

Aspectual Collaborations

Modules and Aspects

Johan Ovlinger

January 7, 2003

Get this presentation at
the research tab on my homepage, or directly:
<http://www.ccs.neu.edu/home/johan/research/AC-tp-p>

Problem Statement

We want to build programs that are:

- Large
- Complex
- Delivered Quickly.

We'll investigate the OO domain of solutions. There we have [at least] two tools for the job:

- Module Systems

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

- Aspect Oriented Programming

Crosscutting concerns control complexity.

Tensions

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

Modules:

- only make assumptions allowed by interface:
easy to reuse
- enforce encapsulation barriers:
safety over ad-hoc power

AOP:

- hard-wire assumptions about the base program into a
difficult to reuse
- assume that advice can ignore encapsulation barriers:
ad-hoc power over safety

Is this really necessary?

or

Mudslinging

If we promote modules over AOP:

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

If we promote AOP over modules:

- hard to reuse
- require omniscience to understand local behavior

Either way, we need godlike powers to effectively write co

Thesis

We believe that a module system explicitly catering to AOP can elegantly combine the strengths of both systems, allowing us to demonstrate a step towards large, complex, and timely projects (Even without godlike powers)

In more detail, we want modules that support

- Separate Analysis (sufficient for compilation)
- External Composition
- Encapsulation
- Flexible Reuse (upto class-valued genericity)
- Support for AOP features

Structure of Talk

1. Introduction and Thesis ← (You are
2. Terminology ⊙
3. Basic Aspectual Collaborations by Example ⊙
4. Advanced Topics ⊙
5. Work in Progress ⊙
6. History and Questions ⊙

And at various points:

- a sampling of bonus slides
- no greek!

AOP Terminology

Original Definitions

- **Concern:** A property of the program that reflects on programmers' interests, an *issue* in the program's behavior. Decomposition into modularized concerns is key for software maintenance.
- **Aspect:** Alternatively, an aspect is identified as a procedure that cannot be implemented as a generalized procedure (method, object, API, and presumably module) *in the target language*. "A well modularized implementation of a *crosscutting* concern."

They are almost the same. Both speak of

- what the programmer cares about.

An Aspect additionally speaks about

- how they interact.

Better Terminology

New Definitions

- **Concern:** Something of interest to the program behavior.
- **Aspect:** The external specification of [possibly crosscut] concern interaction.
- **Joinpoint:** A point of concern interaction.
- **Advice:** how joinpoints interact.

⊙ bonusslide: How do we classify existing systems using these

Concerns

There are several kinds of concerns:

- **Structural:** traditional programming, including mod
- **Behavioral:** oblivious programming (reacting to exte
conditions and allowing external observers to modulat
behavior)
- **Real Time**
- **Robustness**
- **Resource Optimization**

the list is endless

We've chosen to deal with the first two: lots of low hangin

Thesis Restated

In the new terminology:

There exists an attractive balance between the structural and behavioral concerns that captures the power of both, without requiring onerous sacrifices.

We provide an existence proof.

Time for Some Code

A List Widget Col

```
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 }
```

⊙ bonusslide: We can instantiate the `List` several times for v
types.

