

# tagger design

Daniel Jackson

6170 · Lab in Software Engineering

Lecture 18 · October 22, 2001

# topics for today

- what Tagger is and why I wrote it
- design alternatives
- action-based design
- design overview
- design aspects
  - one-shot actions
  - cross-references
  - property maps
  - autonumbering
  - style set view
  - type-safe enums
- process
- conclusions

# what and why

what?

- a text preprocessor for document preparation
- source text in my own markup language
- output in 'tagged text' of Quark Xpress

why?

- unhappy with existing document preparation systems
  - Word too unreliable
  - Framemaker too old and slow
  - TeX too inflexible (and can't use my nifty fonts)

# desiderata

compilation model of TeX

- prepare document in text editor
- symbolic names for special characters
- automatic numbering and cross-refs

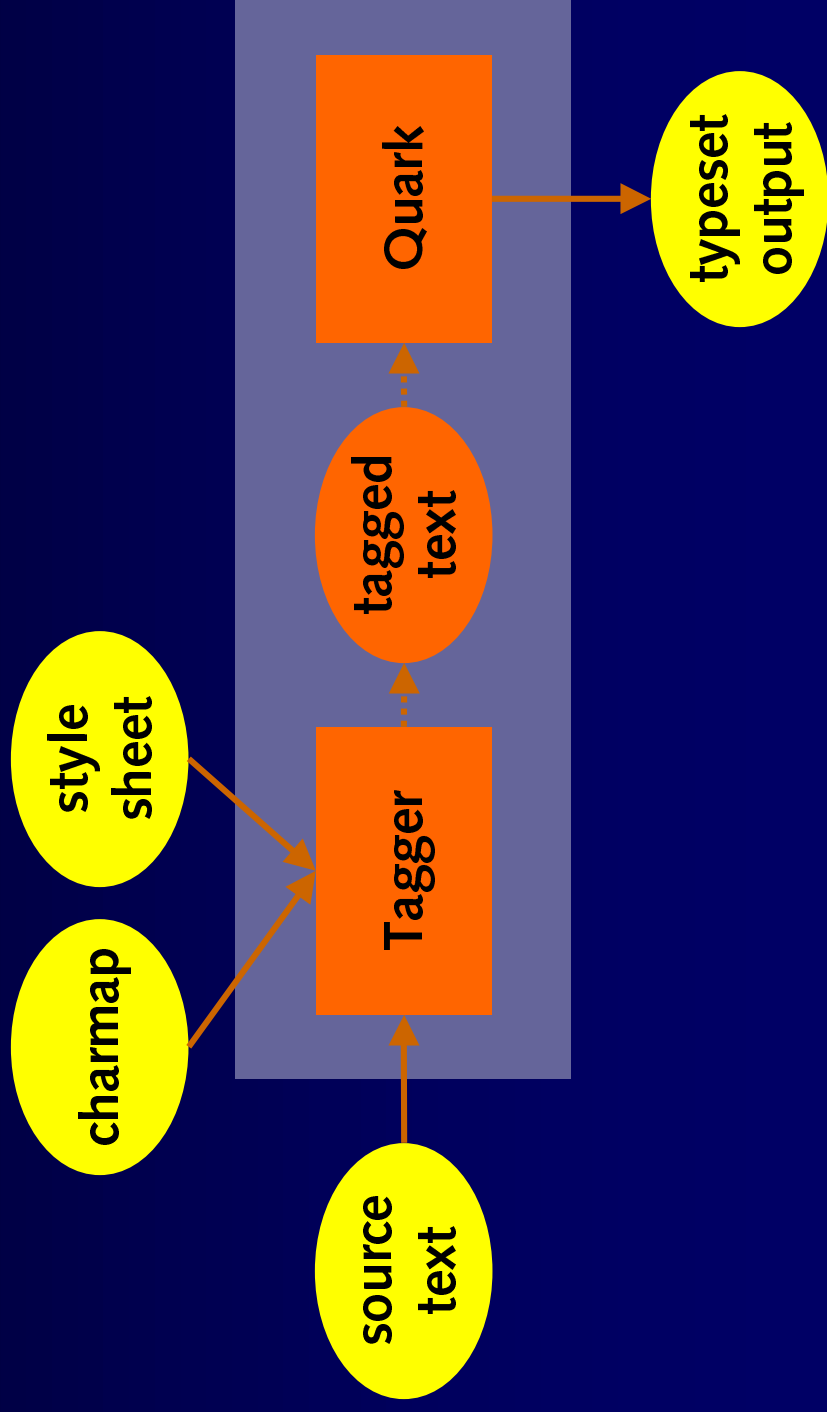
WYSIWYG model of Quark

- high typographic quality
- easy to adjust choice of fonts, spacing, page layout, etc

