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

just
Faster Bresenham’s Algorithm
Sean Parent | Principal Scientist
Today's Talk
Better Code: Human Interface
Sean Parent | Principal Scientist
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
“MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input.”

– Design Patterns: Elements of Reusable Object-Oriented Software, section 1.2
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*
- Views bind to objects and their properties
Observable Models

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

<table>
<thead>
<tr>
<th>Object</th>
<th>Subject</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DreamHost</td>
<td>Introducing the New DreamHost Newsletter</td>
<td>12:37 pm</td>
</tr>
<tr>
<td>Starbucks Rewards</td>
<td>That one afternoon snack</td>
<td>11:49 am</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>New benefits and updates for October</td>
<td>11:33 am</td>
</tr>
<tr>
<td>Erik's DeliCafé</td>
<td>Unleash the Feast with these new sandwiches</td>
<td>11:15 am</td>
</tr>
<tr>
<td>ahstore.com</td>
<td>Spooky Savings at ahstore.com</td>
<td>11:01 am</td>
</tr>
<tr>
<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
Observing Collections

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>sf-</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>sl-</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Observing Collections
### Observing Collections

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>15</td>
<td>14</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

- `f -`
- `sf -`
- `l -`
Observing Collections

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

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

nth_element(f, sf, l);
Observing Collections

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ f \leq *sf \]

\[ sf \geq *sf \]
Observing Collections

\[ \text{nth_element}(f, sf, l); \]
nth_element(f, sf, l);
++sf;
nth_element(f, sf, l);
++sf;
Observing Collections

```cpp
nth_element(f, sf, l);
++sf;
partial_sort(sf, sl, l);
```
nth_element(f, sf, l);
++sf;
partial_sort(sf, sl, l);
Observing Collections

```
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

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

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

```
sort subrange(f, l, sf, sl);
partial sort(sl, nl, l);
```
Observing Collections

```
sort_subrange(f, l, sf, sl);
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
Gather

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

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

\[
\text{stable}\_\text{partition}(f, p, \text{not1}(s))
\]
Gather

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

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

\[
\text{stable\_partition}(f, p, \text{not\_1}(s))
\]
\[
\text{stable\_partition}(p, l, s)
\]
Gather

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

```cpp
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
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) };
}
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

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

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

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

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

\[
\text{if} \ (n == 1) \ \text{return} \ f + p(*f);
\]

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

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

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

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));
}
```
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

```c++
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 repeat-
edly. 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

1 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

Conferences 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

Selections
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>

© 2017 Adobe Systems Incorporated. All Rights Reserved.
## Relationships

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

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

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

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design</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>
<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>num_showers (\geq 1)</td>
<td>(\text{num_showers} = 1;)</td>
</tr>
<tr>
<td>num_toilets (\geq 1)</td>
<td>(\text{num_toilets} = 1;)</td>
</tr>
<tr>
<td>privacy (\geq 0)</td>
<td>(\text{privacy} = 0;)</td>
</tr>
</tbody>
</table>
“Simple” Relationship

\[ a \implies b \]

\((a \text{ implies } b)\)
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.”
Unconstrained

