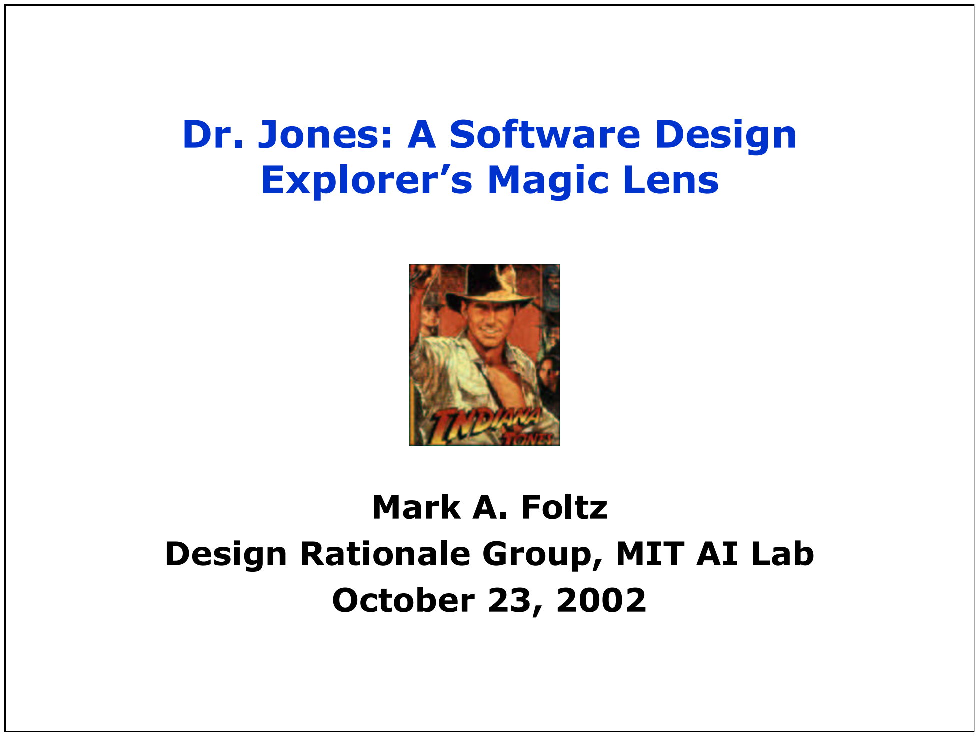
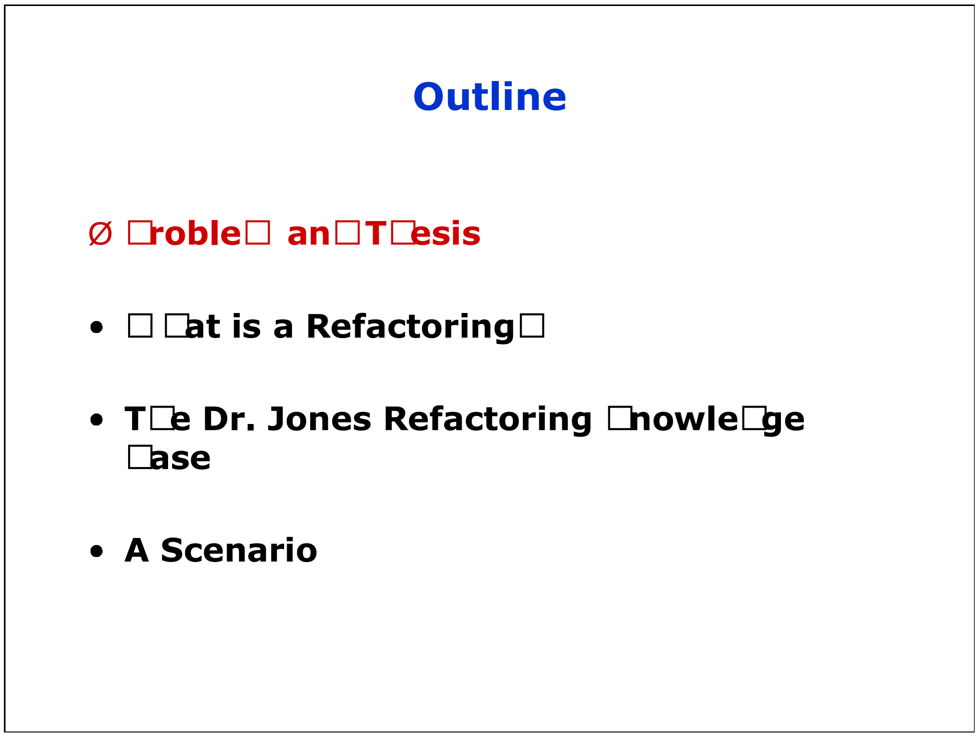