Motivating Problem Client

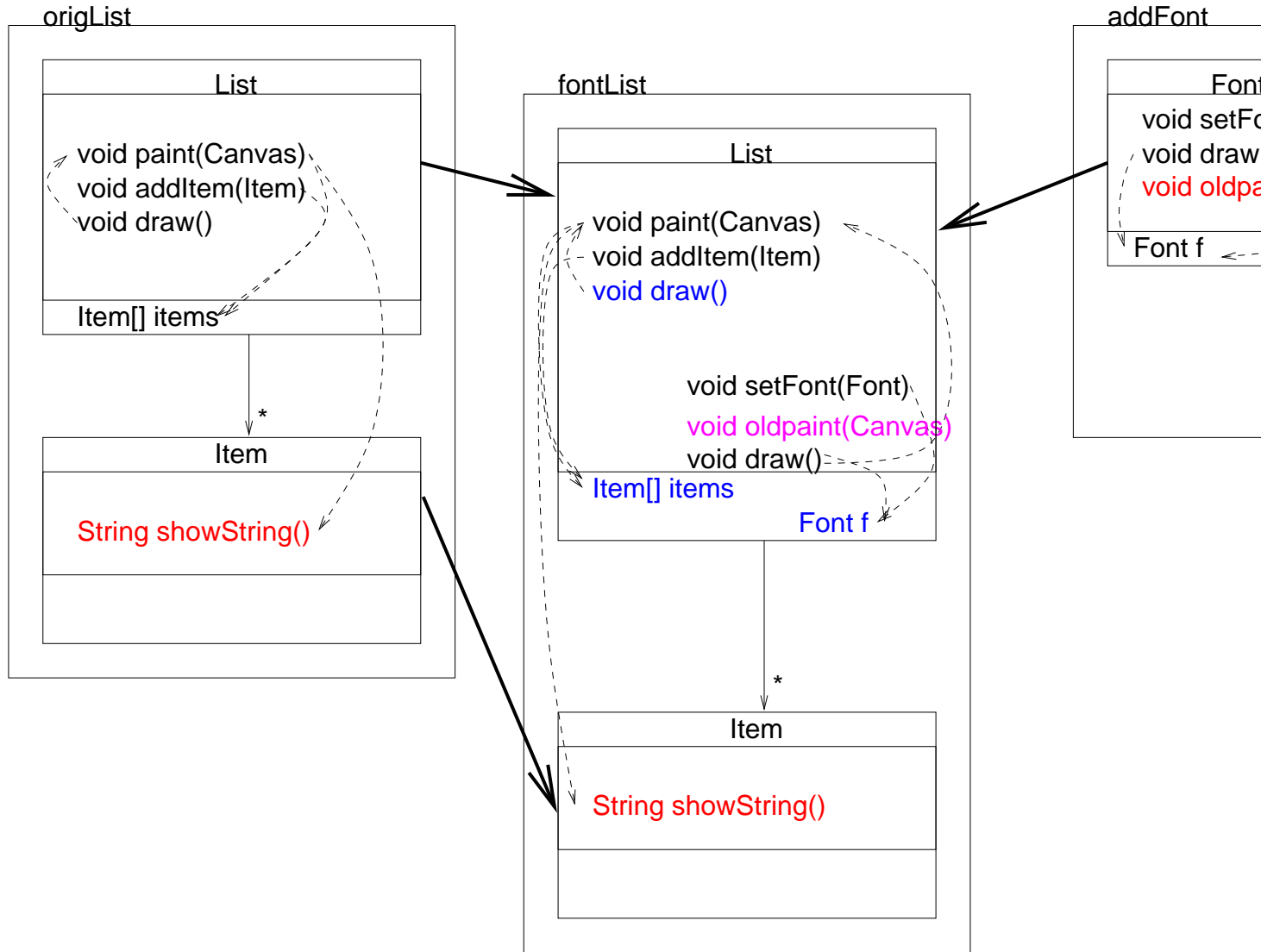
We run a GUI design firm, and have licensed the previous (compiled form) from COTS vendor. Our one client decided they *must* be able to change the font of the List.

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 }
```

Compose List and Fontification