```c
void operation(bool a, bool b) {
    b = a || b; // a implies b
    //...
}
```
void operation(bool a, bool b) {
    b = 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
    //...
}
First Attempt

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

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

Use strong preconditions
Goal

Use strong preconditions and assert them
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
  //...
}
- (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
Contrapositive

\( \neg b \implies \neg a \)
Contra-positive

\(\neg b \Rightarrow \neg a\)

(not b implies not a)
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
  //...
}
void operation(bool a, bool b) {
    assert(!a || b); // a implies b
    //...
}
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
    //...
}
Unconstrained + Disable Operation

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

void operation(bool a, bool b) {
  assert(!a || b);  // a implies b
  //...
}
A hidden layer, a normal layer, and a hidden group. If an object moves from the normal layer to the hidden group, what happens to the parent of an object layer if an object in the hidden group is selected?
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
  \[ a; b; \]

- Disabled
  \[ b <= a || b; \]

- Disabled + Memory
  \[ \text{unlink } b <= a || b; \]

- Contrapositive + Memory
  \[ \text{unlink } a; \text{unlink } b; \text{relate } \{ \]
  \[ b <= a || b; \]
  \[ a <= b && a; \]
  \[ \} \]

- Contrapositive
  \[ \text{relate } \{ \]
  \[ b <= a || b; \]
  \[ a <= b && a; \]
  \[ \} \]

- Unconstrained + Disable Operation
  \[ \text{invariant: } \]
  \[ \text{valid } <= !a || b; \]
For a GUI to remain responsive, it must be able to schedule lengthy asynchronous computations to enforce those dependencies. In essence, a property model dynamically generates a reactive program, adding to it as new events occur. The approach gives programmers help and control mechanisms for explaining or controlling these behaviors. To help programmers help users, the implementation costs of these features should be much lower. Ideally, they could be generated for “free.” This paper shows how several help and control mechanisms can be implemented as algorithms and reused across interfaces, making the cost of their adoption negligible. Specifically, we describe generic help mechanisms for visualizing data flow and explaining deactivation of widgets, and a mechanism for controlling the flow of data. A reusable implementation of these features is enabled by our property model framework, where the data manipulated through a user interface is modeled as a constraint system. Reusable user interface manipulations and the dependencies within this data are modeled explicitly as algorithms or components in a software library, parametrized by a specification of the data manipulated by the user interface. In particular, we have described reusable code for the propagation of values between user interface elements, the establishment and disablement of user interface widgets, and the activation and deactivation of widgets that launch commands.

This paper describes our work to direct these advances to the improvement of existing interfaces. Typically the search results are listed below the text box as a menu of cities, a declarative approach to programming user interfaces, asynchronous programming as the same sequence of editing operations producing different output. In this case, the widget is used to select a city as a travel destination. Figure 1b shows the dependencies that emerge in this seemingly simple GUI element. Test-driven by the user because the query parameter, which determines the menu items. If a menu item is selected, the index of the selected item and points of the menu determine the input field; if no item is selected, the query parameter itself becomes the input string. Finally, a change in the contents of the menu affects the selected index; if the previously selected city is in the new menu, its new index should be used; otherwise the index should be reset. We show this dependency with a dashed line, indicating that the query parameter is not used.

This paper shows that with the power of components, generalizations, and reuse we can go beyond merely implementing existing behavior more economically. If a user interface behavior can be successfully packaged into a reusable component, then we should explore more functionality for assisting users and closing the gulf of evaluation. We should aim for more consistent user interfaces with surprising behavior, not merely functional. If a user interface behaves the way it does, and more abilities to change the behavior of a user interface “on the fly” to better serve users’ goals. In sum, we should aim for more features that help users in their interactions with an interface.

This paper describes several generic realizations of help and control mechanisms that could be provided as standard features of idioms and forms. In particular, we focus on (1) visualizing how data flows in a user interface, (2) providing help messages for commands that activate new features, (3) providing the user with methods to control the direction of the flow of data. We emphasize that the main contributions of the paper are the algorithms and the software architecture that enable implementing these features in a reusable manner. Application domains might also be interested in the more specific techniques in the paper. The algorithms exploit the flexibility of component models and the techniques are also applicable to large, persistent, and complex user interfaces.

We are at an early stage in our effort. To not overstate our contribution, we note that we have not conducted user studies, and we have not applied the proposed tools and algorithms to a large collection of user interfaces drawn from existing software. The computer-human interaction (CHI) research community, however, has devised many help and support features for user interfaces and algorithms or components in a software library, parametrized by a specification of the data manipulated by the user interface.

Interface behaves the way it does, and more abilities to change the behavior of a user interface “on the fly” to better serve users’ goals. In sum, we should aim for more features that help users in their interactions with an interface.

