Better Code: Human Interface
Sean Parent | Principal Scientist
Topics for Discussion
Better Code: Design and Ethics

Sean Parent | Principal Scientist
Better Code: Futures are not Monads

Sean Parent | Principal Scientist
Better Code: Futures are not Monads
Sean Parent | Principal Scientist
Faster Bresenham’s Algorithm

Sean Parent | Principal Scientist
Old Guy Reminiscing
Sean Parent | Principal Scientist
Today's Talk
Relationship Between HI and Code

“The purpose of a human interface is not to hide what the code does but to accurately convey what the code does.” – Darin Adler (personal conversation, best of my recollection)
Goal: Don't Lie
Taxonomy of Everything
Taxonomy of Everything

- Objects
Taxonomy of Everything

- Objects
- Properties
Taxonomy of Everything

- Collections
- Objects
  - Properties
Taxonomy of Everything

▪ Collections
▪ Objects
  ▪ Properties
▪ Operations
Taxonomy of Everything

- Collections
- Objects
  - Properties
- Operations
- Relationships
Model-View-Controller
How did MVC get so F’ed up?
How did MVC get so F’ed up?

Observable Models

- Application model is Objects + Operations + Relationships
Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
  - Trivial controller binds to set property
Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
  - Trivial controller binds to *set property*
- View bind to objects and properties
Observable Models

- Application model is Objects + Operations + Relationships
- Controllers bind to operations
  - Trivial controller binds to set property
- View bind to objects and properties

- A view/controller is a control or widget
Objects

▪ Operations
  ▪ Construct
  ▪ Copy
  ▪ Move
  ▪ Delete

▪ Properties
  ▪ Location
  ▪ Size
  ▪ Name (common)
Objects

- Operations
  - Construct
  - Copy
  - Move
  - Delete

- Properties
  - Location
  - Size
  - Name (common)
Objects

- We associate visual constructs, names, icons, and behaviors with semantics
- In programs operations like *construct* have specific semantics
- In the HI we associate semantics with controls
<table>
<thead>
<tr>
<th>Starred</th>
<th>Subject</th>
<th>Body</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DreamHost</td>
<td>Introducing the New DreamHost Newsletter</td>
<td>12:37 pm</td>
</tr>
<tr>
<td></td>
<td>Starbucks Rewards</td>
<td>That one afternoon snack</td>
<td>11:49 am</td>
</tr>
<tr>
<td></td>
<td>Visual Studio Subscription</td>
<td>New benefits and updates for October</td>
<td>11:33 am</td>
</tr>
<tr>
<td></td>
<td>Erik's DeliCafé</td>
<td>Unleash the Feast with these new sandwiches 🍔</td>
<td>11:15 am</td>
</tr>
<tr>
<td></td>
<td>ahstore.com</td>
<td>Spooky Savings at ahstore.com</td>
<td>11:01 am</td>
</tr>
<tr>
<td></td>
<td>Nextdoor Spring</td>
<td>City of Morgan Hill Weekly 411, 10.30.17</td>
<td>10:38 am</td>
</tr>
</tbody>
</table>
Collections

- Operations
  - Insert
  - Remove

- Properties
  - Count

- Relationships
  - Whole/Part
Collections

- Large collections pose a problem
- How to observe the collection interactively, allowing the user to arrange, filter, and browse
Observing Collections

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>13</td>
<td>12</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>14</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>10</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Observing Collections

<table>
<thead>
<tr>
<th>1</th>
<th>sf -</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>sl -</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Observing Collections

```
x
x
x
x
x
x
x
x
 x
x
x
x
x
x
x
```

sf -

6
7
8
9

sl -

x
x
x
x
x
x
x
x
x
Observing Collections

A diagram showing a collection of numbers from 1 to 16, with arrows labeled 'f-', 'sf-', 'sl-', and 'l-' pointing to specific numbers.
Observing Collections

```
4
13
12
7
9
5
15
14
2
11
6
16
10
1
8
3
```