# solution: tagger

combine

- typographic quality & flexibility of Quark Xpress
- with convenience of textual input of TeX



# sample input

```
\loadchars{maps}\standard.map.txt}
\loadchars{maps}\lucmatharr.map.txt}
\loadstyles{example.styles.txt}

\title Tagger Example

Daniel Jackson//
MIT Lab For Computer Science

\section Introduction

\subsection Some Cool Things
```

```
Here are some things
[\cite{dnj}] you can do:

\point Make _pretty_ documents.

\point Typeset formulas  $a +$ 
b) = c $\$$  and use symbols like
\arrowdblse.

\section References

\ref{tag{dnj}}
Jackson, Daniel. Tagging for
Profit and Pleasure. 2001.
```

# sample style sheet

```
<style:title><next:author>  
<style:author><next:section>  
<style:section><next:body><counter:1><separator:.\><trailer: >  
<style:body><next:body>  
<style:subsection><next:body><parent:section>  
  <counter:a><separator:.\><trailer: >  
<style:point><next:noindent><leader:\periodcentered >  
<style:ref><next:ref><counter: 1><leader:[><trailer:] >
```

# sample character map (extract)

<char:arrowdblne><font:LucidNewMatArrT><iindex:99>

<char:arrowdblsw><font:LucidNewMatArrT><iindex:100>

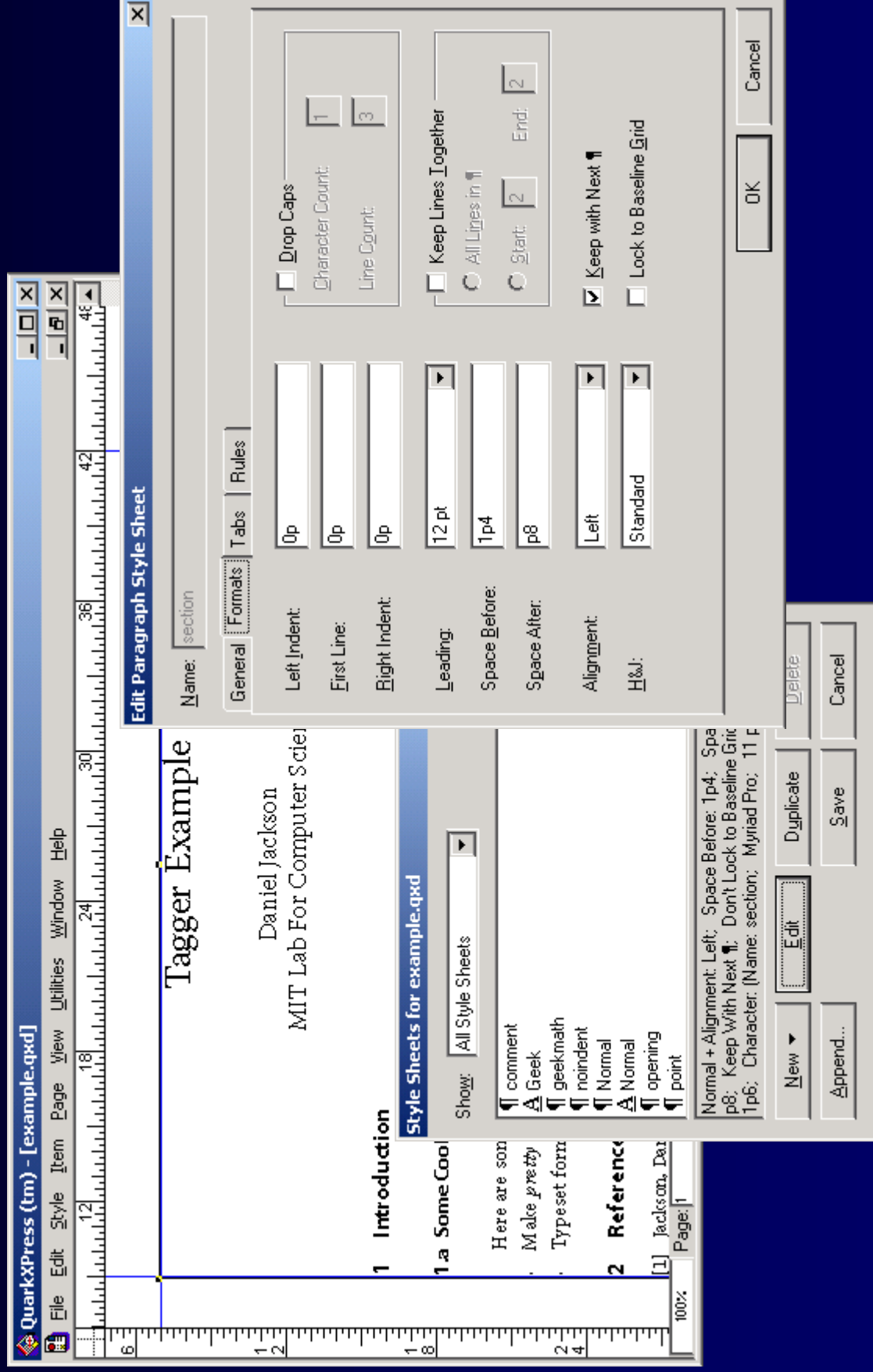
<char:arrowdblse><font:LucidNewMatArrT><iindex:101>