```
1 collaboration fontList;
2 import gui.Widget;
3 participant List extends Widget;
4 participant Item;
5 attach origList , addFont {
6   List += origList.List , FontWidget {
7     provide oldpaint with paint
8     export setFont;
9     export addItem;
10    export FontWidget::draw;
11  }
12  Item += origList.Item {
13    export showString;
14  }
15 }
```



- ⦿ bonusslide: Why I've omitted constructors.
- ⦿ bonusslide: How error locations are propagated through com

Structural Concern

We have seen how to use collaborations as a straight forward module system.

Joinpoint	Advice
collaboration	attach
participant	merge (+=)
expected member	provide
member	export

Notice how these points form a loose hierarchy: the static correctness of providing members depends on participant and collaboration attachment.

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 need a variable to track the length of the list.

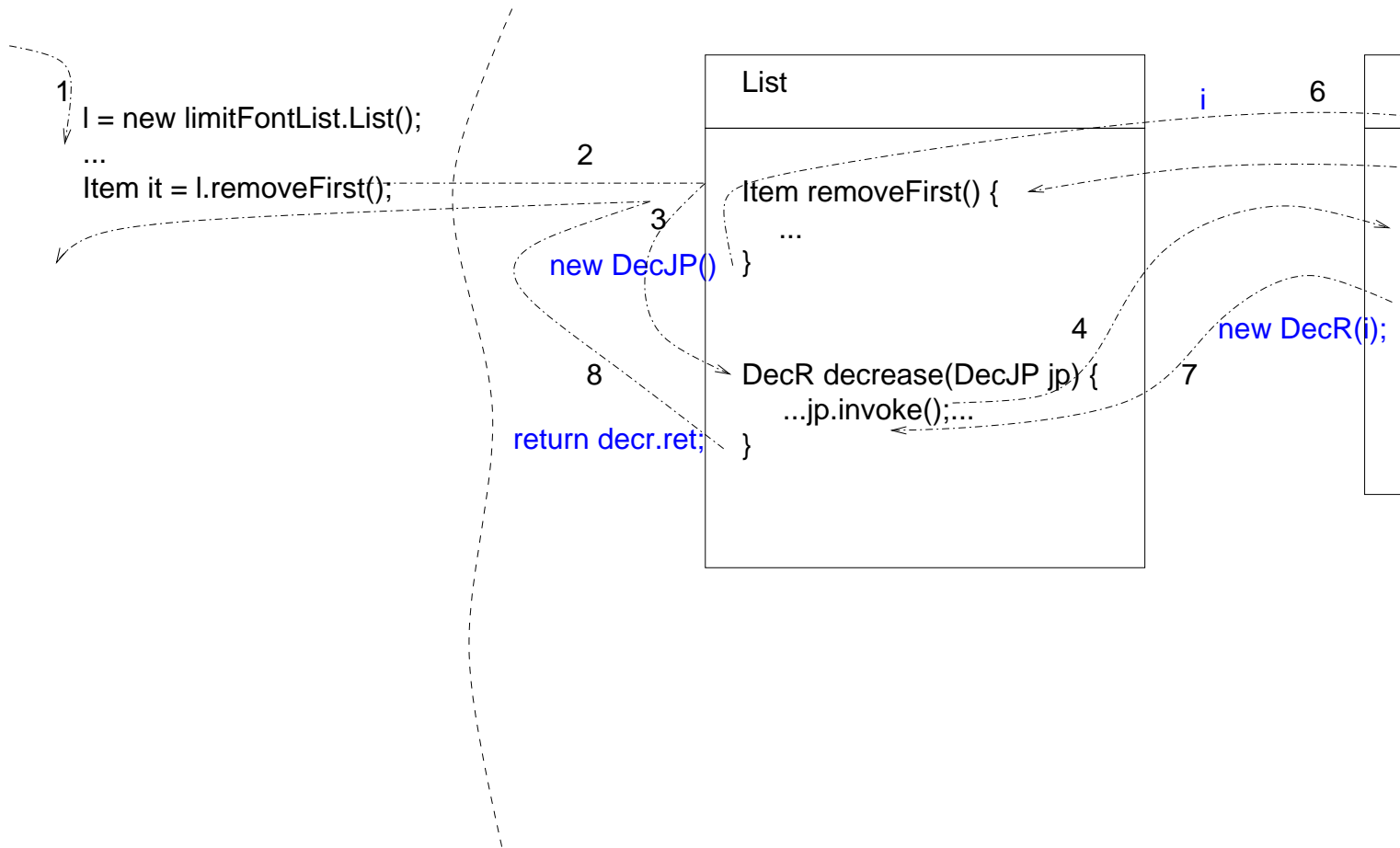
```
1 collaboration addLimit;
2 participant Limited {
3     int limit; int count;
4     void setLimit(int l) {{ limit = l; }}
5     aspectual IncR increase(IncJP jp) {{
6         if (count < limit) {
7             count ++;
8             IncR ret = jp.invoke();
9         } else {
10            IncR ret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