- f -
- sf -
- l -
nth_element(f, sf, l);
Observing Collections

nth_element(f, sf, l);
nth_element(f, sf, l);
nth_element(f, sf, l);
Observing Collections

```c
nth_element(f, sf, l);
```

```
| 2 | 1 | 3 | 4 | 5 | 6 | 7 | 14 | 12 | 15 | 9 | 16 | 10 | 13 | 8 | 11 |
```

- `f` is less than or equal to `sf` and greater than or equal to `l`.
nth_element(f, sf, l);
nth_element(f, sf, l);
++sf;
Observing Collections

```
nth_element(f, sf, l);
++sf;
```
nth_element(f, sf, l);
++sf;

partial_sort(sf, sl, l);
Observing Collections

\[
\begin{align*}
&f \leftarrow \begin{array}{c}
2 \\
1 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
15 \\
14 \\
16 \\
12 \\
13 \\
10 \\
11
\end{array} \\
&sf \leftarrow \begin{array}{c}
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf} \\
\text{sf}
\end{array} \\
&sl \leftarrow \begin{array}{c}
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl} \\
\text{sl}
\end{array}
\end{align*}
\]

\begin{align*}
&\text{nth\_element}(f, sf, l); \\
&\quad ++sf; \\
&\text{partial\_sort}(sf, sl, l);
\end{align*}
nth_element(f, sf, l);
++sf;
partial_sort(sf, sl, l);
if (sf == sl) return;

nth_element(f, sf, l);
++sf;

partial_sort(sf, sl, l);
if (sf == sl) return;
if (sf != f) {
    nth_element(f, sf, l);
    ++sf;
}  
partial_sort(sf, sl, l);
template <typename I> // I models RandomAccessIterator
void sort_subrange(I f, I l, I sf, I sl)
{
    if (sf == sl) return;
    if (sf != f) {
        nth_element(f, sf, l);
        ++sf;
    }
    partial_sort(sf, sl, l);
}
Observing Collections

```c
sort_subrange(f, l, sf, sl);
```
Observing Collections

```c
sort_subrange(f, l, sf, sl);
```
Observing Collections

```
sort_subrange(f, l, sf, sl);
```
Observing Collections

```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```
Observing Collections

\[
\text{sort\_ subrange}(f, l, sf, sl);
\]
\[
\text{partial\_ sort}(sl, nl, l);
\]
Operations

- Operations act on one or more objects
  - Additional arguments to the operation are bound as properties
  - Operations are represented by buttons, menu items, gestures, tools, direct manipulation
- Subject or target of an operation is identified by
  - Selections
  - Direct Manipulation
Selections

- Selecting objects within the hierarchy specifies one or more target paths
  - Application->Document->Object
Selections

- Selecting objects within the hierarchy specifies one or more target paths
- Application->Document->Object
Selections

- Interval sets are a good data structure to represent selections
Gather
Gather
Gather
Gather

```c
stable_partition(p, l, s)
```
Gather

stable_partition(p, l, s)
stable_partition(f, p, not1(s))
Gather

\[\text{stable\_partition}(f, p, \text{not1}(s))\]
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
Gather

\[
\begin{align*}
\text{stable_partition}(f, p, \text{not1}(s)) \\
\text{stable_partition}(p, l, s)
\end{align*}
\]
Gather

stable_partition(f, p, not1(s))
stable_partition(p, l, s)
Gather

return { stable_partition(f, p, not1(s)),
stable_partition(p, l, s) };
Gather

template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
Gather

template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I> {
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
tauto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
Stable Partition
Stable Partition
Stable Partition
Stable Partition
Stable Partition
Stable Partition

\[
\text{stable\_partition}(f, m, p) \\
\text{stable\_partition}(m, l, p)
\]
Stable Partition

stable_partition(f, m, p)
stable_partition(m, l, p)
Stable Partition