<char:arrowdblleftneg><font:LucidNewMatArrT><iindex:102>

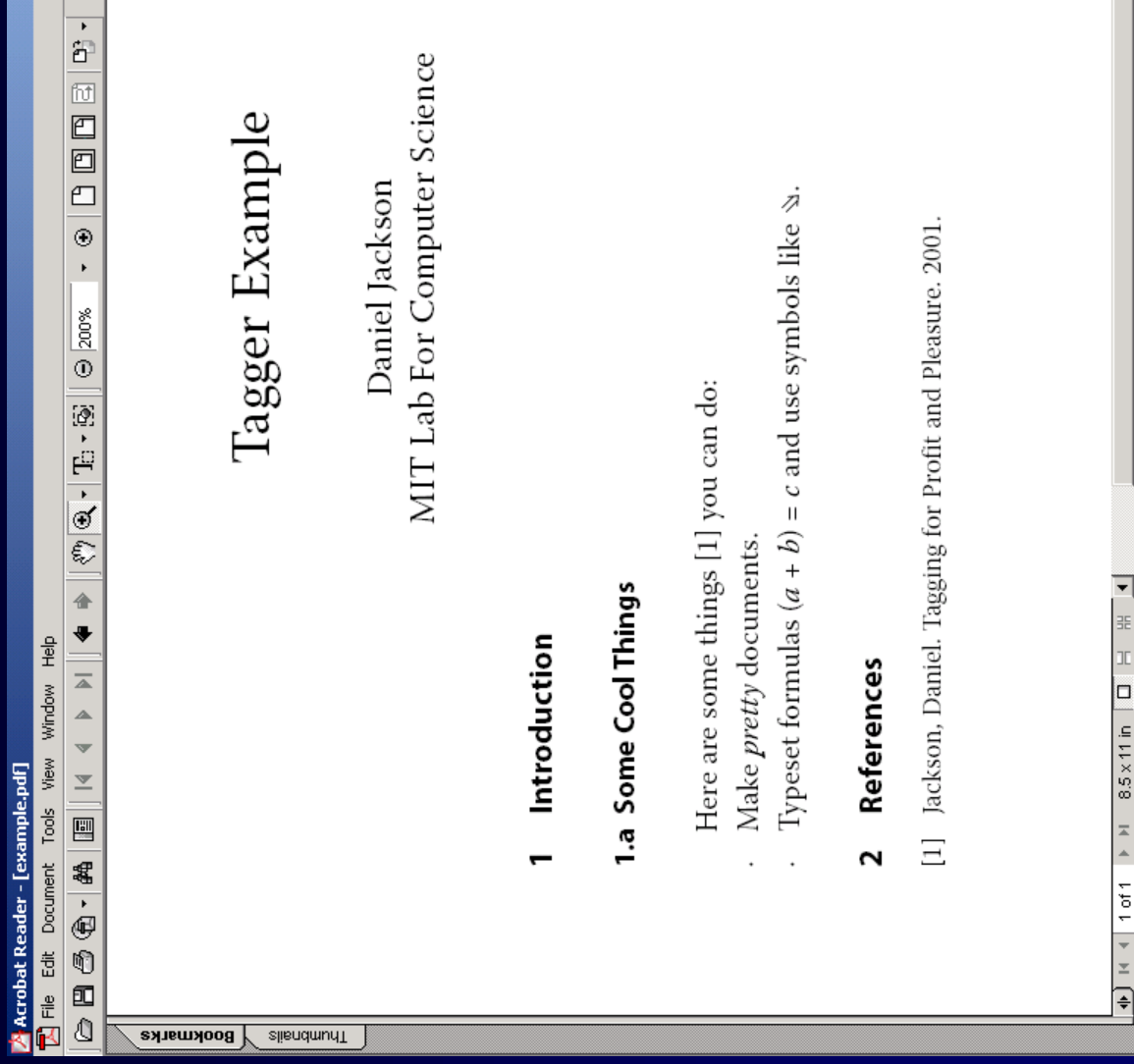
# sample tagger output

```
@title:Tagger Example
@author:Daniel Jackson<\n>MIT Lab For Computer Science
@section:1 Introduction
@subsection:1.a Some Cool Things
@body:Here are some things [1] you can do:
@point:<\#183> Make <I>pretty<I> documents.
@point:<\#183> Typeset formulas (<I>a<I> + <I>b<I>) = <I>c<I> and
use symbols like <f"LucidNewMatArrT"><\#101><f$>.
@section:2 References
@ref:[1] Jackson, Daniel. Tagging for Profit and Pleasure.
2001.
```

# sample Quark style sheet



# sample typeset output



Some things [1]  
documents.  
ulas ( $a + b$ )

Critical Update Notifi  
Click the icon to learn about c  
for your computer.

