Towards More Efficient Aspect Weaving

Nate Roberts, Presenter
Eunjee Song, Mentor

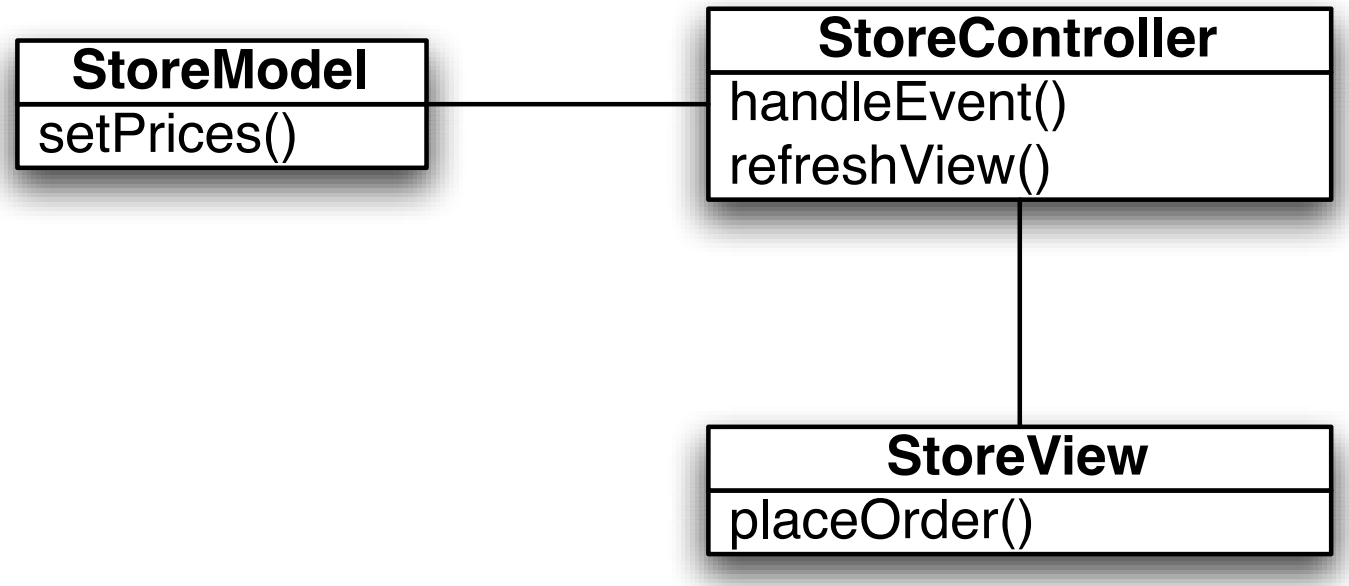
Outline

1. Successes & Failures of OOP
2. AOP as remedy
3. How to *model* for AOP: one (slow) approach
4. Making AOM weaving faster
5. Some other ideas to improve upon the AOM approach
6. Conclusions & Future Work

Background: Object-Oriented Programming

- Allows for substantial separation of concerns
  - Interfaces can be separated from implementation
  - Often one concern's details can change without affecting the code belonging to other concerns

Background: Object-Oriented Programming



Background: Object-Oriented Programming

- But there are situations where OOP cannot neatly separate the concerns!
- Code for logging and other concerns that cut across the class hierarchy gets scattered throughout the codebase.
- Modifications to these concerns can be tedious and error-prone.

Background: Aspect-Oriented Programming

- AOP takes OOP and adds **cross-cutting concerns**.
- You let the compiler do the scattering for you, according to rules that you specify.
- Cross-cutting concerns are called **aspects**.

Background: Aspect-Oriented Programming

- The code you write for the cross-cutting concern is called the **advice**.
- The "scattering" action of the compiler is called **weaving**.
- The points at which the advice is woven are called **join points**.
- A set of join points is called a **point cut**.

Background: AOP— What AspectJ Allows

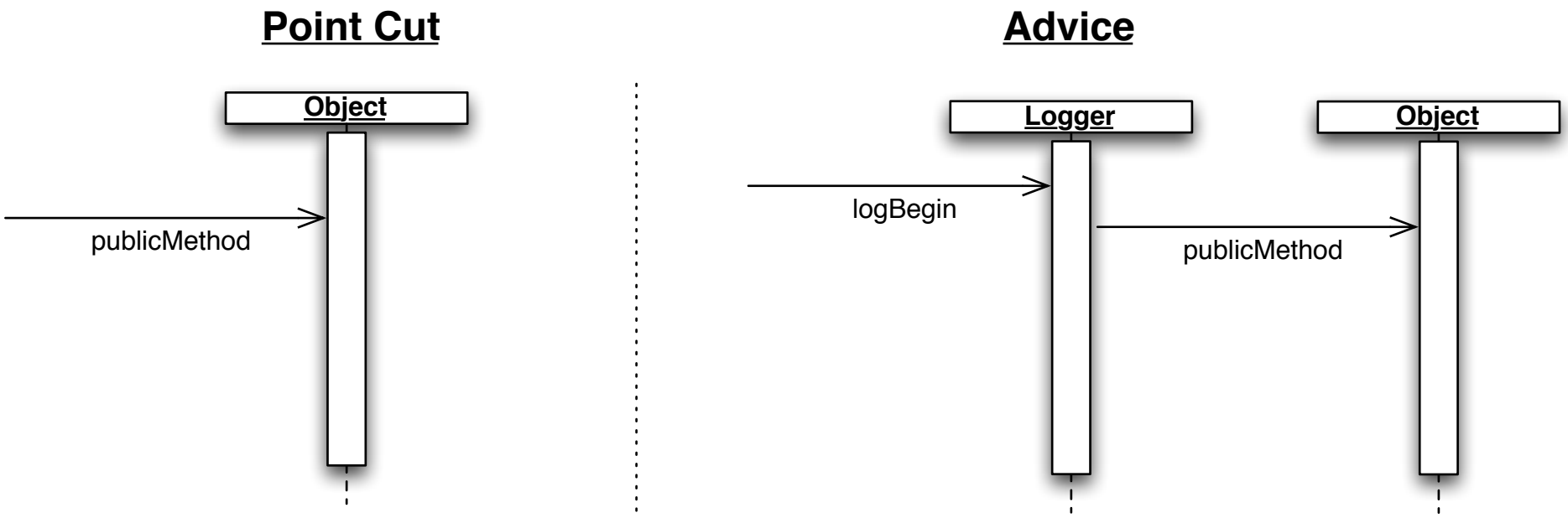
- method call
- method execution
- constructor call
- constructor execution
- static initializer execution
- object pre-initialization
- object initialization
- field reference
- field set
- handler execution
- advice execution