# Dr. Jones: A Software Design Explorer's Magic Lens

**Mark A. Foltz**
**Design Rationale Group, MIT AI Lab**
**October 23, 2002**

Hello, today I am going to talk about my progress on my thesis research on the relationship between software diagramming and software refactoring.



## Outline

Ø □roble□ an□T□esis

- □ □at is a Refactoring□
- T□e Dr. Jones Refactoring □nowle□ge □ase
- A Scenario

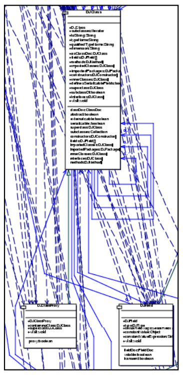
My agenda is to first briefly describe the problem I’m working on and my approach to it.

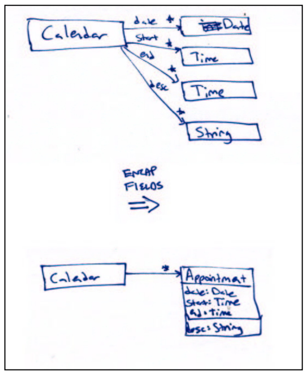
Then I will talk about refactorings in Dr. Jones – what Dr. Jones knows about them and what space of refactorings Dr. Jones will incorporate.

Then I’ll illustrate a scenario of multiple refactorings in Dr. Jones and end with a status update.

First I’ll describe the problem I’m attacking.

## T□e □roble□ Is Re□esign □o□plexit□





- Diagra□s are natural to plan re□esign
- □ut tool□generate□ □iagra□s are too co□plex
- So □ost planning is □one wit□ pen□an□ □paper

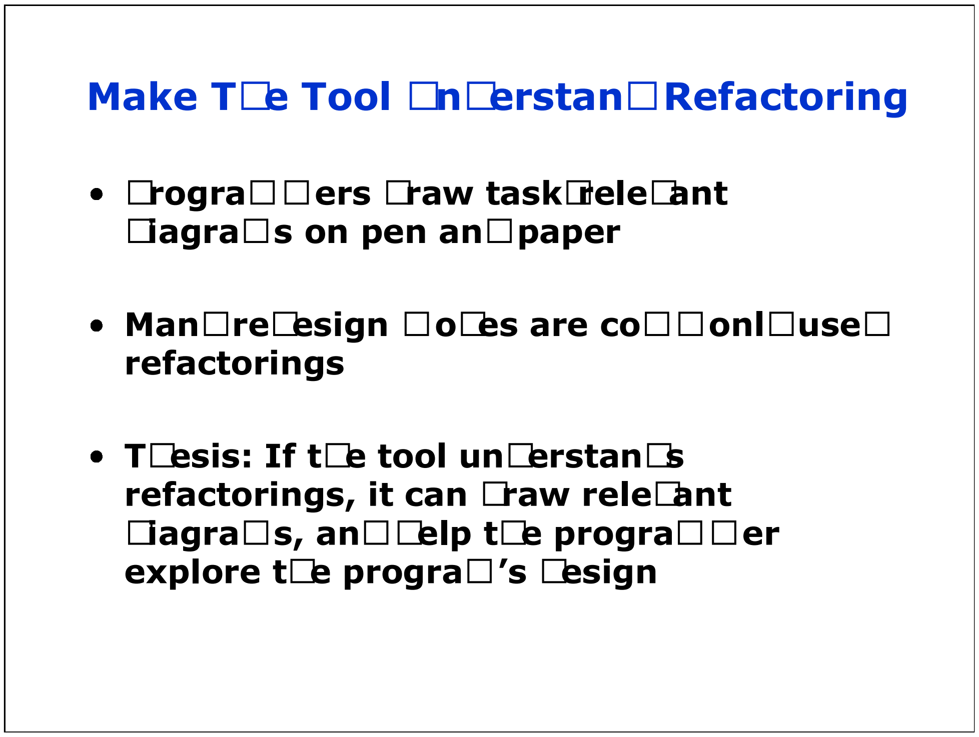
That problem is the complexity of redesigning software.