⊙ bonusslide: What API do Joinpoints provide

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 }
```

You should be in pictures



Behavioral Concern

Joinpoint:

Joinpoint

Advice

method execution

interception and control

This joinpoint is also dominated by structural concerns: specifically, the participant mapping, which decides whether behavioral advice and joinpoint are well specified.

Static Properties of Collaborations

- **Encapsulation:** Things not exported can't be accessed.
- **Type Safety:** We guarantee only the normal Java type system and that composition does not introduce any *location* related runtime errors can occur.
 - Atoms are type safe by compilation.
 - A well typed collaboration remains well typed under systematic type renaming.
 - Adding well typed independent code to a collaboration cannot make it ill typed
 - Composition replaces references by *identically* typed references.
- **Non-invasiveness:** Code in an atom cannot determine whether it has been composed.

Aspectual Collaboration Summary

- Orthogonal features: taken simple module system, ad aspectual methods
- Separate compilation
- External composition, with compositional semantics
- Symmetry: both advice and base can be composed
- Generic reuse, both of structural and behavioral conc
- Have intrinsic signature, rather than views of external signatures
- No runtime representation, like proxy objects

⊙ bonusslide: Why views are a pain.

⊙ bonusslide: Why proxies are a pain.

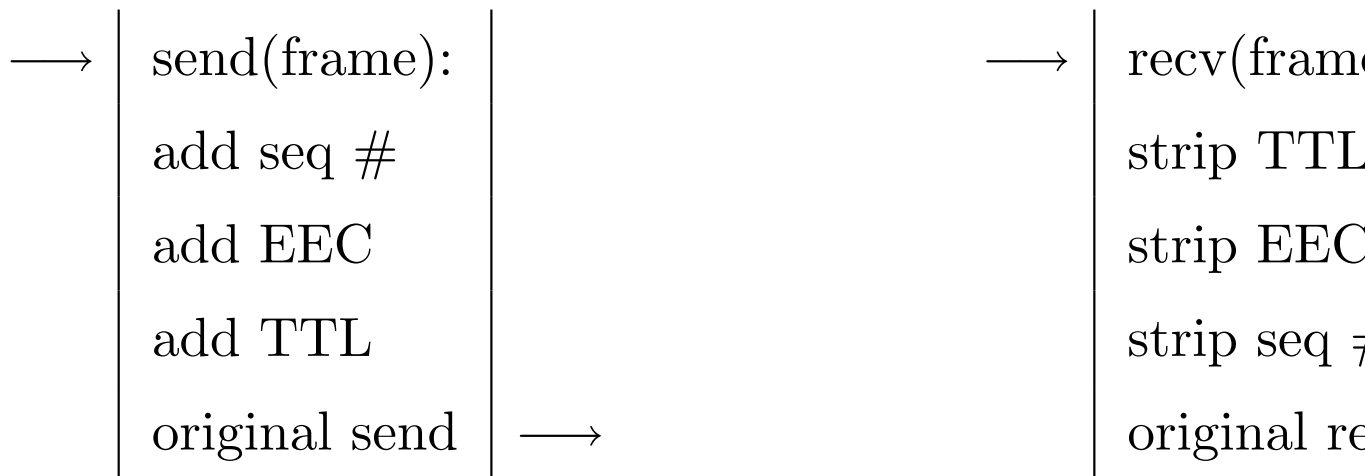
Advanced Topics

The second part of the talk shows how Aspectual Collaborators solve several problems in a clean and orthogonal way.

- Ordering and controlling advice interaction
- Multiple attachments
- Explicit sharing between attachments

Networking Stack Problem

Concerns: Sequence no.s, Error Correcting Codes, Time to live



Controlling Advice Interaction

Need: a method to control advice order.

AspectJ

- explicit mechanism for declaring domination relations
- works on the level of AspectJ aspects, rather than adv

ACs

- by order of around..do clauses: later dominates earlier
- granularity: individual methods

Motivating Problem Client 3

The client wants a menu on the screen as well. Of course, to fit too.

Luckily, we can reuse our limit collaboration.

Manual duplication

