Understanding how classes collaborate
- Understanding what is still missing
Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
  - Working in multiple contexts

- Refactoring

- Testing

- Understanding how classes collaborate

- Understanding what is still missing
Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
- Testing
- Understanding how classes collaborate
- Understanding what is still missing
Incremental work patterns

• Programming with limited knowledge
  – Generic protocols, absence of declarations
• Working in multiple contexts
  – Multiple windows, tiling browsers
• Refactoring
  – Refactoring Browser
• Testing
• Understanding how classes collaborate
• Understanding what is still missing
Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
  - SUnit
- Understanding how classes collaborate
- Understanding what is still missing
Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
  - SUnit
- Understanding how classes collaborate
  - ?
- Understanding what is still missing
Incremental work patterns

- Programming with limited knowledge
  - Generic protocols, absence of declarations
- Working in multiple contexts
  - Multiple windows, tiling browsers
- Refactoring
  - Refactoring Browser
- Testing
  - SUnit
- Understanding how classes collaborate
- Understanding what is still missing
How should we support Incremental Programming?

- Provide information about completeness of classes and collaborations between classes that is
  - statically computed,
  - always accessible, and
  - always up-to-date

- Why not? This information is in the code
  - My PowerBook is 50 times faster than a Dorado
How to present it?
How to present it?

• Virtual Categories
How to present it?

- Virtual Categories

```
ifTrue: alternativeBlock

"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error notification if the receiver is nonBoolean. Execution does not actually reach here because the expression is compiled in-line."

self subclassResponsibility
```
How to present it?

• Virtual Categories

categorization of methods by the browser, based on their characteristics; always up-to-date

```
ifTrue: alternativeBlock

"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error notification if the receiver is nonBoolean. Execution does not actually reach here because the expression is compiled in-line."

self subclassResponsibility
```
Four Categories:

- **requires**
  - all messages sent to this class for which there is no method defined or inherited

- **supplies**
  - all messages required by some other class for which methods are provided in this class

- **overrides**
  - methods defined in this class that override inherited methods

- **sending super**
  - methods that perform super sends
supplies & overrides

```
-own-

ifTrue: alternativeBlock

"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error notification if the receiver is nonBoolean. Execution does not actually reach here because the expression is compiled in-line."

self subclassResponsibility
```
**supplies & overrides**

```plaintext
<table>
<thead>
<tr>
<th>Kernel-Objects</th>
<th>MorphObjectOut</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-- all --</td>
</tr>
<tr>
<td></td>
<td>logical operations</td>
</tr>
<tr>
<td></td>
<td>controlling</td>
</tr>
<tr>
<td></td>
<td>printing</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td>ifFalse:</td>
</tr>
<tr>
<td></td>
<td>ifFalse:ifTrue:</td>
</tr>
<tr>
<td></td>
<td>ifTrue:</td>
</tr>
<tr>
<td></td>
<td>ifTrue:ifFalse:</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or:</td>
</tr>
<tr>
<td>&lt;inst.</td>
<td>?</td>
</tr>
</tbody>
</table>
```

```plaintext
ifTrue: alternativeBlock

"Answer the value of alternativeBlock. Execution does not actually reach here because the expression is compiled in-line."

^alternativeBlock value
```
supplies & overrides

```plaintext
printOn: aStream

aStream nextPutAll: 'true'
```
supplies & overrides

Object>> printOn: aStream
   "Append to the argument, aStream, a sequence of characters that identifies the receiver."
   ^ self printNameOn: aStream
Extended Example/Demonstration

• Creating a new sub-tree of classes

```
DemoList

printOn:
size

DemoEmptyList

head
rest
do:
isEmpty

DemoConsList

head
rest
do:
isEmpty
```
Understanding and Modifying Existing Hierarchies

- Feedback from the browser helps us find:
  - which methods are “core” and which are “support”
  - how the sub- and superclasses depend on each other

- When extending code, the browser helps avoid:
  - introducing inter-level errors
  - accidentally incomplete classes
The Core/Support Split