Modeling with Aspects: Motivations

- Software engineers like to be able to express designs as models.
- Aspects describe design concepts, not implementation details.
- But AOP defines join points as points in the execution of the code!
- How do we express aspects in models?

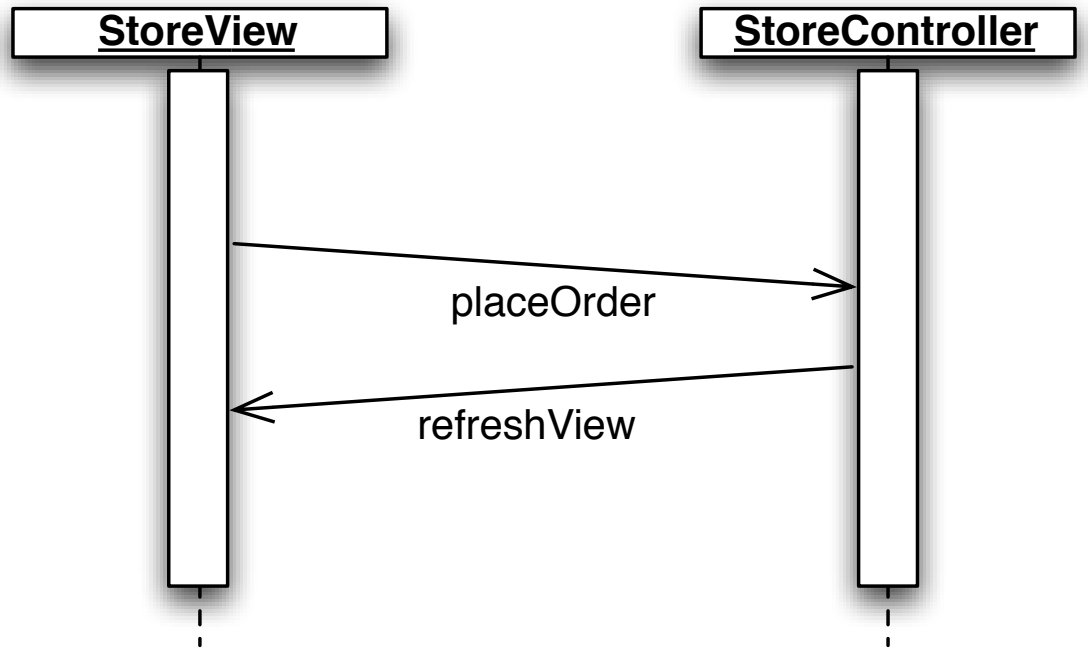
Modeling with Aspects: Approach

- So we express aspects as pairs of sequence diagrams, one to represent the point cut, and one to represent the advice to be woven:



AOM: Ambiguity

- But what counts as a match? Where in the base code do we weave the advice?



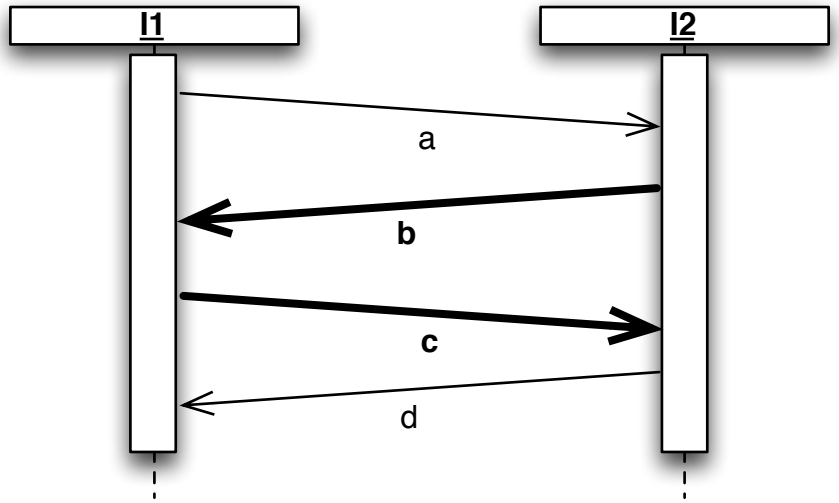
Should placeOrder match? How about refreshView?