```
stable_partition(f, m, p)
stable_partition(m, l, p)
```
rotate(stable_partition(f, m, p),
    m,
    stable_partition(m, l, p));
Stable Partition

rotate(stable_partition(f, m, p),
    m,
    stable_partition(m, l, p));
Stable Partition

\[
\text{return } \text{rotate(stable\_partition}(f, m, p), m, \text{stable\_partition}(m, l, p));
\]
return rotate(stable_partition(f, m, p),
    m,
    stable_partition(m, l, p));
Stable Partition

if (n == 1) return f + p(*f);

return rotate(stable_partition(f, m, p),
               m,
               stable_partition(m, l, p));
Stable Partition

\[
\begin{align*}
    \text{if } (n == 1) & \text{ return } f + p(*f); \\
    \text{return } & \text{rotate(stable_partition(f, m, p),} \\
    & \text{m,} \\
    & \text{stable_partition(m, l, p));}
\end{align*}
\]
template <typename I, 
    typename P>
auto stable_partition(I f, I l, P p) -> I 
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);
    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p), 
               m, 
               stable_partition(m, l, p));
}
Stable Partition

```cpp
template <typename I, 
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```
Stable Partition

template <typename I,  
    typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);
    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),  
                   m,  
                    stable_partition(m, l, p));
}
Stable Partition

```cpp
template <typename I, typename P>
auto stable_partition(I f, I l, P p) -> I {
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);
    auto m = f + (n / 2);
    return rotate(stable_partition(f, m, p), m,
                   stable_partition(m, l, p));
}
```
template <typename I, 
  typename P>
auto stable_partition_position(I f, I l, P p) -> I 
{
  auto n = l - f;
  if (n == 0) return f;
  if (n == 1) return f + p(f);
  auto m = f + (n / 2);

  return rotate(stable_partition_position(f, m, p),
               m,
               stable_partition_position(m, l, p));
}
Using gather_position

```cpp
interval_set<I> selection;

//...

gather_position(f, l, p, [&](auto p) {
    return contains(selection, p);
});
```
Selections

- Multi-select is only sporadically implemented
- Always inconsistently
One Way to Select Many*

Jaakko Järvi1 and Sean Parent2

1 Texas A&M University
College Station, TX, USA
jarvi@cse.tamu.edu

2 Adobe Systems Inc.
San Jose, CA, USA
sparent@adobe.com

Abstract
Selecting items from a collection is one of the most common tasks users perform with graphical user interfaces. Practically every application supports this task with a selection feature different from that of any other application. Defects are common, especially in manipulating selections of non-adjacent elements, and flexible selection features are often missing when they would clearly be useful. As a consequence, user effort is wasted. The loss of productivity is experienced in small doses, but all computer users are impacted. The undesirable state of support for multi-element selection prevails because the same selection features are redesigned and reimplemented repeatedly. This article seeks to establish common abstractions for multi-selection. It gives generic but precise meanings to selection operations and makes multi-selection reusable; a JavaScript implementation is described. Application vendors benefit because of reduced development effort. Users benefit because correct and consistent multi-selection becomes available in more contexts.

1998 ACM Subject Classification
D.2.11 Software Architectures: Domain-specific architectures;
D.2.13 Reusable Software: Reusable libraries

Keywords and phrases User interfaces, Multi-selection, JavaScript

Introduction
Many, perhaps most, interactive software applications present their users one or more collections of elements in the form of lists, trees, grids, or otherwise arranged views, of which a user can select one or more elements. Examples include selecting files and folders in a file explorer; mail folders or mail messages in a mail client; music tracks in a media player; thumbnail images in a photograph organizer; “to do” list items, hours, days, weeks, or months in a calendar application; pages organized into “tabs” in a web browser; and electronic books or videos on a digital library or store. These tasks are typical daily activities for many computer users—we select elements from collections dozens of times per day.

Regardless of which set of modern applications a user chooses for mail, music, photos, calendar, web browsing, books, and videos, the features for selecting elements are likely to differ across applications—even within a single application the selection features for different collections, such as the list of mail folders and list of mail messages, are likely to be different. The differences could presumably stem from optimizing the feature for the best possible user experience in different kinds of selection contexts, but this is not the case. The selection

---

* This work was supported in part by NSF grants CCF-0845861 and CCF-1320092.