```
1 collaboration limitListAndMenu;
2 import gui.Widget;
3 participant List extends Widget;
4 participant Menu extends Widget;
5 participant Item;
6 attach origGUI, addLimit { // origGUI provides both List and I
7   List += origGUI.List, Limited {
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 } //
```

continu

```
19 attach origGUI, addLimit {
20   Window += origGUI.Window, Limited {
21     export addItem;
22     export removeItem;
23     export draw;
24     export setLimit;
25     around addItem do increase;
26     around removeFirst do decrease;
27   }
28   Item += Item {
29     export showString;
30   }
31 }
```

Pointcuts as per AspectJ

pointcut incJP(LimClass): **targetclass**(LimClass) && method(* * >)

pointcut decJP(LimClass): **targetclass**(LimClass) && method(* * <)

incJP:

LimClass::	<i>joinpoint</i>
List ::	void addItem()
Menu ::	void addItem()

decJP:

LimClass::	<i>joinpoint</i>
List ::	Item remove()
Menu ::	Item remove()

which we could use:

... { ...

 LimClass += Limited { **around**(incJP): increase(); }

... } ... { ...

 LimClass += Limited { **around**(decJP): decrease(); }

... }

What we want

limitJP:

LimClass::	<i>joinpoint1</i>	<i>joinpoint2</i>
List ::	void addItem()	Item removeFirst()
Menu ::	void addItem()	Item removeFirst()

```

1 collaboration limitListAndMenu;
2 import gui.Widget;
3 participant List extends Widget;
4 participant Menu extends Widget;
5 participant Item;
6 match origGUI {
7   role <LimClass> {
8     void addItem(Item);
9     Item removeFirst();
10    - draw(-);
11    - setLimit(-);
12  }
13 } attach origGUI, addLimit {
14   <LimClass> += origGUI.<LimClass>, Limited {
15     export addItem; export removeItem; export draw; export
16     around addItem do increase;      around removeFirst do
17   }
18   Item += origGUI.Item { export showString; }
19 }

```

⊙ bonusslide: Existential types make it type safe

Motivating Problem 4

Our client has a new demand. Being a complete control fi demands that the menu never overlaps the list: hence we about their combined count, rather than separate.

Sharing and Related Joinpoints

```
1 collaboration addLimit;
2 participant Limited {
3   shared int limit; shared int count;
4   shared 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 }
```

End of Advanced Issues

or

Segue to the Work in Progress

A closer look at aspectual methods, multiple attachments sharing shows deficiencies:

- need to see a method to advise it.
- need to have regular structure to use matching
- matches are textual: only make sense in context of a template.
- sharing is awkward. When is sharing type safe? How nested sharing?

Aspectual Collaborations mk II

(vaporware)

Key is to be able to

- separate behavioral joinpoint from method
- allow several joinpoints to be grouped into a tuple
- combine tuples into a set of similarly-typed tuples (call it JoinPoint Table)
- export such joinpoints and JPTs as top level entities in a collaboration
- replace attachment clauses by comprehension of JPTs

Top-level Joinpoints

```
1 collaboration origGUI;
2 participant List extends Widget { ...
3   void addItem(Item) {{ ... }}
4   Item removeFirst() {{ ... }}
5   ... }
6 participant DifficultMenu extends Widget { ...
7   void addItem(NotItem) {{ ... }}
8   NotItem removeFirst() {{ ... }}
9   ... }
10 participant Item { ... }
11 participant NotItem { ... }
12
13 JPT listAddRem = List :: void (Item), List :: Item ()
14   { exec List::addItem, exec List::removeFirst }
15 JPT menuAddRem = DifficultMenu :: void (NotItem), DifficultMenu
16   { exec DifficultMenu::addItem, exec DifficultMenu::removeFir
17
18 //
```

continued