AOM: Resolving Ambiguity

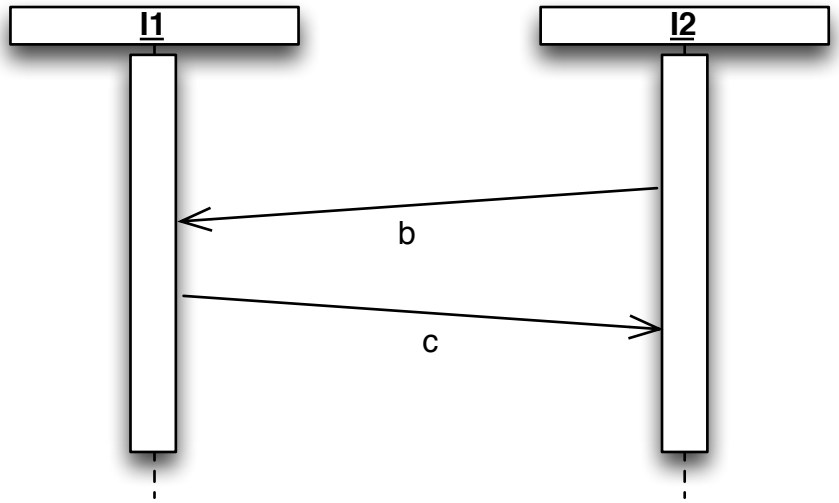
Jacques Klein, et al approach this as follows:

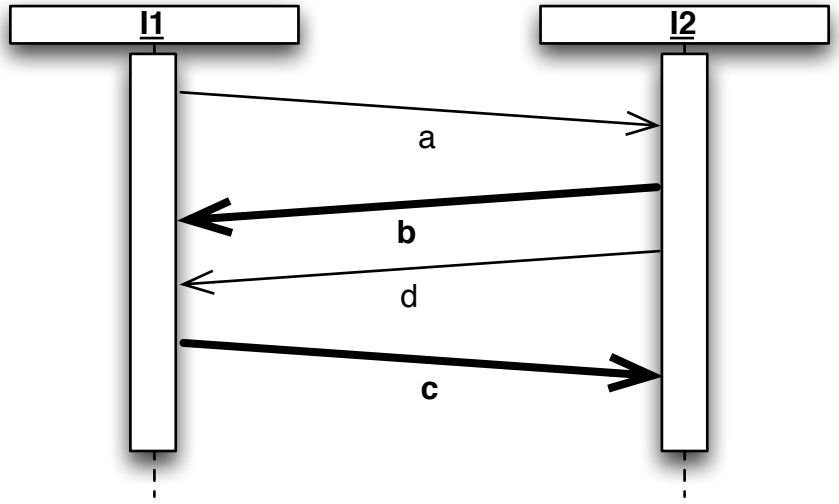
1. add constraints on the SDs allowed
2. provide four choices for aspectual SD interpretation (today: "enclosed part")
3. provide an algorithm to do the weaving