© Jaakko Järvi and Sean Parent; licensed under Creative Commons License CC-BY

First Conference on Very Important Topics (CVIT 2016), Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:25

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Relationships

- A relationship is the way two entities are connected
- Connective tissue between objects and properties
- A structure is formed by connected relationships
- Architecture is the art and science of designed structures
Relationships

- A *relationship* is the way two entities are connected
- Connective tissue between objects and properties
- A *structure* is formed by connected relationships
- *Architecture* is the art and science of designed structures

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation
Relationships

- A relationship is the way two entities are connected
- Connective tissue between objects and properties
- A structure is formed by connected relationships
- Architecture is the art and science of designed structures

- A relationship implies a corresponding predicate that tests if a pair exists in the relation

- Within an HI relationships can be challenging to represent
Relationships
## Relationships

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
</table>
## Relationships

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_showers &gt;= 1</td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Design</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>num_showers &gt;= 1</td>
<td>num_showers = 1;</td>
</tr>
<tr>
<td>Requirement</td>
<td>Design</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><code>num_showers &gt;= 1</code></td>
<td><code>num_showers = 1;</code></td>
</tr>
</tbody>
</table>
### Relationships

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>num_showers &gt;= 1</code></td>
<td><code>num_showers = 1;</code></td>
</tr>
<tr>
<td><code>num_toilets &gt;= 1</code></td>
<td></td>
</tr>
</tbody>
</table>
## Relationships

<table>
<thead>
<tr>
<th><strong>Requirement</strong></th>
<th><strong>Design</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>num_showers $\geq 1$</td>
<td>num_showers = 1;</td>
</tr>
<tr>
<td>num_toilets $\geq 1$</td>
<td>num_toilets = 1;</td>
</tr>
</tbody>
</table>
### Relationships

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>num_showers &gt;= 1</code></td>
<td><code>num_showers = 1;</code></td>
</tr>
<tr>
<td><code>num_toilets &gt;= 1</code></td>
<td><code>num_toilets = 1;</code></td>
</tr>
<tr>
<td>Requirement</td>
<td>Design</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>num_showers &gt;= 1</td>
<td>num_showers = 1;</td>
</tr>
<tr>
<td>num_toilets &gt;= 1</td>
<td>num_toilets = 1;</td>
</tr>
<tr>
<td>privacy &gt;= 0</td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Design</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>num_showers $\geq 1$</td>
<td>num_showers = 1;</td>
</tr>
<tr>
<td>num_toilets $\geq 1$</td>
<td>num_toilets = 1;</td>
</tr>
<tr>
<td>privacy $\geq 0$</td>
<td>privacy = 0;</td>
</tr>
</tbody>
</table>
## Relationships

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>num_showers &gt;= 1</code></td>
<td><code>num_showers = 1;</code></td>
</tr>
<tr>
<td><code>num_toilets &gt;= 1</code></td>
<td><code>num_toilets = 1;</code></td>
</tr>
<tr>
<td><code>privacy &gt;= 0</code></td>
<td><code>privacy = 0;</code></td>
</tr>
</tbody>
</table>
“Simple” Relationship

\[ a \implies b \]

(\( a \) implies \( b \))
Implies (examples from the clang manual)

- "-ggdb, -glldb, -gsce ... Each of these options **implies** -g."
- "-f[no-]diagnostics-show-hotness ... This option is **implied** when -fsave-optimization-record is used."
- "-M, --dependencies ... Like -MD, but also **implies** -E"
- "-MM, --user-dependencies ... Like -MMD, but also **implies** -E"
- "-cl-unsafe-math-optimizations ... Also **implies** -cl-no-signed-zeros and -cl-mad-enable."
void operation(bool a, bool b) {
    b = a || b;  // a implies b
    //...
}
Unconstrained

```c
void operation(bool a, bool b) {
    b = a || b; // a implies b
    //...
}
```
- (IBAction)aChanged {
    if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
First Attempt