When thinking about design, programmers prefer diagrams that abstract from the details in source code.

However, current tools provide diagrams like the one on the left that quickly become as complicated as the program they're trying to depict.

The programmer can try to filter out what she doesn't want to see, but usually it's easier for her pick up a pen and paper and redesign by drawing only the parts she wants to see (like the diagram on the right).

Unfortunately, the computer can't help the programmer when she redesigns with pen and paper – and I believe that it should.



## Make T□e Tool □n□erstan□Refactoring

- □rogra□□ers □raw task□rele□ant □iagra□s on pen an□paper
- Man□re□esign □o□es are co□□onl□use□ refactorings
- T□esis: If t□e tool un□erstan□s refactorings, it can □raw rele□ant □iagra□s, an□□elp t□e progra□□er explore t□e progra□'s □esign

I'd like to bring the computer back into this process by starting with two observations.

First, programmers' pen and paper diagrams are task-relevant – they draw the parts of the program they want to change and the dependencies that are involved, and leave out the rest.

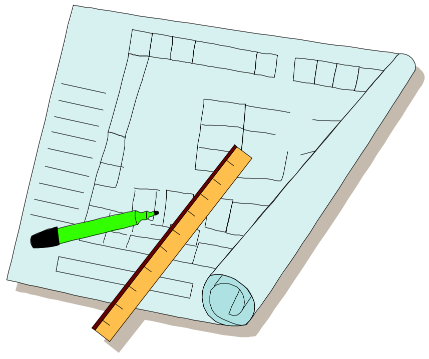
Second, there's a growing body of commonly used design moves called refactorings – local, structural changes to the program that involve a few of its related parts.

These observations led me to the thesis that if a diagramming tool understood the refactorings the programmer wanted to make, it could

(1) draw relevant, task-specific diagrams and

(2) use those diagrams to help the programmer interactively explore the program's design.

## Refactoring Tool Roles



□. □isualize □esign

2. Diagnose proble□s

3. S□ow new □esigns

□. I□ple□ent t□e refactorings

Stepping back for a bit, a redesign tool could potentially support the user in a number of roles, because refactoring is a multi step process.

First, the tool can give the user a visual representation where it's easy to spot opportunities for refactoring.

Second, the tool can diagnose design problems by looking for `bad smells' or `antipatterns', known patterns of design weakness like code duplication.

Third, the tool can show the user the results of proposed refactorings, allowing them to chain together multiple refactorings to visualize new designs.

And finally the tool can verify that the refactorings preserve behavior and implement them by changing the source code.

Choosing the actual refactorings – between steps two and three in this process – is the most difficult step and remains up to the user.