Enclosed Part

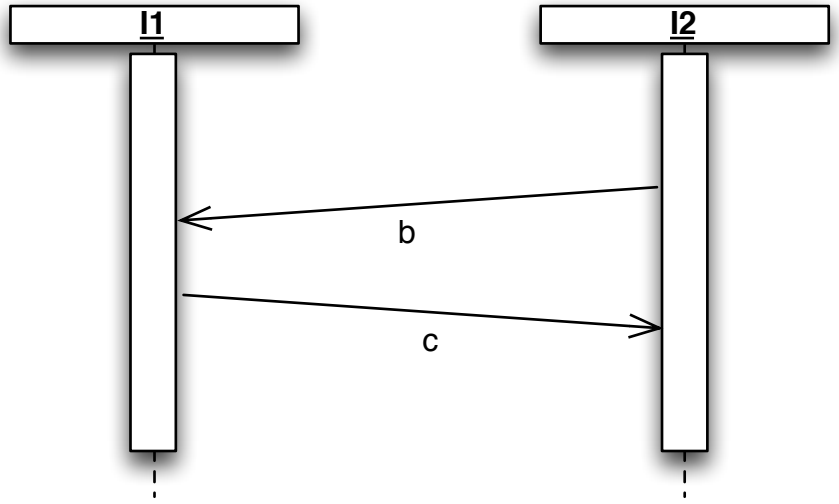


matches

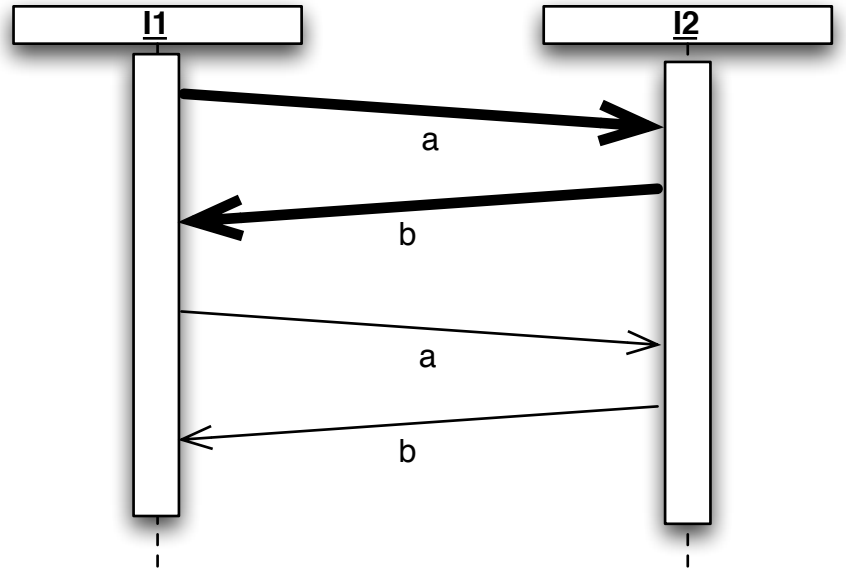




does not match

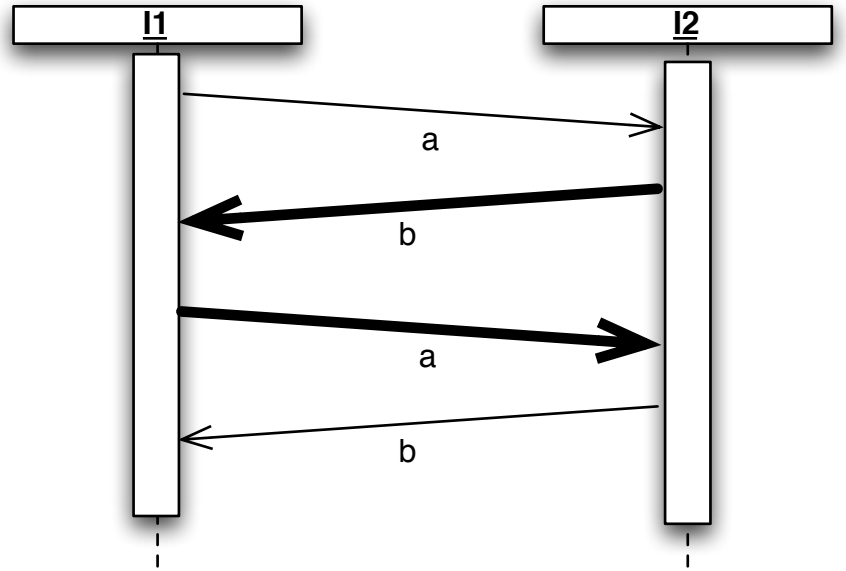


Klein's enclosed part algorithm



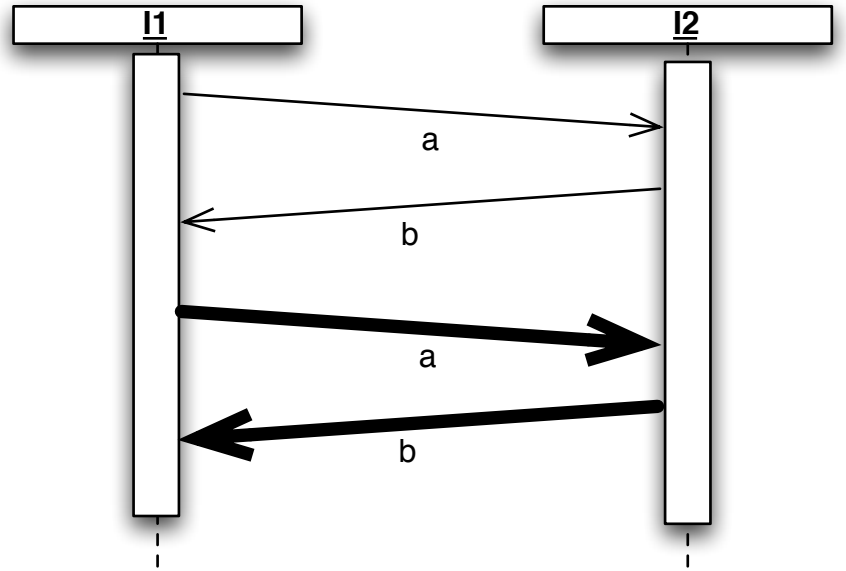
1. Search through the base bSD for matches: sets of event sets of the appropriate length, with matching action names.

Klein's enclosed part algorithm



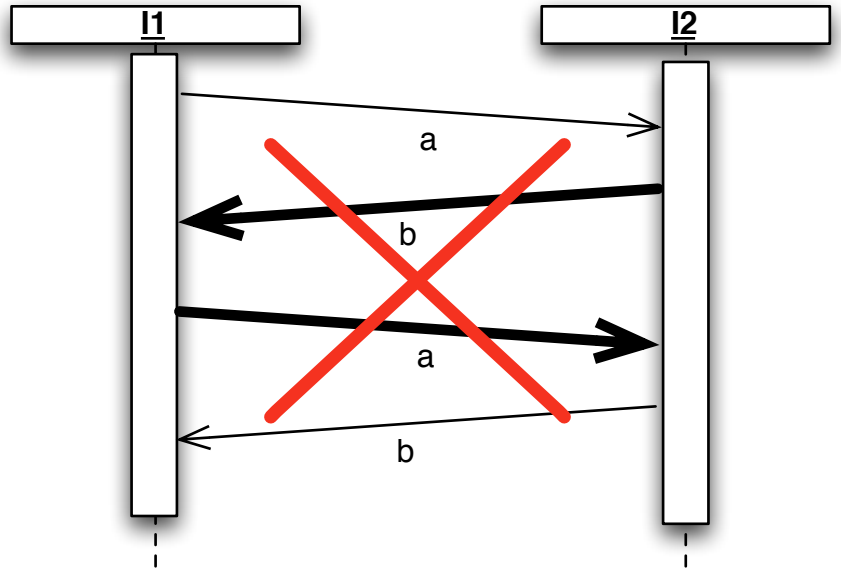
1. Search through the base bSD for matches: sets of event sets of the appropriate length, with matching action names.