- (IBAction)aChanged {
  if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
  assert(!a || b); // a implies b
  //...
}
Goal

Use strong preconditions
Goal

Use strong preconditions and assert them
- (IBAction)aChanged {
    _bSwitch.enabled = !_aSwitch.on;
    if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
Disable

- (IBAction)aChanged {
  _bSwitch.enabled = !_aSwitch.on;
  if (_aSwitch.on) _bSwitch.on = true;
}

void operation(bool a, bool b) {
  assert(!a || b); // a implies b
  //...
}
Disable + Memory

- (IBAction)aChanged {
  _bSwitch.enabled = !_aSwitch.on;
  _bSwitch.on = _aSwitch.on || _b;
}

- (IBAction)bChanged {
  _b = _bSwitch.on;
}

void operation(bool a, bool b) {
  assert(!a || b); // a implies b
  //...
}
Disable + Memory

- (IBAction)aChanged {
  _bSwitch.enabled = !_aSwitch.on;
  _bSwitch.on = _aSwitch.on || _b;
}

- (IBAction)bChanged {
  _b = _bSwitch.on;
}

void operation(bool a, bool b) {
  assert(!a || b); // a implies b
  //...
}
Contrapositive + Memory

- (IBAction)aChanged {
  _a = _aSwitch.on;
  _bSwitch.on = _a || _b;
}

- (IBAction)bChanged {
  _b = _bSwitch.on;
  _aSwitch.on = _b && _a;
}

void operation(bool a, bool b) {
  assert(!a || b);  // a implies b
  //...
}
Contrapositive + Memory

- (IBAction)aChanged {
   _a = _aSwitch.on;
   _bSwitch.on = _a || _b;
}

- (IBAction)bChanged {
   _b = _bSwitch.on;
   _aSwitch.on = _b && _a;
}

void operation(bool a, bool b) {
   assert(!a || b); // a implies b
   //...
}
Contraposition

- (IBAction)aChanged {
   _bSwitch.on = _aSwitch.on || _bSwitch.on
}

- (IBAction)bChanged {
   _aSwitch.on = _bSwitch.on && _aSwitch.on
}

void operation(bool a, bool b) {
   assert(!a || b); // a implies b
   //...
}
Contrapositive

- `(IBAction)aChanged { 
  _bSwitch.on = _aSwitch.on || _bSwitch.on 
}

- `(IBAction)bChanged { 
  _aSwitch.on = _bSwitch.on && _aSwitch.on 
}

void operation(bool a, bool b) {
  assert(!a || b); // a implies b
  //...
}
Unconstrained + Disable Operation

- (IBAction)changed {
  _operation.enabled =
  !_aSwitch.on || !_bSwitch.on;
}

void operation(bool a, bool b) {
  assert(!a || b); // a implies b
  //...
}
void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
A hidden layer, a normal layer & a hidden group are all layers on a timeline. What happens if the layers are all hidden, or if the group is hidden? Do they still have a timeline? Will the timeline have a play button?
What is a good design?

- Toggling a control should restore system to original state
- Result of a click should be predictable without knowing how current state was achieved
- Guided paths are preferred so long as they don't make navigation more difficult

- But there needs to be additional rules to handle conflicts
- Rules derived from
  - Convention
  - Experience
  - Studies
Property Models

- Unconstrained
  ```java
  a; b;
  ```

- Disabled
  ```java
  b <= a || b;
  ```

- Disabled + Memory
  ```java
  unlink b <= a || b;
  ```

- Contrapositive + Memory
  ```java
  unlink a; unlink b;
  relate {
    b <= a || b;
    a <= b && a;
  }
  ```

- Contrapositive
  ```java
  relate {
    b <= a || b;
    a <= b && a;
  }
  ```