# design alternative #1: token subclass

```
abstract class Token {}

class ParagraphCommand extends Token {
    void generateOutput () {...}
}

class AlphabeticSequence extends Token {
    void generateOutput () {...}
}
```

problems:

- too many subclasses
- modes (eg, math mode) split across classes

# design alternative #2: case statement

```
class Token {int type; ...}
class TokenType {static int final PARACMD = 1; ...}

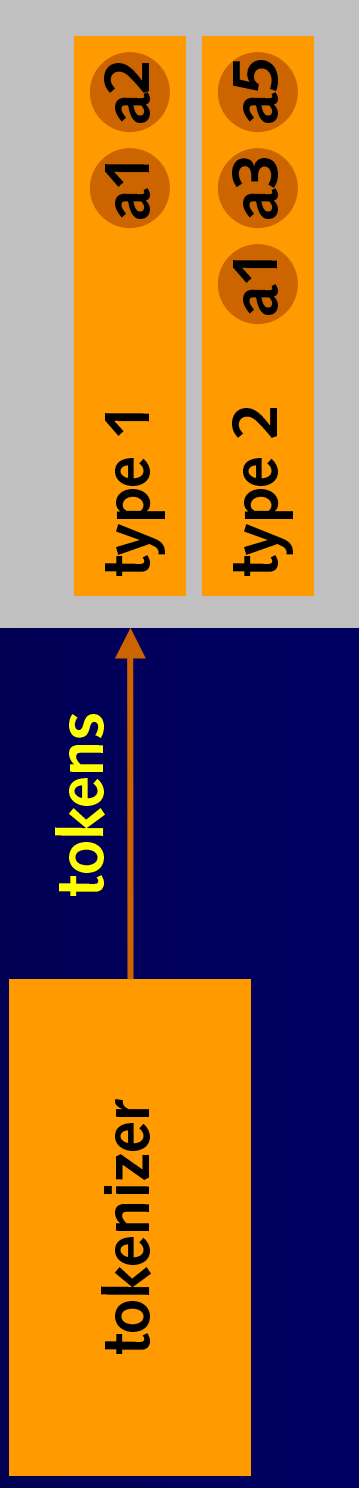
void generateOutput (Token t) {
    if (t.type == TokenType.PARACMD) then
        ...
    else if (t.type == TokenType.ALPHASEQ then
        ...
```

problems:

- huge monolithic piece of code
- with lots of global variables and interactions
- could also implement this with Visitor pattern

# action-based design

- source converted to stream of tokens
- each token consumed by engine
- engine executes all actions registered for token type
- actions may read & write files, register/deregister actions



# what does the code look like?

```
Action plaintextAction = new Action () {
    public void perform (Token t) {
        generator.plaintext (t.arg);
    }
};
registerByType (plaintextAction, TokenType.ALPHABETIC);
registerByType (plaintextAction, TokenType.NUMERIC);
registerByType (plaintextAction, TokenType.WHITESPACE);

registerByType (new Action () {
    public void perform (Token t) {
        errorStream.println ("... done");
    }},
    TokenType.ENDOFSTREAM);
```

# why are actions nice?

- flexible, context-sensitive behaviour
- without one huge, monolithic piece of code
- easy to turn behaviours on & off
- can have subengines: eg, for numbering strings

example: math mode

- $\$$  starts and ends ‘math mode’: puts all letters in italics
- implemented as 2 actions
  - for  $\$ token: dollar\_action$
  - for  $alphabetic string token: italicize\_action$
- $dollar\_action$ 
  - encapsulates mode
  - registers/deregisters  $italicize\_action$