- A common pattern used to increase reuse in a data type implementation [Black ECOOP inh wk 2002]
  - e.g., the abstract superclass Collection defines 110 support methods
    - they don’t access the state of any collection directly
    - instead, they depend on 4 core methods
      - add:, atRandom:, do: and remove:ifAbsent:
        - 3 are defined as self subclassResponsibility
        - atRandom: is not defined at all
    - The browser finds these 4 required methods amongst the 110!
The Core/Support Split (2)

• When we look at a subclass of Collection, e.g., Bag, we can distinguish:
  ° the 4 supplied methods,
  ° 10 methods that override the inherited methods, either to disable them or to improve their efficiency, and
  ° 7 additional methods that widen the interface of Bag beyond that of Collection
Accidentally Abstract Classes

• The browser tells us some surprising things about Squeak’s core classes:
  ◦ **Fraction** is abstract
    • it implements the support method `printOn:`, whereas it should implement the core method `printOn:Base`:
  ◦ **Bitmap** is abstract
    • the programmer sends an error message `primitiveFail`, which he forgot to define
  ◦ **Debugger, CharacterSet, Morph** (and nearly all of its subclasses): all are abstract
Accidentally Abstract Classes (2)

• *Why* are these errors present in a code base that has been used by thousands of users for many years?
  
  ◦ It is *not* because a bad programmer wrote BitMap

• It is because even good programmers will make mistakes unless they have good tools
Accidentally Abstract Classes (2)

• *Why* are these errors present in a code base that has been used by thousands of users for many years?
  - It is *not* because a bad programmer wrote BitMap

• It is because even good programmers will make mistakes unless they have good tools
Implementation

- **sending super** is easy
  - look for the bytecode for *super sends*

- **overrides** is easy
  - compare this class’s selectors with its superclass’s protocol

- **supplies** is easy once one knows **requires**
  - compare this class’s selectors with the **requires** set of other classes

- **requires** is most definitely **not** easy
  - implementing **requires** in real-time required a lot of careful thinking and more careful programming!
What’s in the requires category?

- Pseudocode:

```markdown
Behavior >> requires

self reachableMethods selfMessages
difference:

(self allReallyImplementedSelectors)
```
What’s in the **requires** category?

- Pseudocode:

```
Behavior >> requires

self reachableMethods selfMessages

difference:

(self allReallyImplementedSelectors)
```

We had to invent and formalize a definition of reachability.
What’s in the **requires** category?

- Pseudocode:

  ```plaintext
  Behavior >> requires
  self reachableMethods selfMessages
difference:
  (self allReallyImplementedSelectors)
  ```
What’s in the **requires** category?

- **Pseudocode:**

  ```
  Behavior >> requires
  self reachableMethods selfMessages
  difference:
  (self allReallyImplementedSelectors)
  ```

  *Cannot infer this from bytecode.*
What’s in the **requires** category?

- Pseudocode:

```
Behavior >> requires

self reachableMethods selfMessages

difference:

  (self allReallyImplementedSelectors)
```
What’s in the **requires** category?

- Pseudocode:

  ```
  Behavior >> requires
  self reachableMethods selfMessages
  difference:
  (self allReallyImplementedSelectors)
  ```

  `allSelectors \ those not really implemented (subclass-
  Responsibility, shouldNotImplement, requirement, etc.)`
Recognizing self-sends

• Recognizing self-sends requires a full parse of the method text

• A change in, say, Object, might change the required methods of every class in the system!

• Squeak images contain > 60,000 methods

• We decided that we needed to cache the self-sends for every method when it is compiled
A problem of scale

- Even with these caches for self- and super sends, the first implementation took over 3 minutes to ascertain the required methods of a class!
Two key insights

• The caches should be arranged “backwards”
  ◦ for each *message*, cache the methods that self-send it

• We don’t need to know the requires set, all we need to know is whether it is empty
  ◦ Does a subclass override *all* of the methods that self-send a message required by the superclass?
    • if not, we *immediately* know that it is also required in the subclass
The Complete Algorithm...

- is far too complex to put on a slide
  - that’s what the paper is for!

- Computing the required set now takes less than 100 ms — fast enough to provide “real-time” feedback
The Complete Algorithm...

• is far too complex to put on a slide
  ◦ that’s what the paper is for!

