Software Engineering 301:
Software Analysis and Design

Refactoring

Agenda

- Basic concepts
- Standard refactorings
- Tool support
- Pitfalls

©2009–2016 R .J. Walker. All rights reserved.

What if your design is wrong?

- If it is just a design on paper/on a blackboard/in your head …
  - change it
- However, what if you already have an implementation or a partial implementation?
  - you can still change it, but this is less straightforward

©2009–2016 R .J. Walker. All rights reserved.

Consider an example

- JHotDraw drawing application
- Permits objects to be drawn on a canvas
- Overlapping objects have a Z-order
  - Z-axis comes out of the screen
- Reorder objects through …
  - Send to back
  - Bring to front

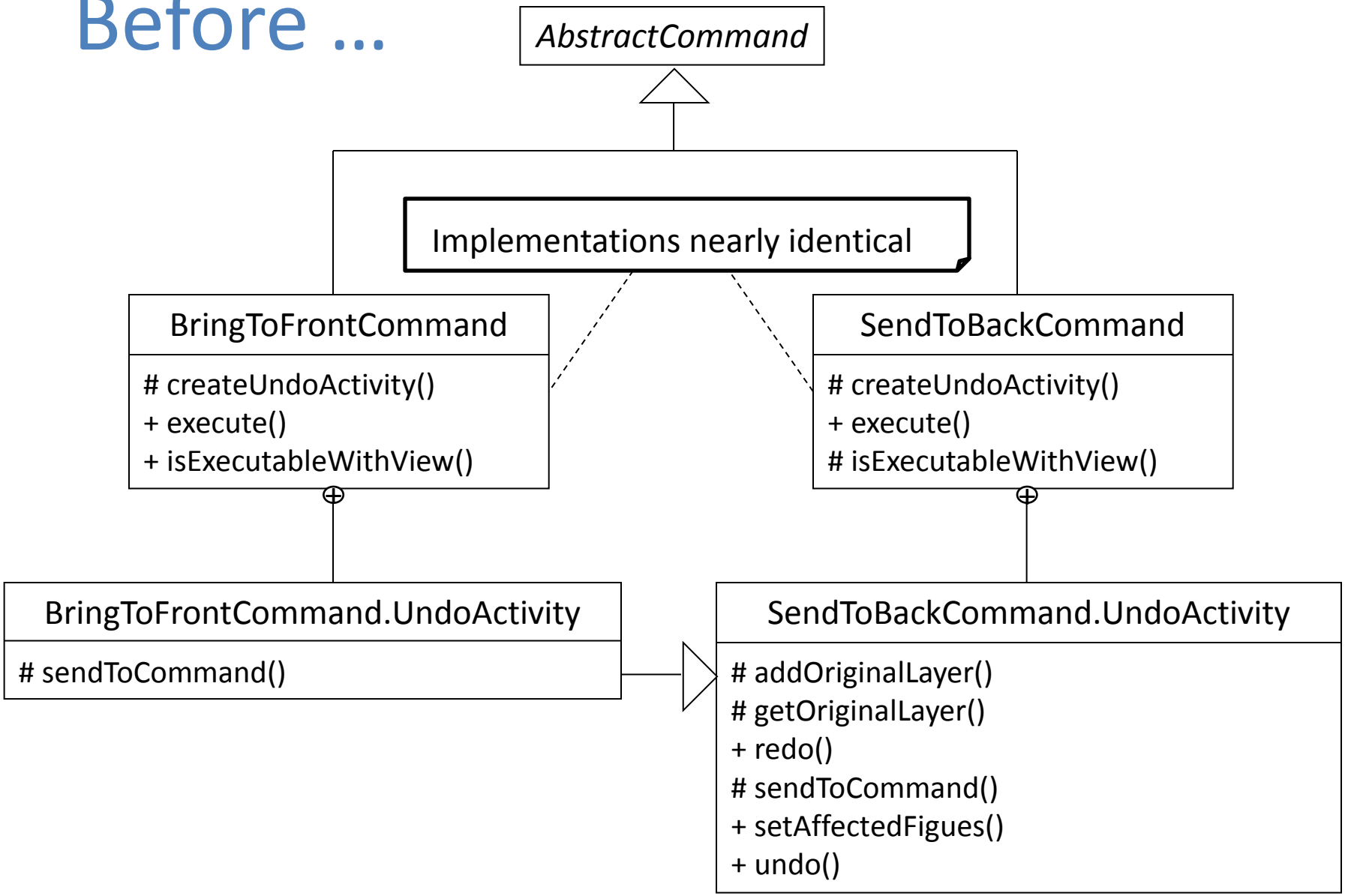
©2009–2016 R .J. Walker. All rights reserved.

JHotDraw

- Consider the classes SendToBackCommand and BringToFrontCommand …

©2009–2016 R .J. Walker. All rights reserved.

Before …



©2009–2016 R .J. Walker. All rights reserved.

Why is this a problem?