Klein's enclosed part algorithm



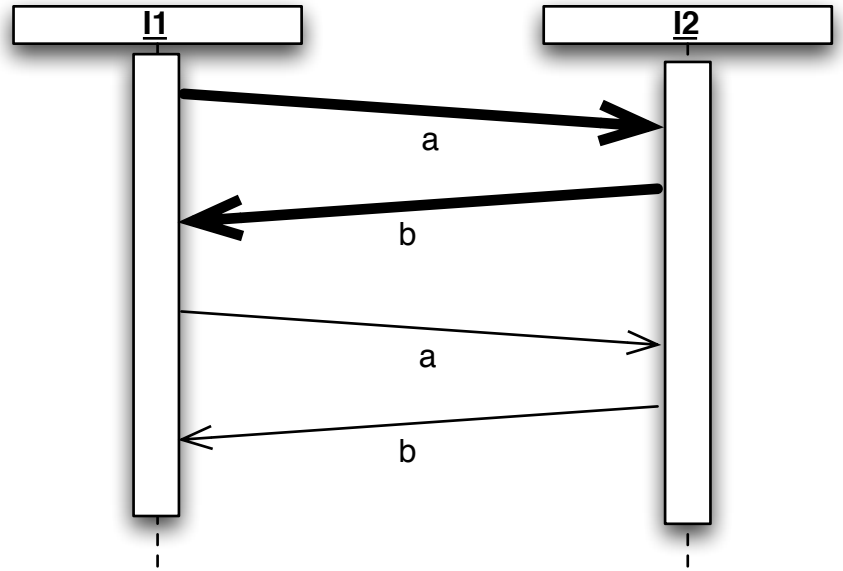
1. Search through the base bSD for matches: sets of event sets of the appropriate length, with matching action names.

Klein's enclosed part algorithm



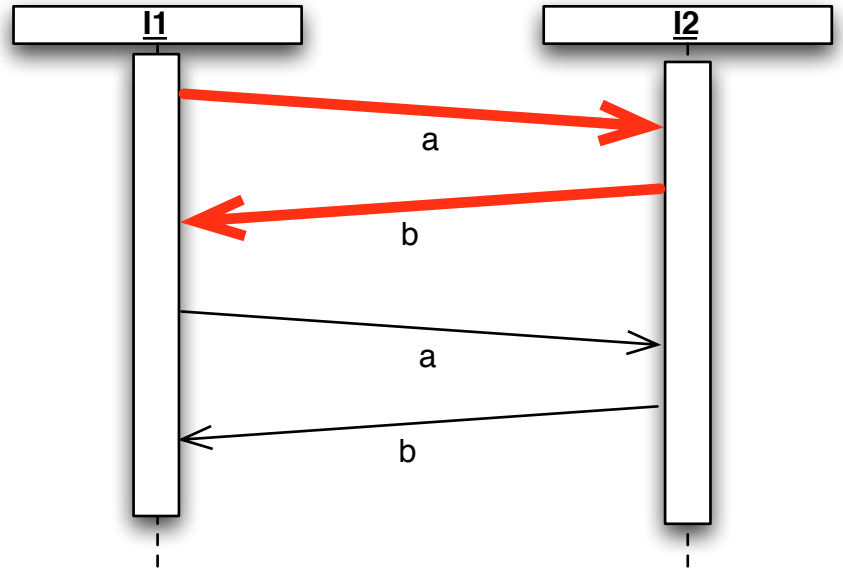
2. Because the name match check is done as a set comparison, we still need to check that the events are in the right order.

Klein's enclosed part algorithm



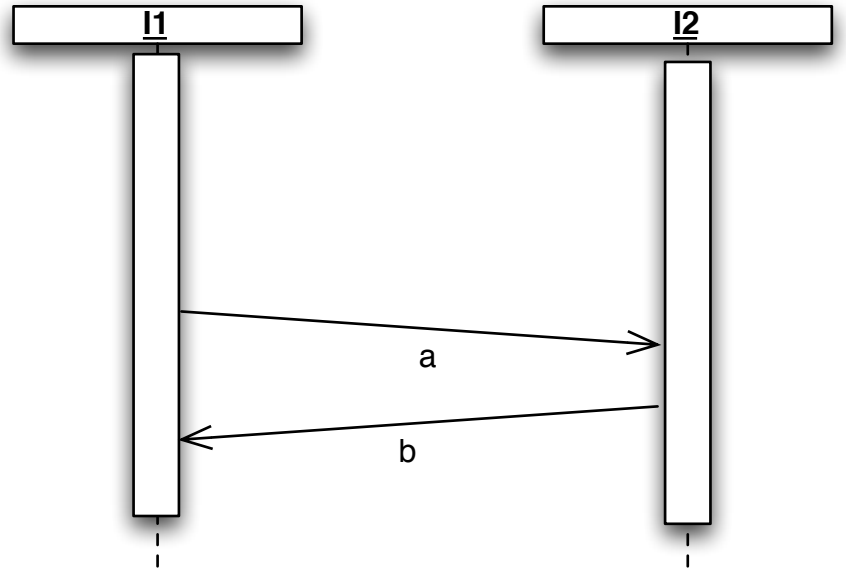
3. Then take the minimal (“earliest”) match from among the matches. This is the first join point.