• Computing the required set now takes less than 100 ms — fast enough to provide “real-time” feedback
Related Work

• To do lists
  ◦ Trellis’s “grass catcher” was also the product of changing a single method
  ◦ More commonly, as with Eclipse’s “Tasks” window, to do lists are updated only on global recompilation.

• Browser extensions
  ◦ decoration of names to indicate local properties such as overrides or sends to super, e.g., in VisualWorks
  ◦ Star Browser allows the definition of intentional classifications that are recomputed when necessary
Future Work

• Other visualizations of the *self-send* information
  ◦ *e.g.*, Blueprint-like diagrams

• Two directions for extension:
  ◦ Help in understanding other kinds of collaboration
    • *e.g.*, delegation, aggregation, Mudpie’s package dependencies
  ◦ A pluggable browser framework
    • what are the key features?
Conclusion

• The Browser is *Feasible*
  • with careful design and implementation, it *is* feasible to provide real-time feedback even for global properties such as *required* methods

• The Browser is *Useful*
  • Simplifies Intentional Programming
  • Makes it easier to understanding existing classes
  • Clarifies the relationship between sub- and superclasses
  • Exposes many bugs in existing code
Questions!
Which methods are reachable?
Which methods are reachable?

![Class diagram]

- **Object**
  - `printOn:
  - `...

- **Collection**
  - `printOn:
  - `printElementsOn:
  - `...

- **Bag**
  - `tally
  - `array
  - `printOn:
Which methods are reachable?

```
Object
  printOn:

... 
...

Collection
  printOn:
  printElementsOn:

Bag
  tally
  array
  printOn:
```
Which methods are reachable?

```ruby
printOn: s
  s nextPutAll: 'Bag'.
  super printOn: s.
```
Which methods are reachable?

```
Object
    printOn:

Collection
    printOn:
    printElementsOn:

Bag
    tally
    array
    printOn:

... printOn: ...

... printOn: ...

printOn: s
    s put: ${.
    s printElementsOn: s.
    s put: $}...

... printOn: ...
```

Which methods are reachable?
Which methods are reachable?

Object

Collection

Bag

...
Which methods are reachable?

Object
printOn:

... 

Collection
printOn:
printElementsOn:

Bag
tally
array
printOn:

printOn: s
s nextPutAll: 'Bag'.
super printOn: s.

printOn: s
s put: $.
s printElementsOn: s.
s put: $}...

printOn: s
s put: ${.
s printElementsOn: s.
s put: $}...
Which methods are reachable?

```
Object
    printOn:
        ...
        ...

Collection
    printOn:
    printElementsOn:

Bag
    tally
    array
    printOn:

printOn: s
    s nextPutAll: 'Bag'.
    super printOn: s.
```
Which methods are reachable?

```plaintext
printOn: s
  super printOn: s.
  s nextPut: ${.
  s printElementsOn: s.
  s nextPut: $}…

printOn: s
  s nextPutAll: 'Bag'.
  super printOn: s.
```

```
Object
  printOn:
  ...
  ...

Collection
  printOn:
  printElementsOn:
  ...

Bag
  tally
  array
  printOn:
```
Which methods are reachable?

```
... printOn: s
... self className
...
...

Collection

printOn: s
printElementsOn:
...
...

Bag

tally
array
printOn:

printOn: s
super printOn: s.
s nextPut: ${.
s printElementsOn: s.
s nextPut: ${}...

printOn: s
s nextPutAll: 'Bag'.
super printOn: s.
```
Which messages are self-sent?

```plaintext
fastenVerySecurely

| temp |

self hook.

temp := self

temp button.

self class new clipTo: self
```
Which messages are self-sent?

```smalltalk
fastenVerySecurely

| temp |

self hook.

temp := self
temp button.

self class new clipTo: self
```

This is a self-send, and we recognize it.
Which messages are self-sent?

```smalltalk
fastenVerySecurely
| temp |
self hook.
temp := self
temp button.
self class new clipTo: self
```
Which messages are self-sent?

`fastenVerySecurely`

```
| temp |
self hook.
temp := self
```

`temp button.`

```
self class new clipTo: self
```

This is a self-send, but we don’t recognize it
Which messages are self-sent?

```smalltalk
fastenVerySecurely
| temp |
self hook.
temp := self
```  

This is a self-send, but we don’t recognize it