- Comprehensibility
  - Is BringToFrontCommand.UndoActivity a specialized kind of SendToBackCommand.UndoActivity?
    - No, but that's what the generalization relationship there implies
    - Confusion will result
  - If SendToBackCommand needs to be changed, what should be the effect on BringToFrontCommand?
- Why was the generalization used?
    - For convenience: no need to reimplement all those methods
    - In other words, this is a bad design

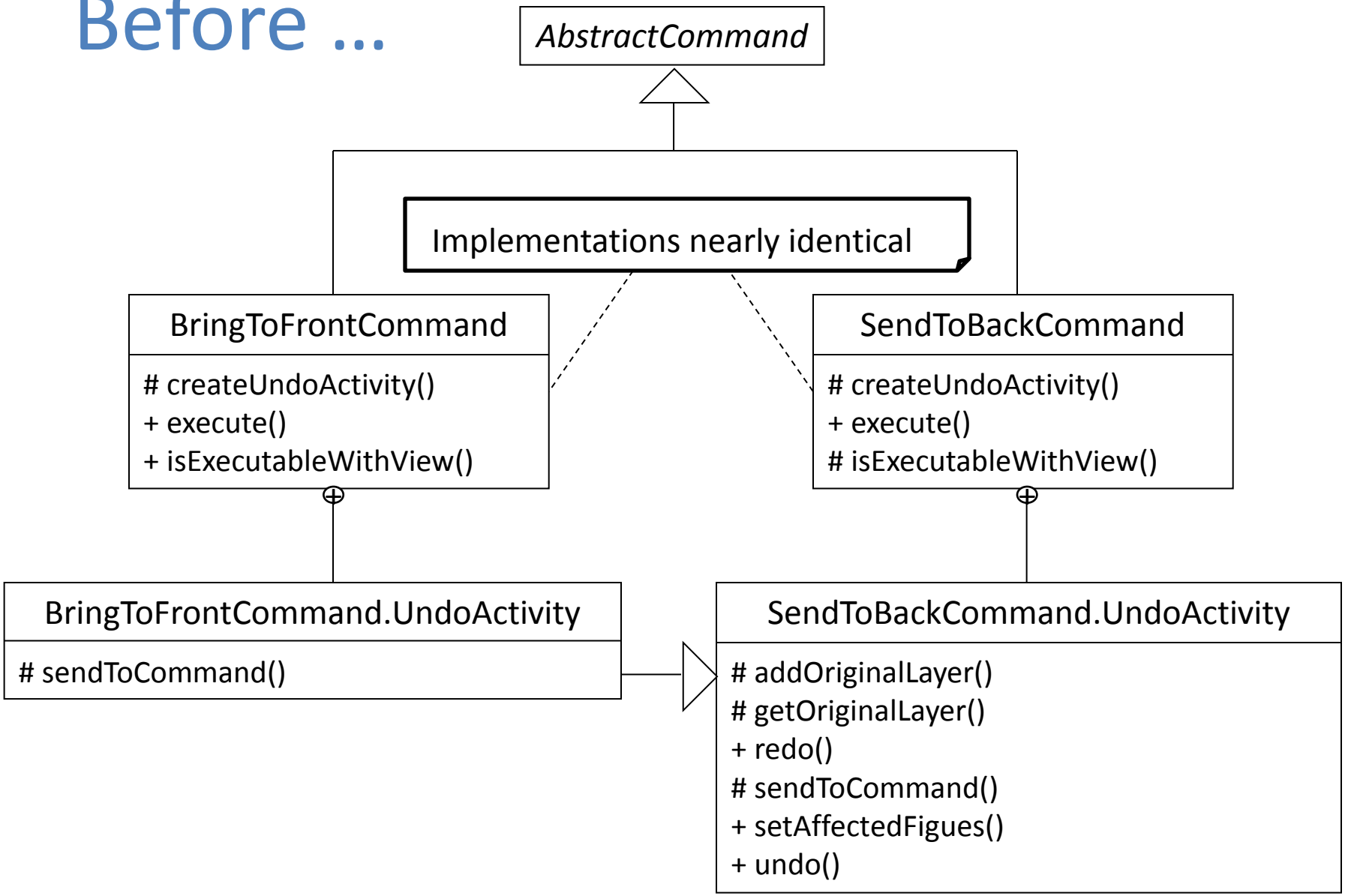
©2009–2016 R .J. Walker. All rights reserved.

Why is this a problem?

- Maintainability
  - If two new classes (BringForwardCommand and SendBackwardCommand) are created for moving one layer at a time, how should their associated UndoActivity classes be organized?
    - Mistakes are likely