Klein's enclosed part algorithm



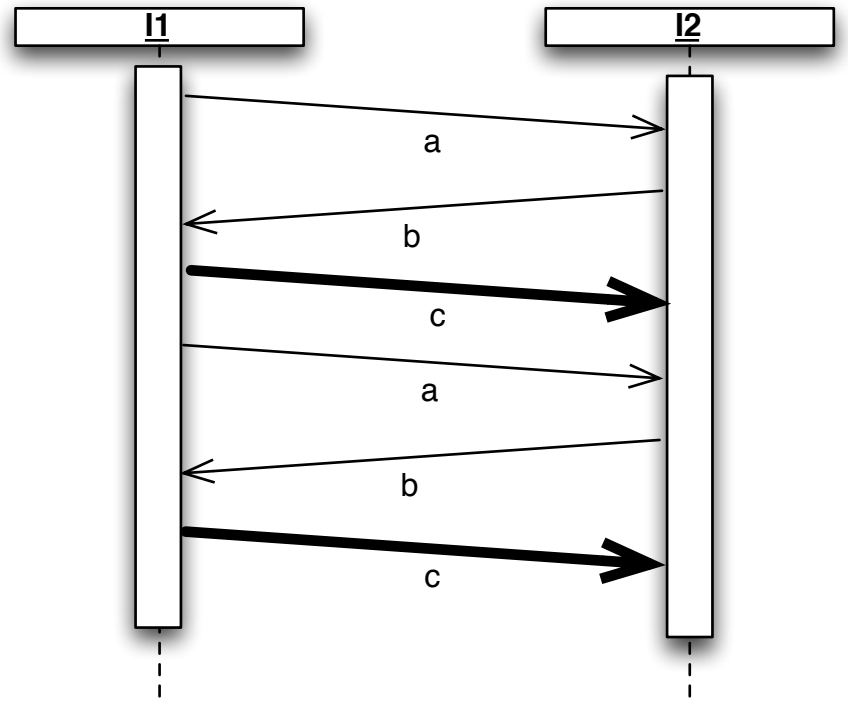
4. Delete from the bSD all the events prior to the join point (these will never be matched), and repeat the algorithm to determine any further join points.

Klein's enclosed part algorithm



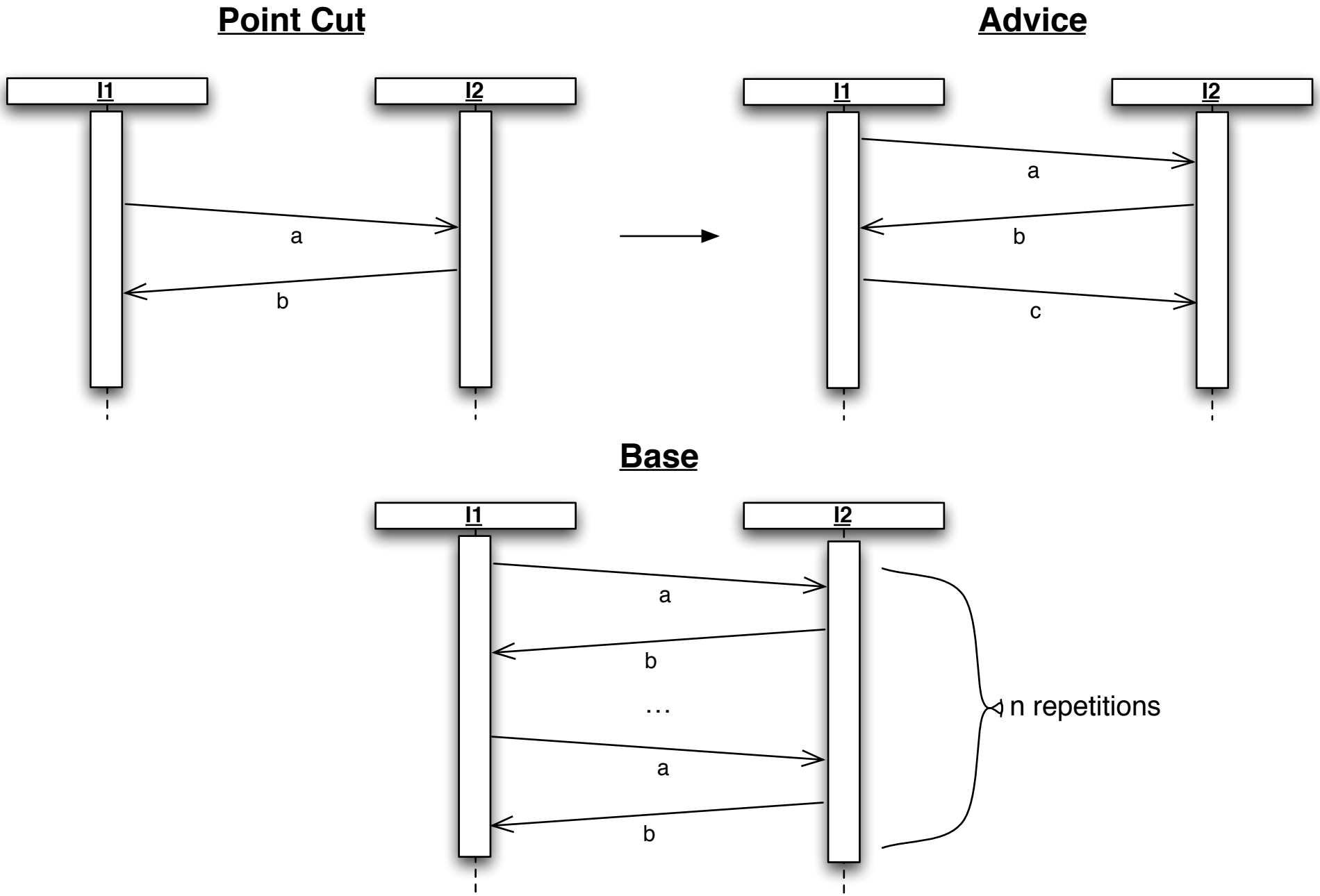
4. Delete from the bSD all the events prior to the join point (these will never be matched), and repeat the algorithm to determine any further join points.