# code for math mode

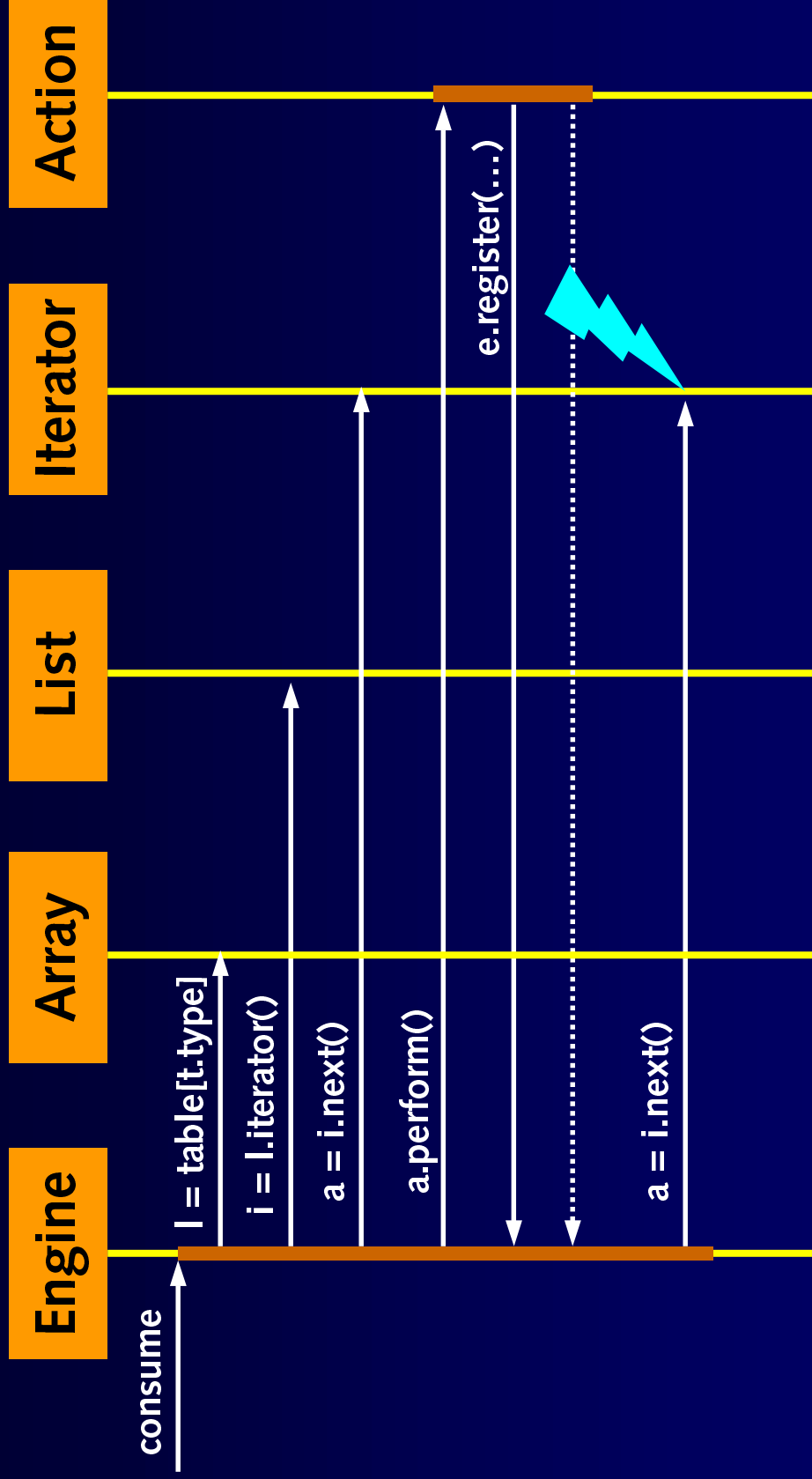
```
registerByType (new Action () {  
    boolean mathModeOn = false;  
    public void perform (Token t) {  
        if (mathModeOn) {  
            mathModeOn = false;  
            registerByType (apostropheAction, TokenType.APOSTROPHE);  
            unregisterByType (primeAction, TokenType.APOSTROPHE);  
            unregisterByType (pushItalicsAction, TokenType.ALPHABETIC);  
            unregisterByType (popItalicsAction, TokenType.ALPHABETIC);  
        } else {  
            mathModeOn = true;  
            ...  
        }  
    }  
}}, TokenType.DOLLAR);
```