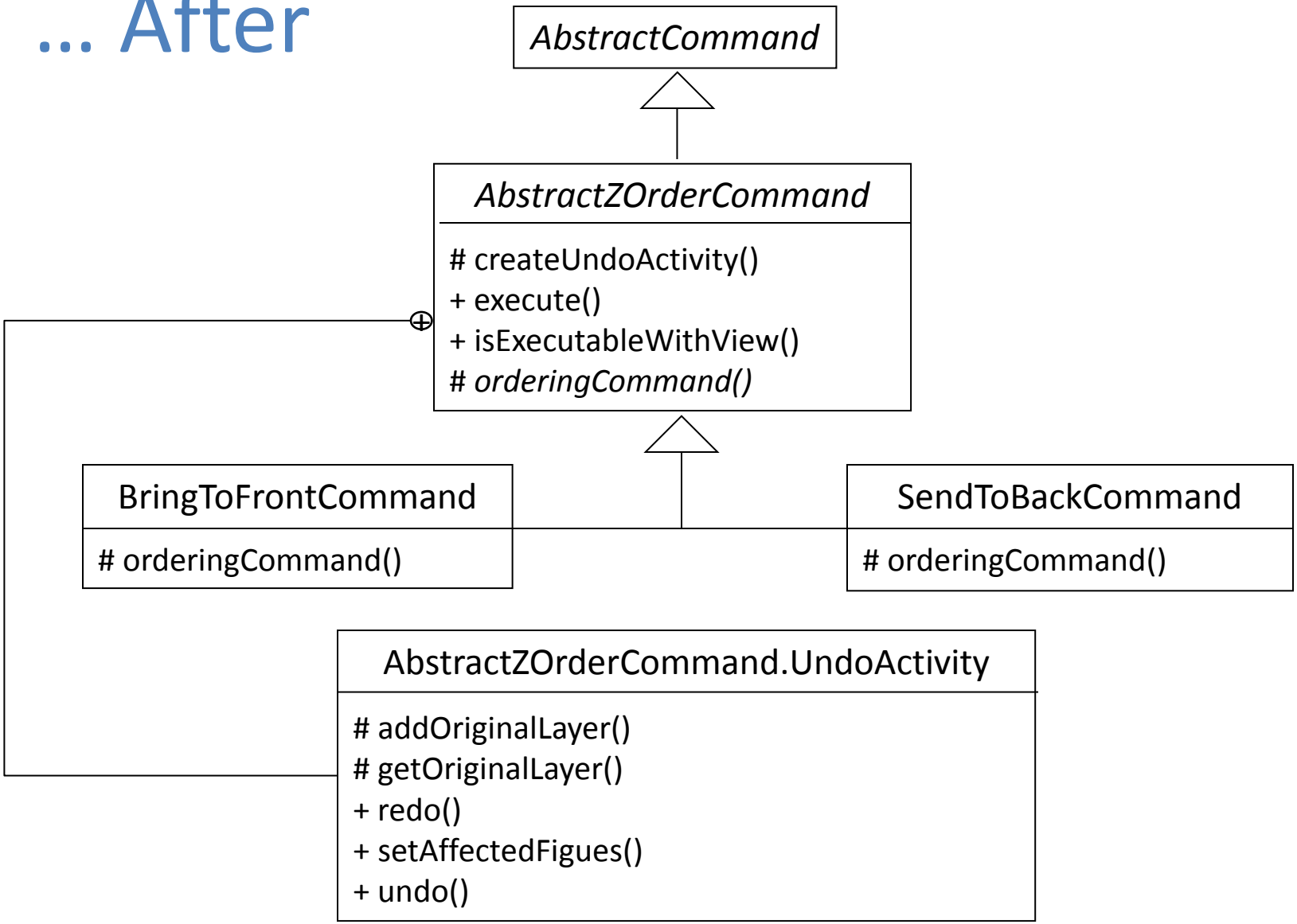
©2009–2016 R .J. Walker. All rights reserved.

Before …



©2009–2016 R .J. Walker. All rights reserved.

… After



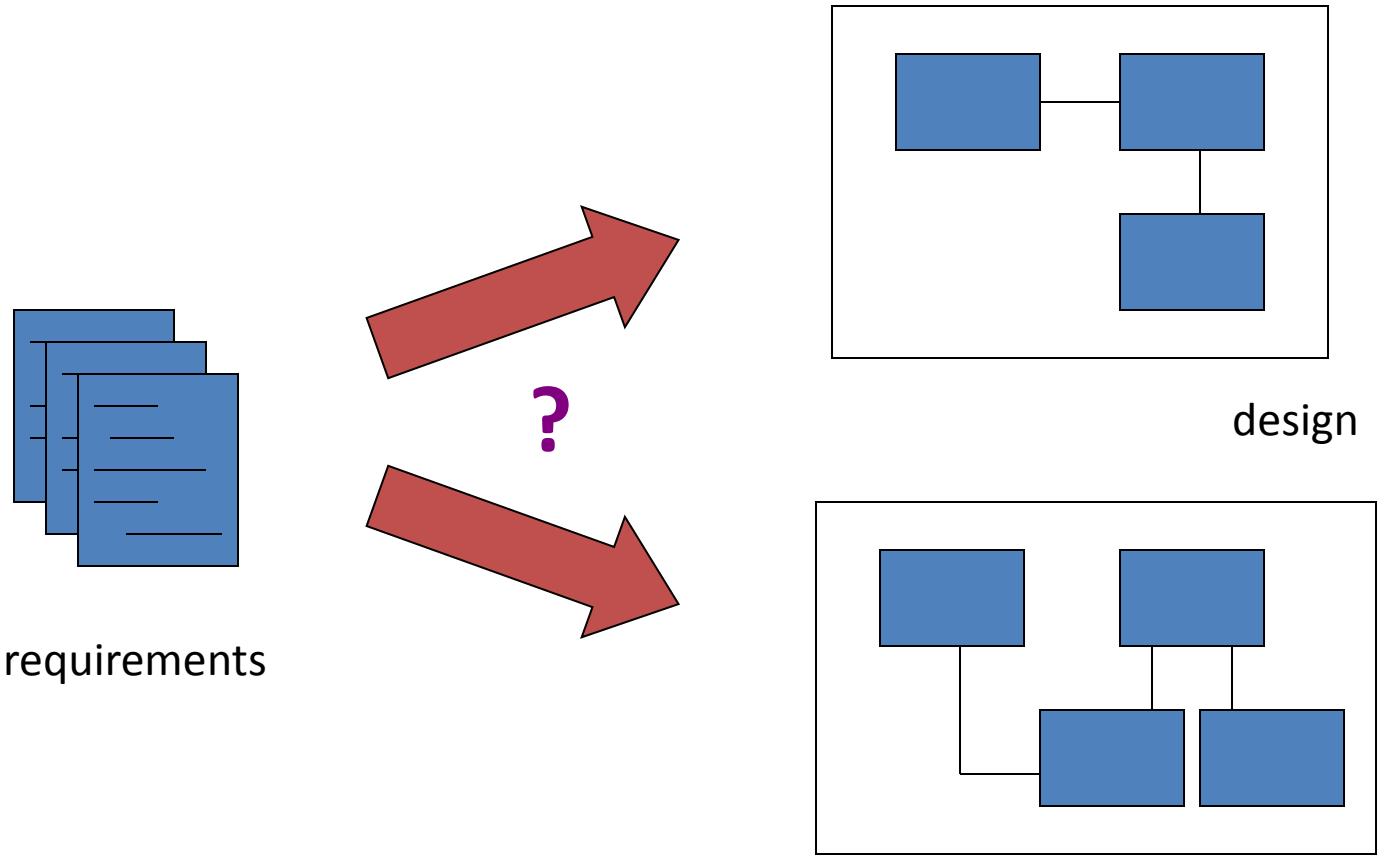
©2009–2016 R .J. Walker. All rights reserved.