Klein's enclosed part algorithm

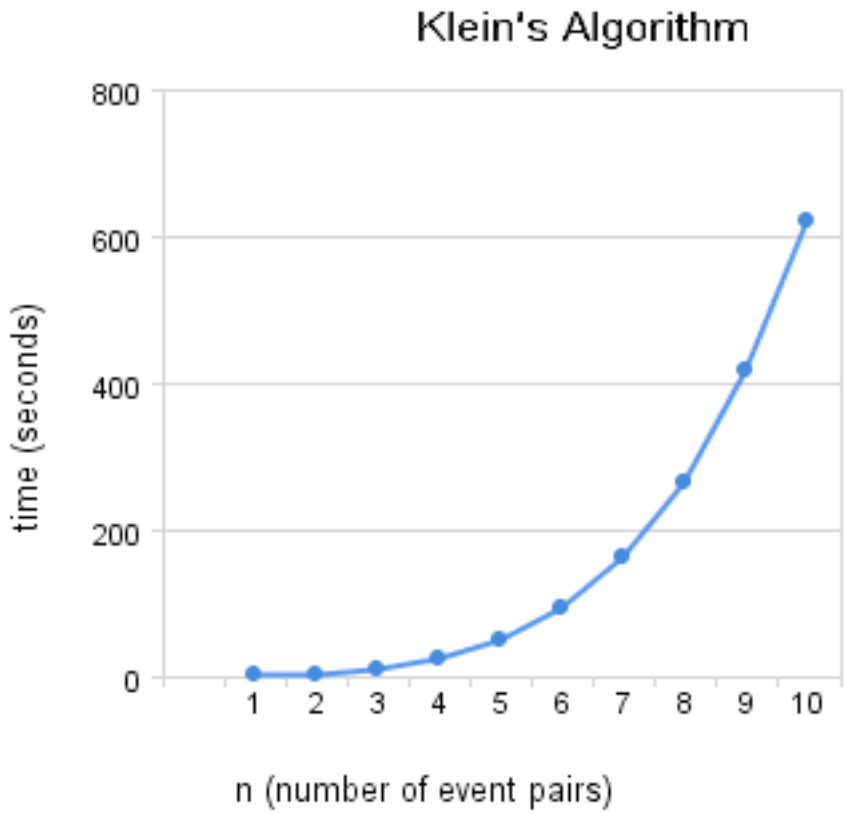


5. Finally, replace the join points in the original bSD with the advice.

Measuring Performance of Klein's Algorithm



Measuring Performance of Klein's Algorithm



(It's worth noting that the test setup, with $O(n)$ matches, adds a factor of n to the timing; so $O(n^4)$ for the test corresponds to a more general $O(n^3)$.)

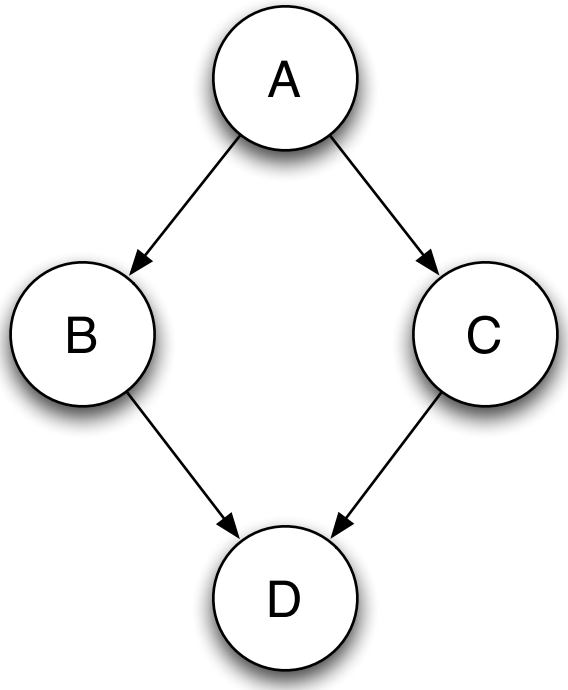
Can we improve on this?