```

19 JPT listAddRem = List :: void (Item), List :: Item ()
20   { exec List::addItem, exec List::removeFirst }
21 JPT menuAddRem = DifficultMenu :: void (NotItem), DifficultMenu
22   { exec DifficultMenu::addItem, exec DifficultMenu::removeFir
23
24 JPT anonListAddRem =  $\exists$  Owner Type . Owner :: void (Type),
25   {  $\exists$  List, Item . listAddRem }
26 JPT anonMenuAddRem =  $\exists$  Owner Type . Owner :: void (Type)
27   {  $\exists$  DifficultMenu, NotItem . menuAddRem }
28 JPT allAddRem =  $\exists$  Owner Type . Owner :: void (Type), Owner
29   { anonListAddRem || anonMenuAddRem }

```

\exists	τ_1	τ_2	$\tau_1 :: \text{void } (\tau_2)$	$\tau_1 :: \tau_2 ()$
	List	Item	Line :: void addItem()	Line :: Item removeFir
	Menu	NotItem	Menu :: void addItem()	Menu :: NotItem remo

Using 'em

This bit is still unfinished.

\exists	τ_1	τ_2	$\tau_1 :: \text{void } (\tau_2)$	$\tau_1 :: \tau_2 ()$
	List	Item	Line :: void addItem()	Line :: Item removeFir
	Menu	NotItem	Menu :: void addItem()	Menu :: NotItem remo

```
1 origGUI.allAddRem for-row (OwnerClass, ItemType, incJP, de
2   attach addLimit {
3     OwnerClass += Limited {
4       around incJP do increase;
5       around decJP do decrease;
6       export setLimit;
7     }
8   }
9 }
```

Brief History, Related Work

- Ian Holland's Contracts
- Crista Lopez's COOL and RIDL
- Karl and Mira's AP&PC
- My Class Graph Views (aborted)
- AspectJ
- Hyper/J
- Jiazzi

⊙ bonusslide: Why Class Graph Views were aborted

Proposed Further Work

- Finish precise semantics for top-level joinpoints (for a small language), and prove sound.
- Deal with constructors, at least partially
- Implement mk II. Use Jiazias a back end?

⊙ bonusslide: Why constructors are a pain.

Contribution






We have presented a combination of modules and AOP that provide the benefits of modules for Java, while also providing Aspect Oriented Programming features.

- Separate Analysis (and compilation)
- External Composition
- Encapsulation
- Generic, type-safe aspectual behavior
- Maintaining relationships between joinpoints (e.g. inc/dec)
- Finegrained control of advice state and ordering
- Separate ability to advise from ability to invoke.

We additionally provide a better (at least non-overlapping) definition of Aspect.

You don't have to go home, but you can't stay here

I can answer questions on anything, but if you ask about you'll get pretty slides too.

- cool ideas that will never get implemented
 - Collaboration Parametricity 
 - Object Models as Interfaces 
- Relationship of aspects and events 
- Why members, rather than participants, are unit of in
- Compare and Contrast early binding (a-la ACs) and l
binding (a-la Jiazzi). 
- When am I graduating? 

this slide intentionally left blank

Blue Sky - Parameterizing over collabora

parametricity for collaborations: allow attachments to work with as-yet unknown collaborations.

- Need to separate collaboration name from signature.
- Want concept of subtyping/implementation of signature.

Blue Sky - Object Models as Interface Re

- Collaborations will induce different object models when executing.
- Just because participants have been mapped doesn't mean object models are compatible.
- Would like to determine this statically.
- Difficult to find interesting *requires* and *provides* constraints that are flexible, also unclear how to compose constraints.

Members as unit of Import

and

Early vs Late Binding of Class Name

- Would need to specify visible members on imported participants anyway
- Importing participants would require late binding of final name (either programming pattern: import final participant well or factory pattern).
- Would require concept of participant View.

Why Views are a Pain

- A view of a participant declares the minimum signature it expects to have. However, the actual definition may have more members.
- Trying to export one of these unseen members, will generate a name clash.

Surprise!

Magic Eight Ball Schedule

- October: AOSD submission (11th), BCS revision, Pre OOPSLA
- November: go to OOPSLA (4-8th), work on Soundness grant
- December: Hopefully soundness work complete by xmas
- January-June: Write thesis and implement semantics parallel.

How we propagate error locations from ne

JVM Constraints:

- Compiled classes can report error from at most 1 source file
- Bytecode ranges specify which lines in the file they correspond to.

~~Hacks~~ Solutions:

- Create a fake source file that has combined source: write a plugin for IDEs or jdb. If you have the source, that is...
- Write a new `java.lang.throwable` that is able to translate bytecode line numbers into real source locations. Put this before the standard libs in your CLASSPATH
- Write IDE plugin.

Why Constructors are a Pain

Constructors:

- Internal name `<init>` is hard wired: guaranteed name
- Create new constructors, unioning bodies
- If constructors for a class are overloaded, which should correspond? What order?
- Want to initialize an object before invoking methods (How to make sure that we don't cross to other particular expected method) which is not initialized yet?
- Even plain Java gets this wrong

Why Proxy Objects are a Pain

ACs are composed at compile time, leaving no representation to differentiate a composite collaboration from one programmed equivalently as an atom.

More dynamic systems (using a broad brush):

- composition generates proxy subclasses of base classes
- base instances can be dynamically “wrapped” with new behavior by instantiating proxy classes.
- component types are different “kind” than base types

Has side-effects

- Need to manage correspondence between base object and virtual object.
- Need to invoke virtual types to achieve type safety. See Bruce and Ernst for limitations. Bruce points out that static typing requires contravariant arguments to be “exact”.
- Need to traverse entire object structure of arguments across component boundaries to translate between kind of call by copy

Using the terminology

		Concern	Aspect
AspectJ	base program	×	
	aspect	×	×
Hyper/J	hyperslice	×	
	hypermodule		×
Jiazzi	atom	×	
	compound		×

Instantiating Different Typ