Another example

- For the vending machine software
  - Approach #1: lots of abstraction, quite modular
  - Approach #2: little abstraction, lack of modularity
- If you were stuck with Approach #2, it would be nice to be able to transform it to Approach #1 without just rewriting the whole system
  - In practice, this could be very hard to accomplish

©2009–2016 R .J. Walker. All rights reserved.

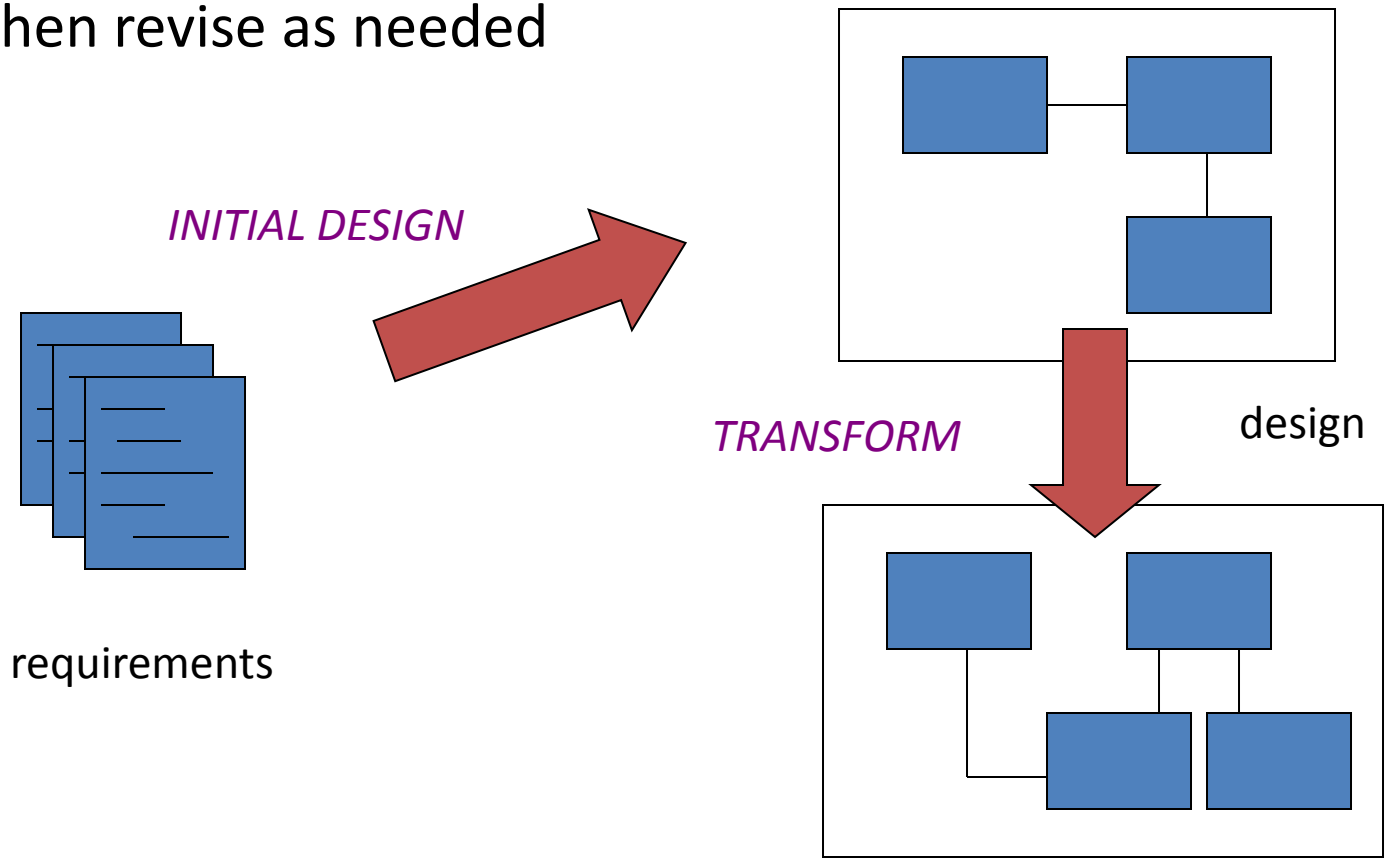
Alternative designs



©2009–2016 R .J. Walker. All rights reserved.

