Aspectual Collaborations

Modules and Aspects

Johan Ovlinger

November 5, 2002

Get this presentation at
the research tab on my homepage:
`http://www.ccs.neu.edu/home/johan/research`

2 minute summary

Overview

Contrast two ways to decompose programs:

- Aspects capture complex crosscutting concerns
- Modules promote reuse and separate development

Both sound like good ideas.

2 minute summary

Modules: Safe and Reusable

Benefits

- Separate interface from implementation allows parallel development
- External linking promotes flexible reuse
- Analyze and understand modules in separation

Drawbacks

- Complex behaviors: have to choose between
  - implement as one monolithic module
  - have behavior *scattered* across several modules
- Module interface must predict both static and dynamic needs

2 minute summary

Aspects: Expressive

Benefits

- Localize complex behaviors for easy comprehension
- Extensions can be added without modifying the base

Drawbacks

- Hardwired details limit reuse
- Hard to combine genericity with type safety
- Need to understand whole program to understand local behavior

2 minute summary

Contributions

Show that combining Aspects and Modules is

- **Desirable**: Neither aspects nor modules are "good enough" on their own.
- **Possible**: A simple yet powerful module system that has support for aspect-oriented features is designed and implemented.
- **Safe**: The formal properties of the system are investigated. Type soundness is proven for a simplified system.

20 minutes start now

To Seek the Holy Grail

We want to build programs that are:

- Large
- Complex
- Delivered Quickly (or at least on time).

The OO domain of solutions has [at least] two tools for the job:

Module Systems

- Separating interface from implementation allows parallel development
- Analyze and understand modules in separation
- External linking promotes flexible reuse – and allows compositional understanding

Large and On-time programs can be written by teams of programmers, leveraging code reuse and parallel development.

Aspect-Oriented Programming

- Localize complex behaviors for easy comprehension
- Extensions can be added without modifying the base

Inherently complex behaviors are localized for easier specification and comprehension.

Contradictory Assumptions

However, we find that these tools do not mesh cleanly.

Modules:

- external composition
- enforce encapsulation barriers
- separate implementation from interface

Modules become: easy to reuse, promote safety over ad-hoc power

AOP:

- hard-wire assumptions about the base program into advice
- can intercede in internals execution of base program
- advice can ignore encapsulation barriers

The effects are: expressive, promote ad-hoc power over safety

If we have to choose

If we promote modules over AOP:

- scattering and tangling
- require prescient ability to prepare code for all future uses

If we promote AOP over modules:

- hard to reuse
- require omniscience to understand local behavior

Either way, we need godlike powers to effectively write code.

Why bother?

Many programmers think Object-Orientation is good enough. Rising software costs will question this assertion: project costs need to decrease, either through working less (reuse of modules) or smarter (separation of crosscutting concerns using aspects).

This research illustrates how these options can be combined. The best of both worlds is both attractive and possible.

Questions Answered

We want to answer:

- Can modules' encapsulation be balanced with aspects' need for internal access?
- What are semantics of separately analyzed aspects?
- What static guarantees can we make for such aspects?

In general:
How do we leverage safety of modules to temper the power of aspects?

(you have to wait 'till the end to find out)

Aspectual Collaboration Sketch

- Orthogonal features: taken simple module system, added aspectual features
- Separate compilation
- External composition, with compositional semantics
- Symmetry: both advice and base can be composed
- Generic reuse, both of OO and AO code
- Provably type safe (I think): no casts or fancy type tricks
- Encapsulation is key

Deliverables

There are three deliverables that will be produced

- The design of Aspectual Collaborations. This functions as an existence proof that it is possible to combine modules and aspects.
- The Aspectual Collaboration Compiler `acc`. This verifies that the design is plausible and implementable.
- A smaller version of the language design is formalized and proven sound. This illustrates that the design is possibly type-safe.
- Case studies show that the design is workable on real-scale problems.

Achievement

And you will know us by...