```
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 }
```

API of Joinpoints

The argument type of aspectual method (IncJP):

- always get `invoke` and `dontInvoke`
- optionally, [subset] of arguments to method call

The return value (IncR):

- [always] query whether exception was thrown (not implemented)
- optionally, returned value

Choosing Which Arguments To Pass

```
1 collaboration fontList;
2 import gui.Widget;
3 participant List extends Widget;
4 participant Item;
5 attach origList , addFont {
6   List += origList.List , FontWidget {
7     around - paint(Canvas canvas) do fontPaint
8     export setFont;
9     export addItem;
10    export draw;
11  }
12  Item += origList.Item {
13    export showString;
14  }
15 }
```

Eventual Continuum

	Events	ACs	
publisher	Controls events and exposed info	Oblivious	Obl
external		Controls events and exposed info	
subscriber	Separately compiled from publisher	Separately compiled from publisher	Con exp Can com

Are You Sure it's Type Safe?

```
1  aspectual DecR decrease(DecJP jp) {{
2    count--;
3    DecR ret = jp.invoke();
4    dataOutputStream.write(ret);
5    return (DecR) dataInputStream.read();
6  }}
```

```
1  participant List {
2    Item removeFirst()
3  }
4  ...
5  attach ...
6  List += origGUI.List, Limited {
7    around removeFirst do decrease;
8  }
```

```
1  participant Difficult
2  NotItem removeFi
3  }
4  ...
5  attach ...
6  Menu += Difficult
7  around removeF
8  }
```

⊙ bonusslide: How error locations are propagated through cor

How to read Aspectual Collaboration

```
1  ∃IncR, IncJP, DecR, DecJP {
2    collaboration addLimit;
3    participant Limited {
4      int limit; int count;
5      void setLimit(int l) {{ limit = l; }}
6      aspectual IncR increase(IncJP jp) {{
7        ...
8      }}
9      aspectual DecR decrease(DecJP jp) {{
10       count--;
11       DecR ret = jp.invoke();
12       dataOutputStream.write(ret);
13       return (DecR) dataInputStream.read();
14     }}
15   }
16 }
```