Restructuring

- Choose then revise as needed



©2009–2016 R .J. Walker. All rights reserved.

Refactoring

- The process of changing a software system
  - in such a way that it does not alter the externally-visible behaviour of the code
  - yet improves its internal structure
- But many changes *will* alter externally-visible behaviour
  - Especially if we're not careful

©2009–2016 R .J. Walker. All rights reserved.

Agenda

- ~~Basic concepts~~
- Standard refactorings
- Tool support
- Pitfalls

©2009–2016 R .J. Walker. All rights reserved.

Standard refactorings

- There are some refactoring operations that can be applied in many places
  - As opposed to performing a particular restructuring on particular code
- Whole catalogues devoted to these
  - More identified all the time
- We'll look at a few examples

©2009–2016 R .J. Walker. All rights reserved.

Rename

- Names matter
  - fix them when they fail to express purpose clearly
- Can be applied to:
  - methods, fields, local variables, types, packages, …
- References to the named element can also be updated (and usually should be)

©2009–2016 R .J. Walker. All rights reserved.

Rename

- What if the name is already being used?
  - Obvious problem sometimes
  - Can't have two methods with identical signatures
- Possible, but poor practice in other situations
  - Hiding a local variable in an outer scope (compilation error in Java)
  - Hiding a field of the same name from a superclass
- What about polymorphism?
  - Need to worry about super- and subclasses
- What if it forms part of external API?
  - Mark old name as "deprecated"; yuck!

©2009–2016 R .J. Walker. All rights reserved.

Pull Up

- You realize that each subclass of a given class implements the same method
  - eliminate the redundant code by moving these methods to the superclass
- Need to ensure that the methods are identical
  - If not, need to modify each to pull out the common functionality: not meaning-preserving
    - Probably beyond abilities of a refactoring tool suite
- Sometimes, you might not want that method in the superclass
  - Insert an additional, abstract class between?

©2009–2016 R .J. Walker. All rights reserved.

Inline

- Sometimes the implementation of a method is as obvious as its name
  - Extra level of indirection not helpful
  - Performance issues to consider
- Replace calls to the method with the implementation itself
- Beware: simple methods can be helpful for evolvability!

©2009–2016 R .J. Walker. All rights reserved.

Inline

- What if the implementation accesses private fields or methods?
- What if you end up with “magic numbers” replicated all over the code?
  - Consider replacing the magic number with a static final field reference
- This refactoring can be abused badly, leading to bloated source code and replicated errors

©2009–2016 R .J. Walker. All rights reserved.

Add Parameter

- Decide that a method implementation needs an extra piece of information
  - Pass it as an argument
- Where does that argument come from?
  - Sometimes a default value can be used
  - Sometimes, the extra argument needs to come from the caller of the caller of the caller …
    - parameters end up being added all the way up the call chain
- What about super- and subclasses?

©2009–2016 R .J. Walker. All rights reserved.

Add Parameter

- Big concern is that long parameter lists are dangerous
  - The more parameters, the less likely that developers will understand all the options
  - Demands a lot of excess baggage in situations where flexibility not needed
- Can lead to widespread, non-trivial modification

©2009–2016 R .J. Walker. All rights reserved.

Agenda

- ~~Basic concepts~~
- ~~Standard refactorings~~
- Tool support
- Pitfalls

©2009–2016 R .J. Walker. All rights reserved.

Tool support

- Software tools can automate the tedious parts of many refactorings
- Some transformations would not alter the behaviour but are beyond the ability of a tool to perform correctly
  - Fundamental limitations that cannot be overcome
- Tools generally assume that you control all the code, so watch out!
  - Are you modifying a published interface?

©2009–2016 R .J. Walker. All rights reserved.

Lexing

- Starts from a stream of characters
- Identifies keywords, numbers, whitespace, comments, …
  - usually specified via regular expressions
- Groups important characters into tokens
- (Usually) discards whitespace & comments

©2009–2016 R .J. Walker. All rights reserved.

Lexing

- Input (character stream):

```
package ca.ucalgary.lsmr.decaf;

public abstract class ASTNode
   implements ASTNodeI { // a comment
     public void
       accept(AbstractNodeVisitor visitor) {
     }
}
```

©2009–2016 R .J. Walker. All rights reserved.