# code of engine

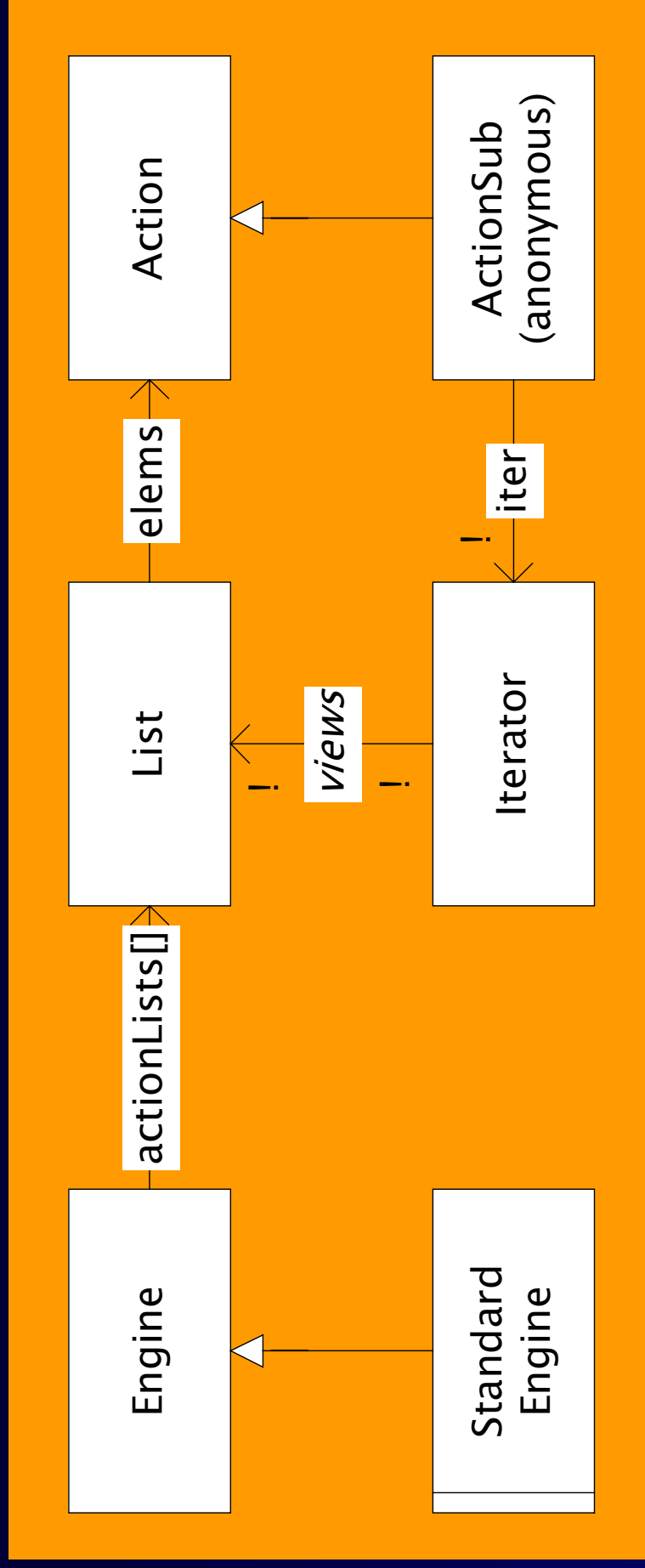
```
private LinkedList [] table;
...
public void register (Action action, int type) {
    actions[type].add (action);
}

public void consume_token (Token token) {
    Iterator i = table[token.type].iterator ();
    while (i.hasNext ()) {
        Action a = (Action) i.next ();
        a.perform (token);
    }
}
```

# failure!



# design aspect #1: one shot actions



how it works

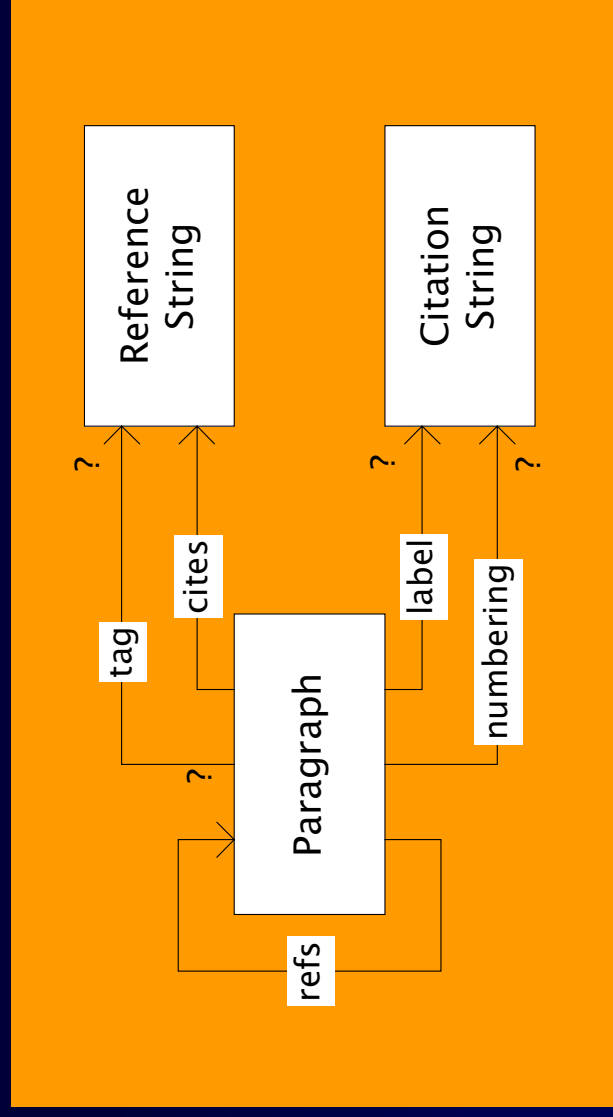
- action removes itself from action list
- by calling remove on iterator passed to Action.perform