## Dr. Jones' Roles

□. □isualize □esign

2. Diagnose proble□s

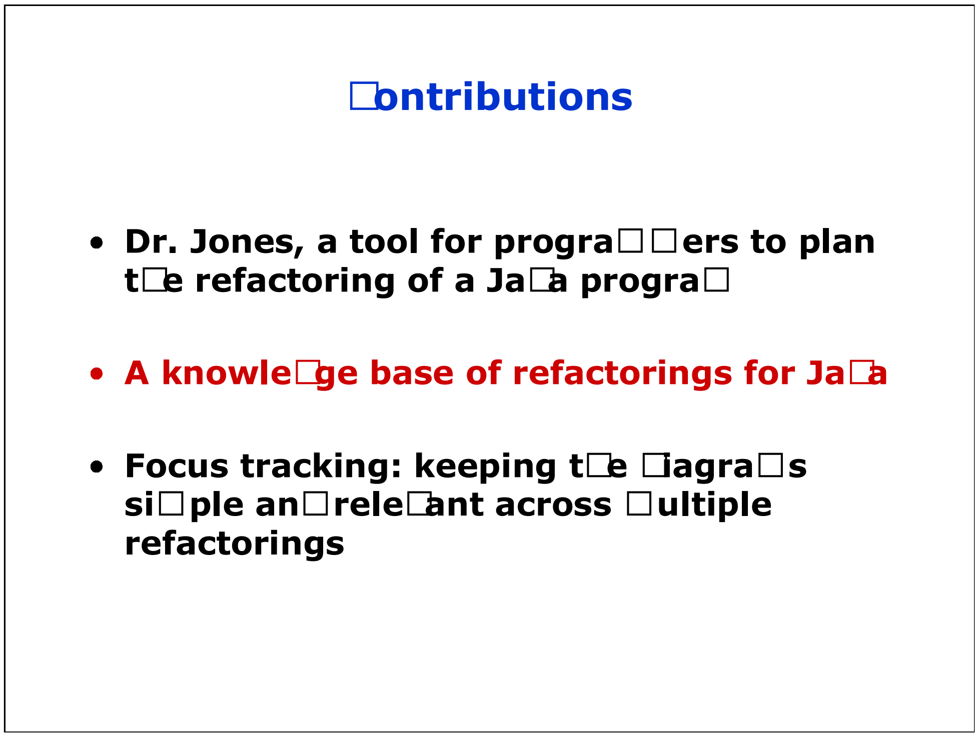
3. S□ow new □esigns

□. I□ple□ent t□e refactorings

Dr. Jones addresses the first and third roles listed here.

The metaphor is that of a fellow programmer who knows the program you're refactoring (although not what it does), can draw accurate diagrams of it, and give the programmer guidance while refactoring.

It innovates by decoupling the steps of planning and implementing the refactorings -- current tools transform the source immediately when the user makes a refactoring decision.



## □ontributions

- Dr. Jones, a tool for progra□□ers to plan t□e refactoring of a Ja□a progra□
- A knowle□ge base of refactorings for Ja□a
- Focus tracking: keeping t□e □iagra□s si□ple an□rele□ant across □ultiple refactorings

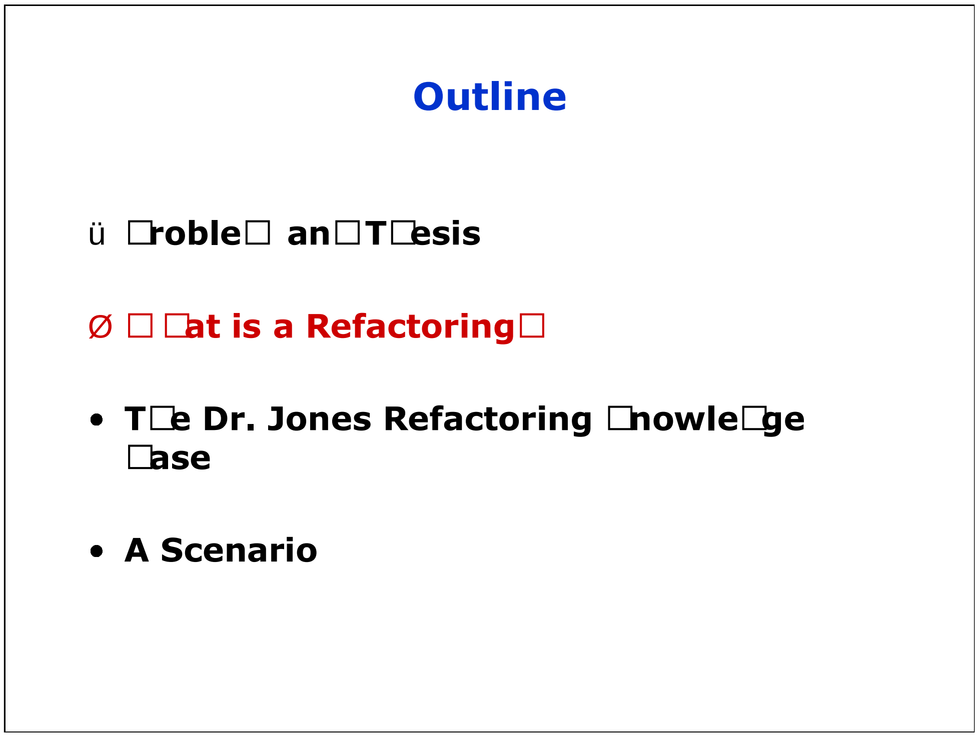
I see three main contributions resulting from this research.