Eating your own dog food is good for you. The current prototype compiler will be rewritten in itself. This will provide datapoints for either accepting the hypothesis that the language works (at least for writing compilers), or else indicate the factors that make it unrealistic for everyday use.

⊙ Out of time? Go to the end.

A List Widget Collaboration

```
1  collaboration origList;
2  import gui.Canvas; import gui.Screen; import gui.Widget;
3  participant List extends Widget {
4    Item[] items;
5    void addItem(Item) {{ /* elided for space reasons */ }}
6    Item removeFirst()  {{ /*      -- ditto  --       */ }}
7    void draw() {{ paint(Screen.getCanvas()); }}
8    void paint(Canvas canvas) {{
9      for (int i=0; i++; i<items.length) {
10       Canvas label = Canvas.drawString(items[i].showString());
11       canvas.append(label);
12     }
13   }}
14 }
15 participant Item {
16   expected String showString();
17 }
```

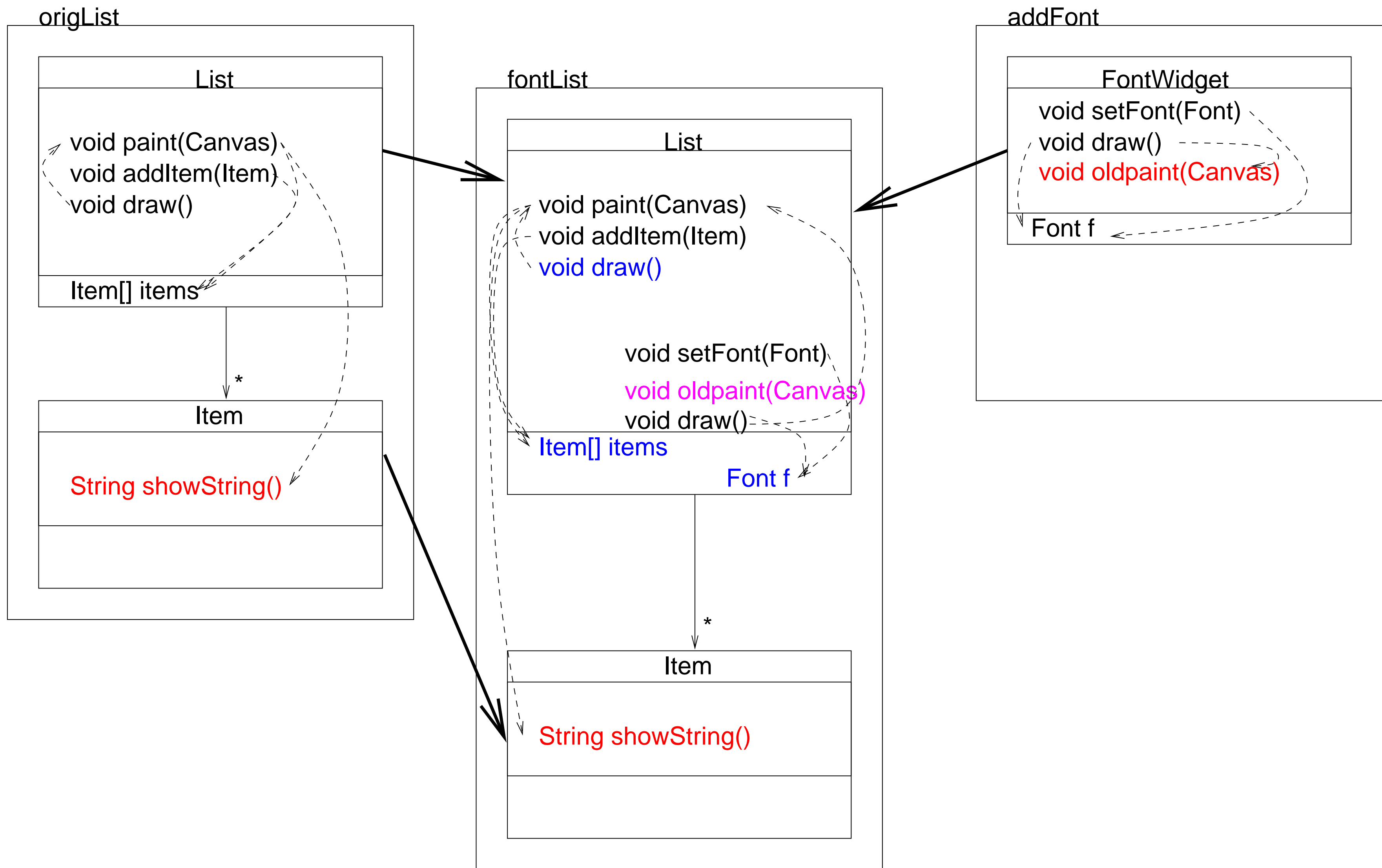
- Reuse `List` several times for various `Item` types.
- skip to aspectual method.