# code for engine and one-shot action

```
public void consume_token (Token token) {
    Iterator i = table[token.type].iterator ();
    while (i.hasNext ()) {
        Action a = (Action) i.next ();
        a.perform (token, i);
    }
}

final Action paragraphAction = new Action () {
    public void perform (Token t, Iterator iter) {
        if (t.type != TokenType.PARATYPECOMMAND) {
            generator.newPara (currentPara.styleName);
            String numbers= numbering.getNumberString (...);
            ...
            iter.remove ();
        }
    }
};
```

# design aspect #2: cross references



how it works

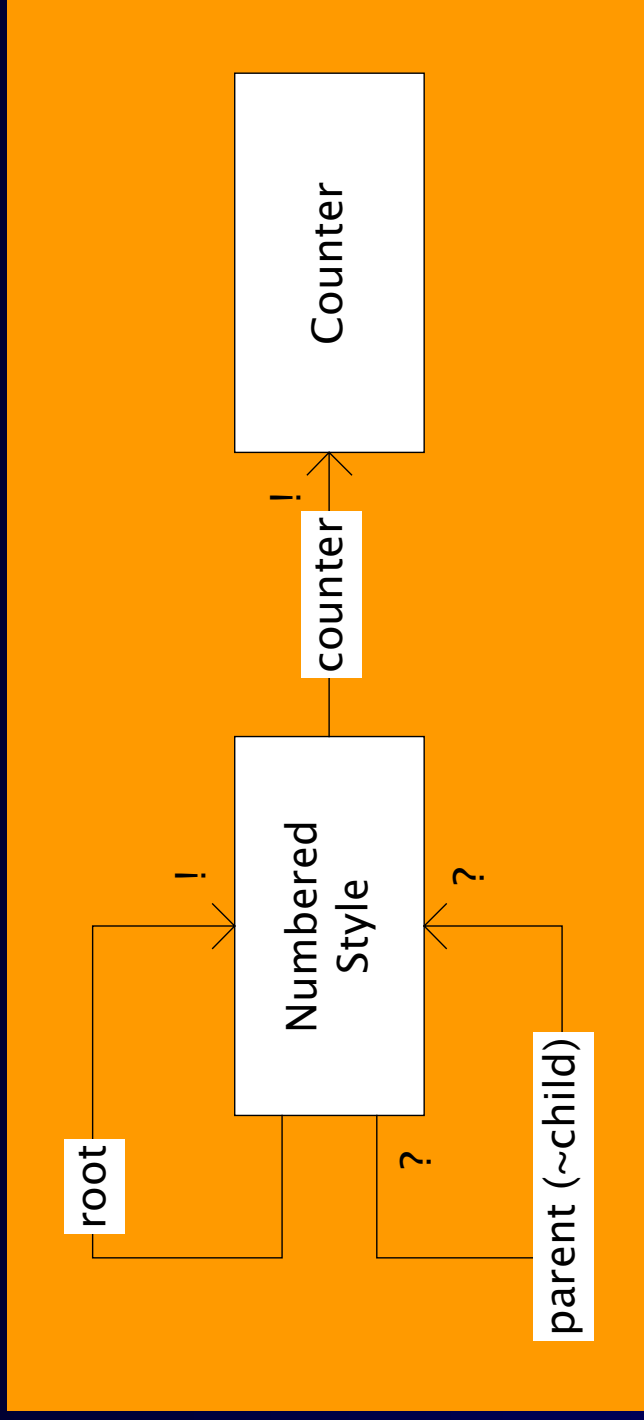
- user may tag each para with symbolic name
- citation elsewhere by symbolic name
- citation string is numbering string or label

# design aspect #3: property maps

style sheets, character maps, cross-ref index files

- all given same syntax
- one parser (PropertyParser)
- one internal representation (PropertyMap)