- Unconstrain + Disable Operation
  ```java
  invariant:
  valid <= !a || b;
  ```
Generating Reactive Programs for Graphical User Interfaces from Multi-way Dataflow Constraint Systems

Gabriel Foust
Texas A&M University, USA
gfoust@cs.tamu.edu

Jaakko Jarvi
Texas A&M University, USA
jarvi@cs.tamu.edu

Sean paren
Adobe Systems, Inc.
sparen@adobe.com

Helping Programmers Help Users

Mat Marcus
Canyonlands Software Design
mmarcus@canyonlands.org

Jaakko Jarvi
Texas A&M University
jarvi@cs.tamu.edu

Wonseok Kim
Texas A&M University
gatora@cs.tamu.edu

General Terms
General Terms
Design, Theory

Categories and Subject Descriptors
C.2.2 [Software Engineering]: Software Architectures—Domain-specific architectures

Abstract
User interfaces exhibit a wide range of features that are designed to assist users. Interaction with one widget may trigger value changes, disabling, or other behaviors in other widgets. Such automatic behavior may be confusing or disruptive to users. Research literature on user interfaces offers a number of solutions, including interface features for explaining or controlling these behaviors. To help pro-
grammers anticipate these changes, this paper proposes a declarative approach to provide the user with a clearly interpreted view of conceptual models for the internal states of the application and the interface itself. To the extent that the interface fails to do this, there exists a gap of evaluation [7]. The goal of evaluation magnifies the cognitive effort required to understand and use an application, and can lead to user frustration.

This paper shows that with the power of components, generality, and reuse we can go beyond merely implementing existing behavior more economically. If a user interface can be successfully packaged into a reusable component, then we should explore more functionality for assisting users and closing the gap of evaluation. We should aim for more consistent user interfaces with less surprising behavior, even when a user interfaces with applications with widely varying user interfaces.

Keywords
Chatbot interfaces, software reuse, constraint systems, software architecture

1. Introduction

User interfaces exhibit a wide range of features that are designed to assist users. Interaction with one widget may trigger value changes, disabling, or other behaviors in other widgets. Such automatic behavior may be confusing or disruptive to users. Research literature on user interfaces offers a number of solutions, including interface features for explaining or controlling these behaviors. To help programmers anticipate these changes, this paper proposes a declarative approach to provide the user with a clearly interpreted view of conceptual models for the internal states of the application and the interface itself. To the extent that the interface fails to do this, there exists a gap of evaluation [7]. The goal of evaluation magnifies the cognitive effort required to understand and use an application, and can lead to user frustration. To not overstate our ambition, we are at an early stage in our effort. To not overstate our ambition, we are at an early stage in our effort.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2017 ACM. 978-1-4503-4368-7/17/10...$15.00

Figure 1: An example of an auto-complete text box, and a diagram showing the data dependencies involved in its implementation.

Asynchronous execution is complicated by data dependencies between tasks. Such dependencies mean that the execution of one task may affect the outcome of another; therefore running tasks on different or in parallel may yield different outcomes. The programmer must carefully guard against execution schedules that could produce incorrect results. This is not easy in the event-driven GUI programming paradigm, where data dependencies implicitly arise whenever users interact with the user interface.

By way of illustration, we examine one common GUI element: the auto-complete text box. This element helps the user produce a string to be used as input by some part of the application. Text entered by the user becomes the input string, but is also used as a parameter in an asynchronous search for related input strings. Typically the search results are organized into a menu, and displayed in a text box as a menu from which the user, with a mouse or keyboard, may select an alternate input string. Figure 1a shows an auto-complete text box being used to select a city as a travel destination.

Figure 1b shows the dependencies that emerge in this seemingly simple GUI element. Text entered by the user becomes the query parameter, which determines the menu items. If a menu item is selected, the index of the selected item and the contents of the menu determine the input string; if no item is selected, the query parameter itself becomes the input string. Typically, when the contents of the menu changes, the selected index is reset. We show this dependency with a dashed line, as it is only in effect when the menu changes.