Lexing

- Output (tokens):

```
package ca.ucalgary.lsmr.decaf;

public abstract class ASTNode
   implements ASTNodeI { // a comment
     public void
        accept(AbstractNodeVisitor visitor) {
     }
}
```

WHITESPACE & COMMENTS IGNORED HERE

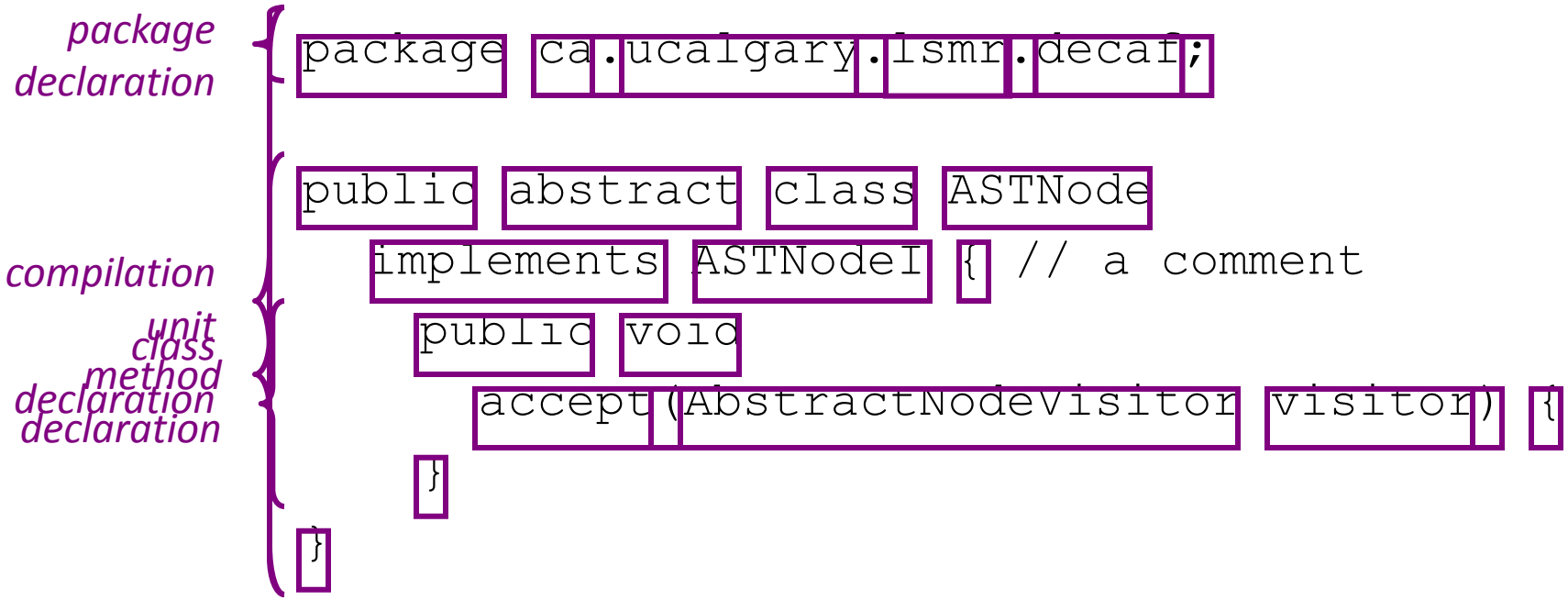
©2009–2016 R .J. Walker. All rights reserved.

Parsing

- Identifies specific patterns in token sequences
- Groups tokens into syntactically meaningful constructs
  - e.g.: ClassDeclaration: [Modifier*] “class” Identifier [“extends” Name] [“implements” Name (“,” Name)*] ClassBody
- Output usually organized as a tree
  - abstract syntax tree (AST)

©2009–2016 R .J. Walker. All rights reserved.