# design aspect #4: autonumbering



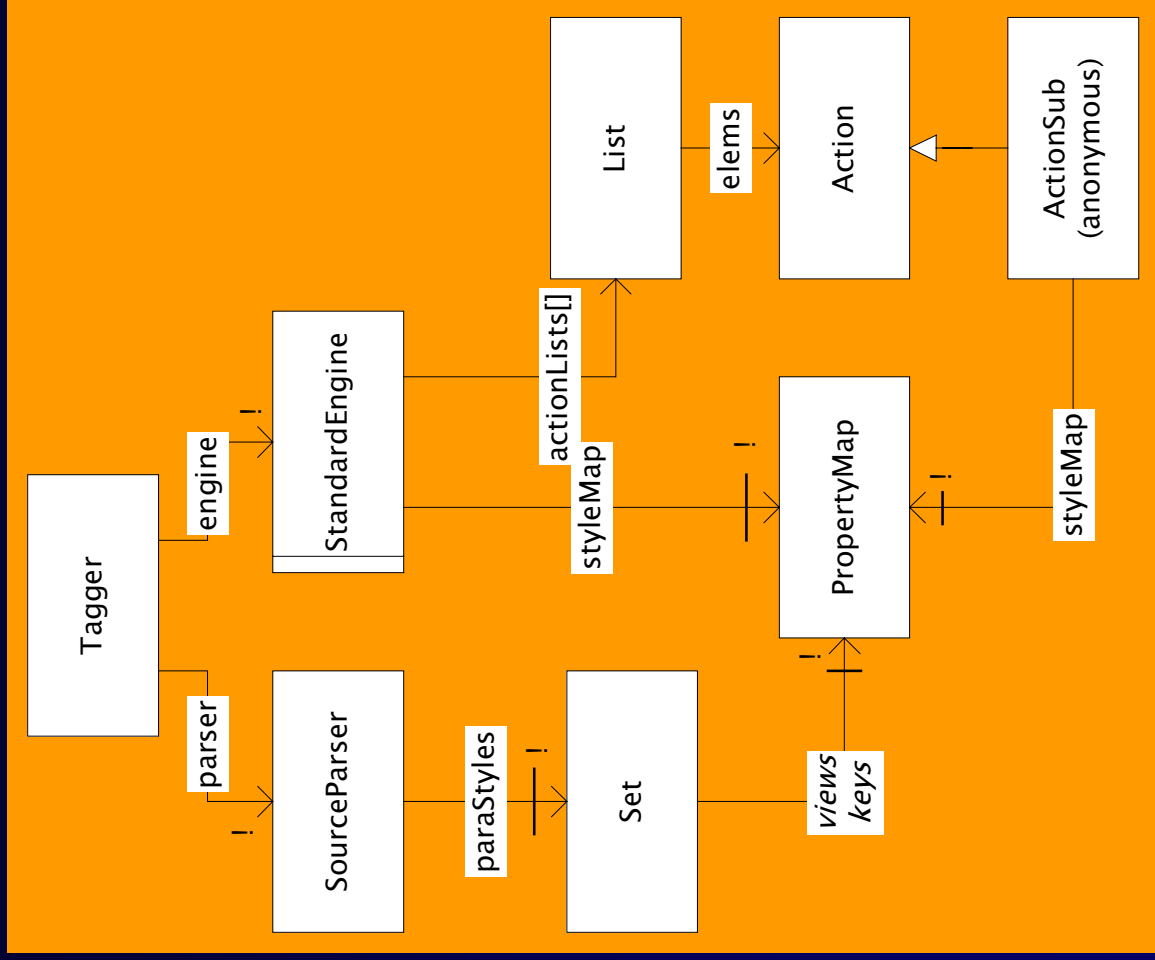
how it works

- style sheet only needs to give
  - parent: enclosing style in hierarchy
  - counter: whether numbered, and how
- Numbering class maintains state
  - and augments property map with child and root

# design aspect #5: style set view

how it works

- Tagger creates empty PropertyMap
- passes map to StandardEngine
- passes set view to SourceParser
- SourceParser looks up strings in set to see if paragraph styles
- StandardEngine updates map when style sheet loaded



# design aspect #6: type-safe enums (1)

```
// the bad way
public final class Format {
    public final static int ROMAN = 1;
    public final static int ITALICS = 2;
    public final static int SUBSCRIPT = 3;
    public final static int SUPERSCRIPT = 4;
}
```

problem

- formats declared in client code as ints
- non-formats (eg, -1) not caught at compile time

## design aspect #6: type-safe enums (2)

```
public final class Format {
    public static Format ROMAN = new Format ("Roman");
    public static Format ITALICS = new Format ("Italics");
    public static Format BOLD = new Format ("Bold");
    public static Format SUBSCRIPT = new Format ("Subscript");
    public static Format SUPERScript =
        new Format ("Superscript");

    private final String name;
    private Format (String name) {this.name = name;}
    public String toString () {return name;}
}
```

# process: how I built it

## early prototyping

- wrote Perl scripts and experimented with Quark tagged text

## Java development

- spent a few days designing; OM and MDD sketches
- developed code hand-in-hand with language
- no systematic testing; just tested during usage

## refactoring

- improved structure (eg, typesafe enums)
- completed specs for all public methods
- wrote crude regression testbed to guard against new bugs

# conclusions

different systems have different quality needs

- here, ad hoc testing reasonable
- would need systematic testing for release

action-based idiom

- powerful and clean, most of the time
- now doing object model analysis of action-machines

pattern density

- more typical than JUnit in this respect
- effort spent mainly on user-defined types and idioms