This paper describes several generic realizations of help and control mechanisms that could be provided as standard features of idioms and forms. In particular, we focus on (1) visualizing how data flows in a user interface, (2) providing help messages for commands that activate new features, (3) providing the user with methods to control the direction of the flow of data. We emphasize that the main contributions of the paper are the algorithms and the software architecture that enable implementing these features in a reusable manner. Application domains might also be interested in the more specific techniques in the paper. The algorithms exploit the flexibility of component models and the techniques are also applicable to large, persistent, and complex user interfaces.

We are at an early stage in our effort. To not overstate our contribution, we note that we have not conducted user studies, and we have not applied the proposed tools and algorithms to a large collection of user interfaces drawn from existing software. The computer-human interaction (CHI) research community, however, has devised many help and support features for user interfaces and algorithms or components in a software library, parametrized by a specification of the data manipulated by the user interface.
Abstract

We report on the increased reuse, reduced defect rates, and im-
proved interferface quality that we have experienced through the
use of declarative property models. We show that while property
models can capture explicitly the conditional interactions of
user interface code, they also allow the code to be generated in
separate, reusable modules. These modules can then be applied
and integrated with many other applications, providing more uni-
form access to functionality. Where we have applied property
models, we have measured significant reductions in source-code size
with equivalent or increased functionality in the resulting
user interfaces. This is an important result for applications that
require code to be reused, and can be reused, across many different
applications. We believe that the increased reuse makes property
models a viable alternative to more traditional user interface
development methods.

Categories and Subject Descriptors
D.2.2 [Design Tools and Techniques]: User Interfaces – D.2.3 [Reus-
able Software]: Reuse models

General Terms
Algorithms, Design

Keywords
Software reuse, Component software, User interfaces, Declarative speci-
fications, Constraint systems

1. Introduction

The role of a user interface, such as a dialog, assistant, or software to
support this rich set of interactive features. It is commonplace to find applications with
dependencies between values manipulated by user interface ele-
ments, which can control or manipulate, the state of the rest of the program. For
example, a dialog box may have controls that are disabled or de-
active until a particular condition is satisfied. Such dependencies lead to complex,
and sometimes surprising, behavior that can be difficult to manage.

User interfaces are a key concern of modern software development. They are critical to the
user experience, and are often a primary source of software defects. In fact, software de-
fects in user interfaces are one of the most common types of defects found in software
systems. This is because user interface code is often complex, with many different
dependencies and interactions between different components.

Property Models

Property models are a formalism for specifying the behavior of a
user interface. They provide a way to describe how, as a by-
product of computations that maintain the state of the rest of the
program, the user interface must change. This behavior is com-
plicated by the fact that user interfaces can have many different
dependencies and interactions, and these dependencies can be
very complex.

The role of a user interface, such as a dialog window, can be
defined as a function that transforms the state of the rest of the
program. This function is often represented by a set of rules that
specify how the state of the user interface changes when certain
events occur. For example, when a user clicks a button, the
state of the user interface may change, and the value of a
property model may change as well.

Abstract

We report on the increased reuse, reduced defect rates, and im-
proved interferface quality that we have experienced through the
use of declarative property models. We show that while property
models can capture explicitly the conditional interactions of
user interface code, they also allow the code to be generated in
separate, reusable modules. These modules can then be applied
and integrated with many other applications, providing more uni-
form access to functionality. Where we have applied property
models, we have measured significant reductions in source-code size
with equivalent or increased functionality in the resulting
user interfaces. This is an important result for applications that
require code to be reused, and can be reused, across many different
applications. We believe that the increased reuse makes property
models a viable alternative to more traditional user interface
development methods.

Categories and Subject Descriptors
D.2.2 [Design Tools and Techniques]: User Interfaces – D.2.3 [Reus-
able Software]: Reuse models

General Terms
Algorithms, Design

Keywords
Software reuse, Component software, User interfaces, Declarative speci-
fications, Constraint systems

1. Introduction

The role of a user interface, such as a dialog, assistant, or software to
support this rich set of interactive features. It is commonplace to find applications with
dependencies between values manipulated by user interface ele-
ments, which can control or manipulate, the state of the rest of the program. For
example, a dialog box may have controls that are disabled or de-
active until a particular condition is satisfied. Such dependencies lead to complex,
and sometimes surprising, behavior that can be difficult to manage.

User interfaces are a key concern of modern software development. They are critical to the
user experience, and are often a primary source of software defects. In fact, software de-
fects in user interfaces are one of the most common types of defects found in software
systems. This is because user interface code is often complex, with many different
dependencies and interactions between different components.
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