We claim these dependencies are non-trivial, and that writing code that enforces them is difficult using the traditional event-driven programming paradigm. To investigate this claim, we performed an informal survey of six popular commercial travel sites (expedia, orbitz, travelocity, bing, google, and tripadvisor) and found that all six contained auto-complete text boxes exhibiting inconsistent behavior. We describe the auto-complete text boxes and the full context of their behavior on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Abstract

User interfaces for modern applications must support a rich set of interactive features. It is commonplace to find applications with dependencies between values manipulated by user interface elements, conditionally enabled controls, and script-record and playback against different documents. A significant fraction of the application programming effort is devoted to implementing such functionality, and the resulting code is typically not reusable.

This paper introduces "property models," an approach to programming user interfaces. Property models allow a large part of the functionality of a user interface to be implemented in reusable libraries, reducing application specific code to a set of declarative rules. We describe how, as a by-product of computations that maintain the values of user interface elements, property models obtain accurate information of the currently active dependences among those elements. This information enables further expanding the class of user interface functionality that we can encode as generic algorithms.

In particular, we describe automatic generation of the mechanisms of user interface widgets and activation of command widgets. Failures due to increased time, increased cost, reduced defects rate, and improved user interface design turnarounds in a commercial software development process. We report on the increased reuse, reduced defect rates, and improved quality of the code and notably more of the defects in desktop applications. As the scale of software increases, software development requires more of reusable components—at the same time, there is an increase in the amount of software libraries and relations. Often such code is not explicitly designed, and it is rarely reusable in large collections of components, network relationships between components. We introduce a new implementation mechanism for defining and maintaining instances of typical user interfaces, such as dialog windows, that are composed of many components, conditions and constraints. We describe how our implementation mechanism can help in creating reusable libraries for common user interface building blocks. We provide a new implementation mechanism for defining and maintaining instances of typical user interfaces, such as dialog windows, that are composed of many components, conditions and constraints. We describe how our implementation mechanism can help in creating reusable libraries for common user interface building blocks.

1. Introduction

User interface, such as a dialog, assists a user in synthesizing a set of values, typically parameters for a command object. Code for "command parameter synthesis" is usually application-specific and non-reusable, consisting of code for the user interface and code that controls how values of user interface elements change in response to a user's actions. These software artifacts are incidental—they are not explicitly designed and their implementation emerges from a combination of locally defined behaviors. This article presents property models to capture explicitly the algorithms, validation, and interaction rules, arising from command parameter synthesis. User interface's behavior can be derived from a declarative property model specification, with the assistance of a constraint solver. This allows multiple user interfaces, both human and programmatic, to reuse a single model along with associated validation logic and widget activation logic. The proposed technology is deployed in large commercial software applications.

We have previously introduced a property model [13], an approach to explicitly model the behavior of a class of typical user interfaces, such as dialog windows. We have shown how a constraint logic for computing new values of user interface elements after changing values of other elements and an algorithm for recording and playback can be reused across user interfaces. These algorithms are generic, parameterized by a declaratively specified model for the user interface and the functional dependences among those variables. We suggested that where property models can be applied, the amount of code is notably reduced and software quality improves compared to using a traditional UI framework.

In this paper we describe some of the architectural challenges in creating reusable libraries for user interfaces. We identify the communication and relationships between different elements of user interfaces as an architectural domain where accidental data structures and algorithms are prevalent; we refer to this domain as command parameter synthesis. We introduce an accurate, reusable representation of components and have neither an explicit encoding in the component model. There is potential for wide adoption: by our measure, 90% of the common command parameter synthesis assists a client in selecting and validating parameters for some command to be executed in the program. This is an important task in interacting with an application with a non-trivial, human or programmatic interface. Typical examples of user interfaces requiring command parameter...
Closing

- Good code is necessary, but not sufficient, for building a good UI.
- There is significant work in the area of data structures, algorithms to support good UI
- And significant work remaining
References

MAKE IT AN EXPERIENCE