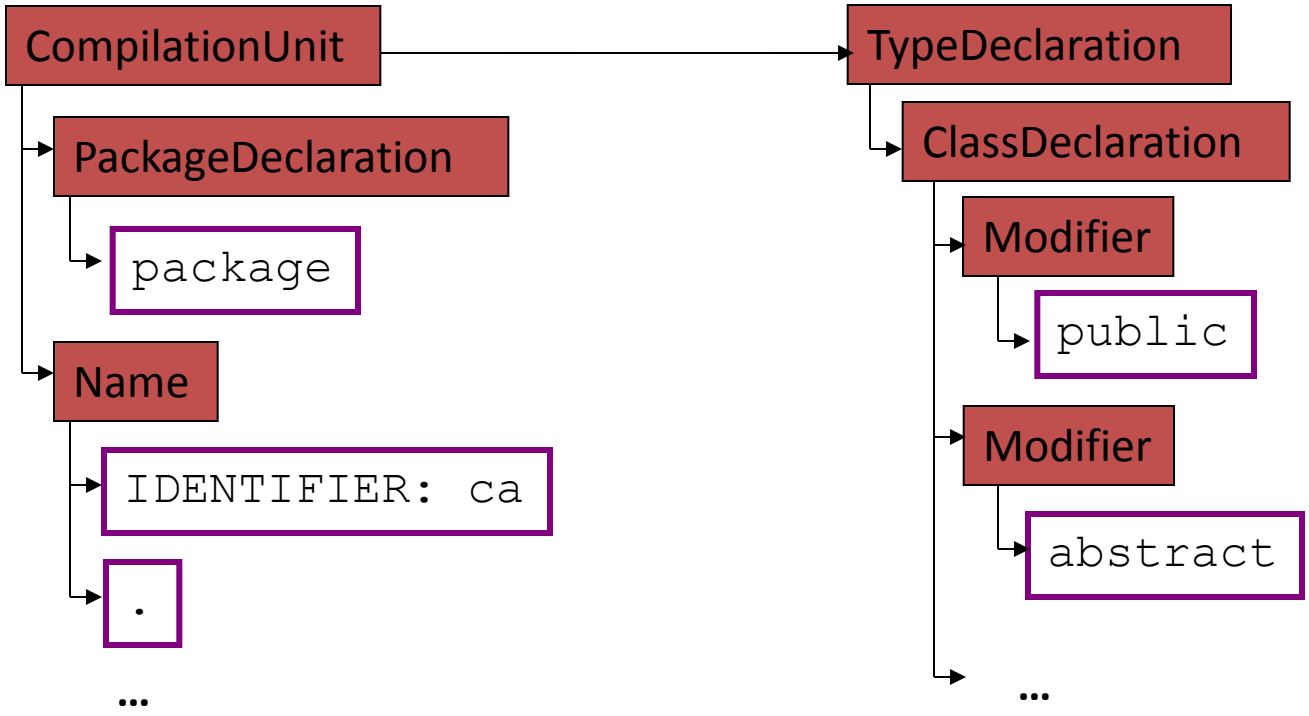
Parsing



©2009–2016 R .J. Walker. All rights reserved.

Parsing

- Output of a (possible) parser



©2009–2016 R .J. Walker. All rights reserved.

Semantic information

```
package ca.ucalgary.lsmr.decaf;

public abstract class ASTNode
   implements ASTNodeI { // a comment
     public void
      accept(AbstractNodeVisitor visitor) {
     }
}
```