Motivating Problem Client

We run a GUI design firm, and have licensed the previous code (in compiled form) from COTS vendor. Our one client decides that they *must* be able to change the font of the List.

Supporting Fontification

```
1  collaboration addFont;
2  import gui.Canvas; import gui.Screen; import gui.Font;
3  participant FontWidget {
4    Font font;
5    void setFont(Font f) {{
6      font = f;
7    }}
8    void draw() {{
9      Canvas canvas = Screen.getCanvas();
10     canvas.useFont(font);
11     oldpaint(canvas);
12   }}
13   expected void oldpaint(Canvas canvas);
14 }
```


origList
List
void paint(Canvas)
void addItem(Item)
void draw()
Item[] items
*
Item
String showString()
fontList
List
void paint(Canvas)
void addItem(Item)
void draw()
void setFont(Font)
void oldpaint(Canvas)
void draw()
Item[] items
Font f
*
Item
String showString()
addFont
FontWidget
void setFont(Font)
void draw()
void oldpaint(Canvas)
Font f

Motivating Problem Client 2

Our client wants the list to always fit on screen.

We now need to verify that we never add beyond a certain number of items to the list.

Solution: capture each execution of a method that can change the length and only proceed if the list is not at capacity. We will want to track the length of the list.

Aspectual Collaborations

```
1  collaboration addLimit;
2  participant Limited {
3    int limit; int count;
4    void setLimit(int l) {{ limit = l; }}
5    aspectual IncR increase(IncJP jp) {{
6      if (count < limit) {
7        count++;
8        IncR ret = jp.invoke();
9      } else {
10       IncR ret = jp.dontInvoke();
11     }
12     return ret;
13   }}
14   aspectual DecR decrease(DecJP jp) {{
15     count--;
16     return jp.invoke();
17   }}
18 }
```

Attaching Aspects

```
 1 collaboration limitFontList;
 2 import gui.Widget;
 3 participant List extends Widget;
 4 participant Item;
 5 attach fontList, addLimit {
 6   List += fontList.List, Limited {
 7     export setFont;
 8     export addItem;
 9     export removeItem;
10     export draw;
11     export setLimit;
12     around addItem do increase;
13     around removeFirst do decrease;
14   }
15   Item += Item {
16     export showString;
17   }
18 }
```

Syllables

Aspect	2	bad
Module	2	bad
Aspectual Collaboration	9	good

Conclusion: longer names are better

20 minutes end now

⊙ The example we probably didn't have time for.

Instantiating Lists for Different Types

```
1  collaboration instFooList;
2  participant FooList extends Widget;
3  participant Item;
4  attach origList, myDataTypes {
5    FooList += origList.List
6      { export draw; }
7    Item += origList.Item, myDataTypes.Foo
8      { provide showString with Foo.toPrettyString; }
9  }
10
11 collaboration instFileList;
12 participant FileList extends Widget;
13 participant Item;
14 attach origList, io {
15   FileList += origList.List
16     { export draw; }
17   Item += origList.Item, io.File
18     { provide showString with fileName; }
19 }
```