- Main idea: don't discard the ordering information
- Result: no measurable speed-up!?
- …Something else is going on…

Culprit: "Transitive Closure"

- Turns out that the main cost came in **step 5** from the algorithm: delete the events prior to the join point.
- For *n=4*, more than half the time is spent in a single method: `transitiveClosure()`.
- "Prior to" is not as simple as it might seem, because the events in a bSD are only partially ordered.

Partial Ordering



- *A* > *B* > *D* and *A* > *C* > *D*, but B ?? C.
- Corresponds to potentially parallel execution in SDs.

Culprit: "Transitive Closure"

- Klein's strategy:
  - compute transitive closure of all events, finding all events that are either before or after a matched event.
  - determine which of these are, according to the partial ordering, **prior to** matched events, and delete those from the bSD.
- Time complexity of transitive closure is $O(n^3)$, where n is the number of events in the set.

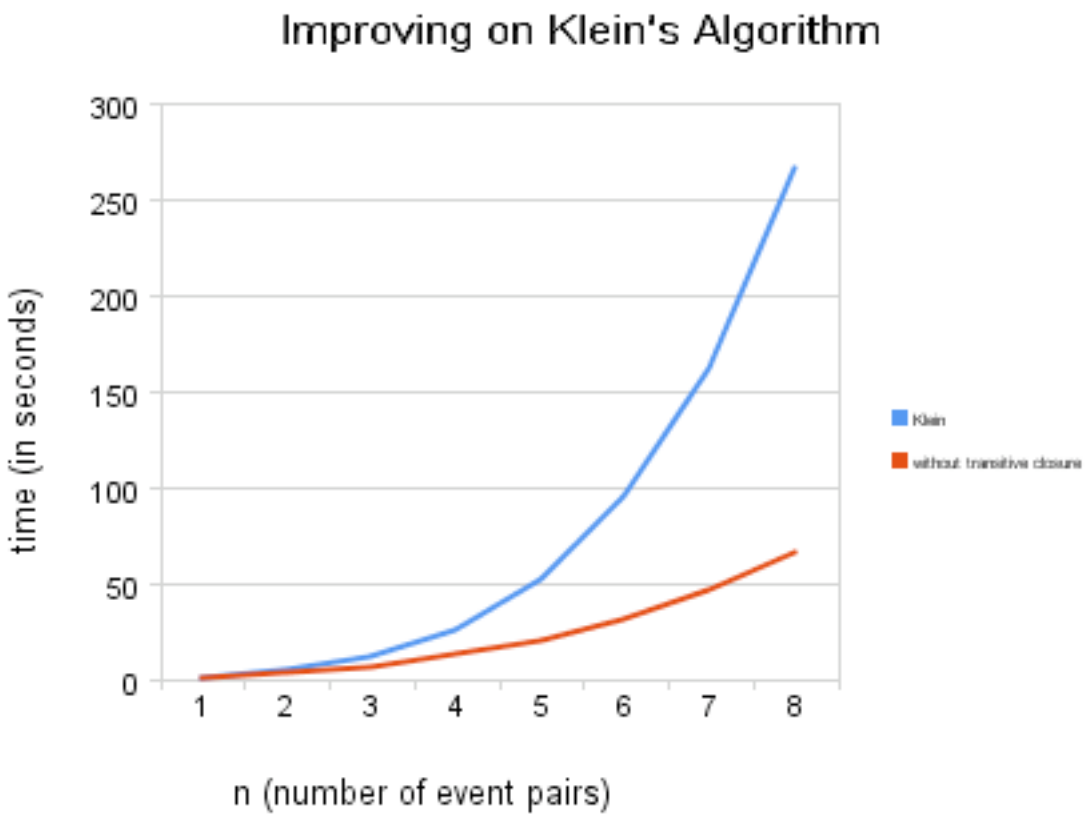
Improving on Klein's Algorithm

- Transitive Closure replacement: `getAllPredecessorsOfEvent()`.
- This just iteratively adds immediate predecessors of each matched event, until no new ones are added (remembering which ones we've already processed).
- amounts to a lazily computed transitive closure, only going backwards in time, and only computing the closure of the matched events.

Improving on Klein's Algorithm

- Assuming matched event set has constant size, worst case time complexity of `getAllPredecessorsOfEvent()`: $O(n^2)$.
- Best case is $O(n)$, for events that have only a small number of predecessors. (Transitive closure strategy, which constructs the entire transitive closure regardless of which event is in question.)
- Could improve on this by having events know about their predecessors, etc., but `getAllPredecessorsOfEvent()` is no longer the bottleneck.

Improving on Klein's Algorithm



- It might be worth trying the experiment in a somewhat different context: what if we just make a large base SD, which only includes a single match (maybe at the end)?

Future improvement possibilities

- Some other areas of his approach appear to be quite inefficient—especially the "left amalgamated sum," which is what's responsible for the actual weaving.
- Rethink certain assumptions in the setup. Do we need partial orderings? (The partial orderings are a big part of the essential complexity of Klein's approach.) Could we do better by restricting further the sequence diagrams allowed for AOM?

Conclusions

- AOM will be necessary to allow efficient use of aspects in software engineering.
- Good weaving algorithms will be necessary if AOM is to be practical.
- Klein's approach, while very flexible, also appears to have some unnecessary complexity. It would be great if we could develop a semantically clearer approach, even if that means giving up a certain amount of flexibility.