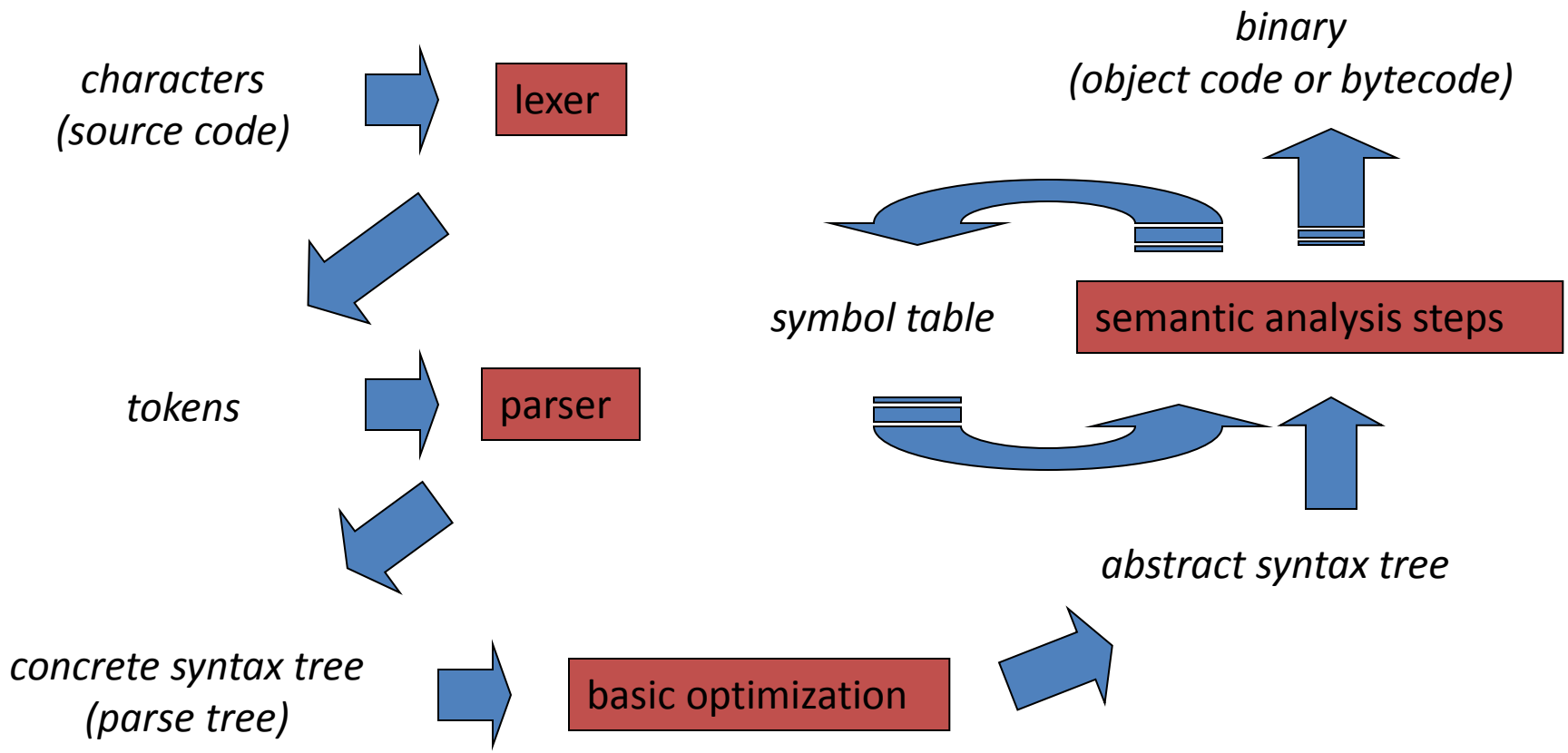
declared names

referenced names

reference resolution determines what these names point to

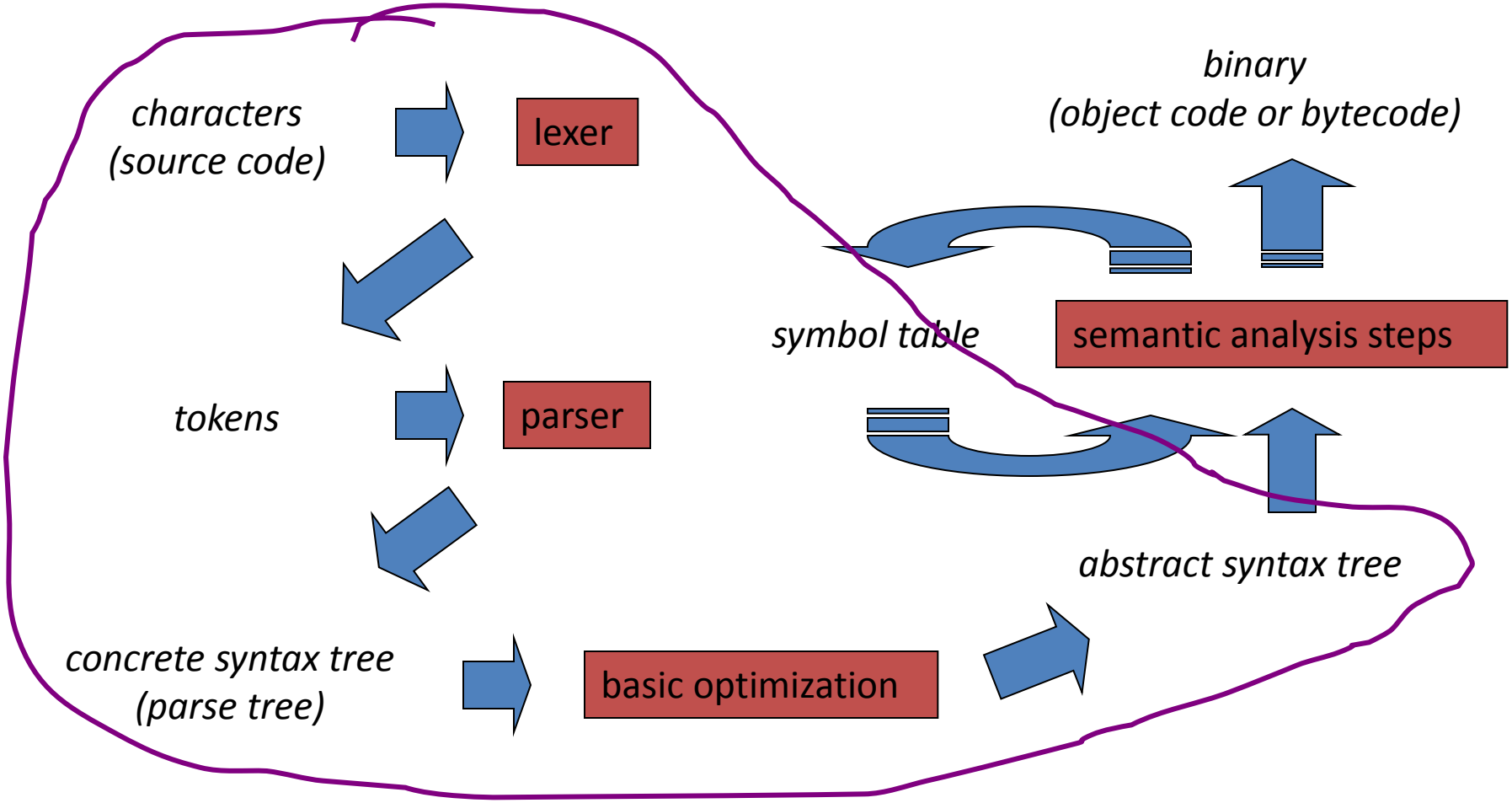
©2009–2016 R .J. Walker. All rights reserved.

Compiler overview



©2009–2016 R .J. Walker. All rights reserved.

Other development tools



©2009–2016 R .J. Walker. All rights reserved.

Refactoring tools

- Developer indicates element to change, and kind of change
- Tool constructs ASTs for the program
- Tool locates the node that represents the element to change
- Tool determines other elements that reference it
- Tool determines where/how the element would move/change
- Tool determines whether these consequences would violate any constraints for that refactoring kind
  - e.g., for rename, the new name cannot collide with an existing name

©2009–2016 R .J. Walker. All rights reserved.

Agenda

- ~~Basic concepts~~
- ~~Standard refactorings~~
- ~~Tool support~~
- Pitfalls

©2009–2016 R .J. Walker. All rights reserved.

Problem

- In constructing a refactoring tool, it is not always possible to account for the full set of constraints that human developers would need to worry about
- Let's look at Pull Up run on BringToFrontCommand...

©2009–2016 R .J. Walker. All rights reserved.

“If it’s automated, it must work right!”

- Maybe the tool is not implemented correctly
- Maybe the tool fails to consider all the consequences that should concern you
  - Are APIs broken?
- Are the resulting IS-A and HAS-A properties meaningful?
- Understand what you are trying to do
- Inspect the results; run tests!

©2009–2016 R .J. Walker. All rights reserved.

Next time

- Human and programmatic interfaces

©2009–2016 R .J. Walker. All rights reserved.