The motivating contribution is Dr. Jones, the tool I am developing that requires two main innovations.

The first innovation is a knowledge base of refactorings for Java programs, built from the perspective of a tool that assists the user in visual design exploration.

The second innovation is a mechanism for keeping the contents of software diagrams relevant across multiple refactorings by tracking the focus of refactoring attention.

The rest of this talk will focus on my progress towards realizing this first innovation.



## Outline

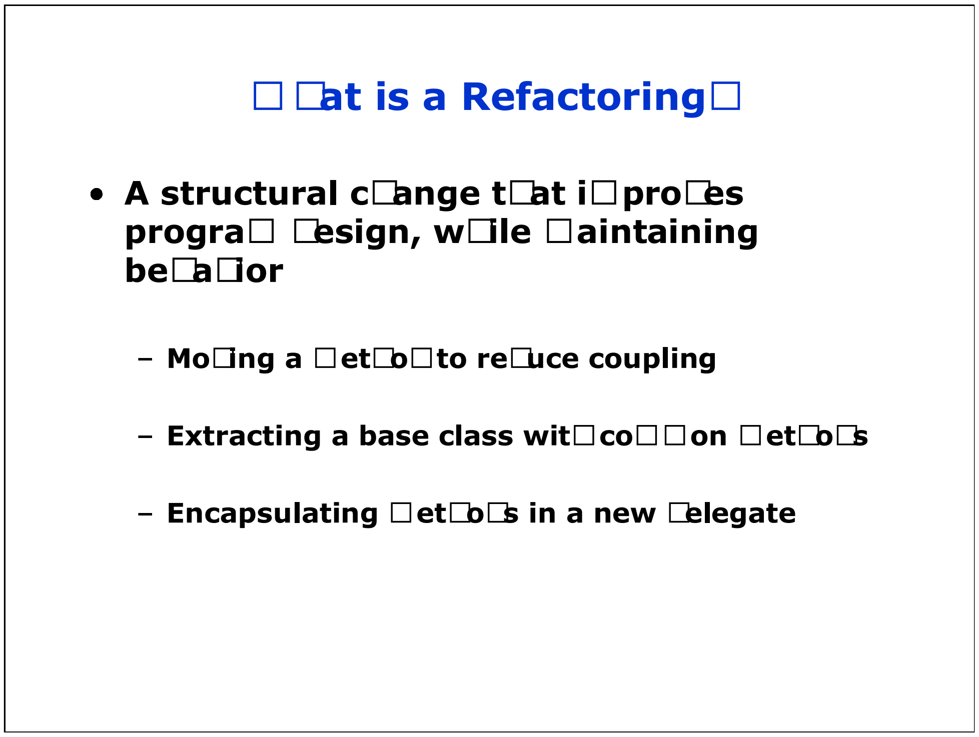
ü □roble□ an□T□esis

Ø □ □at is a Refactoring□

- T□e Dr. Jones Refactoring □nowle□ge □ase

- A Scenario

First I will describe what Dr. Jones knows about each individual refactoring for Java.

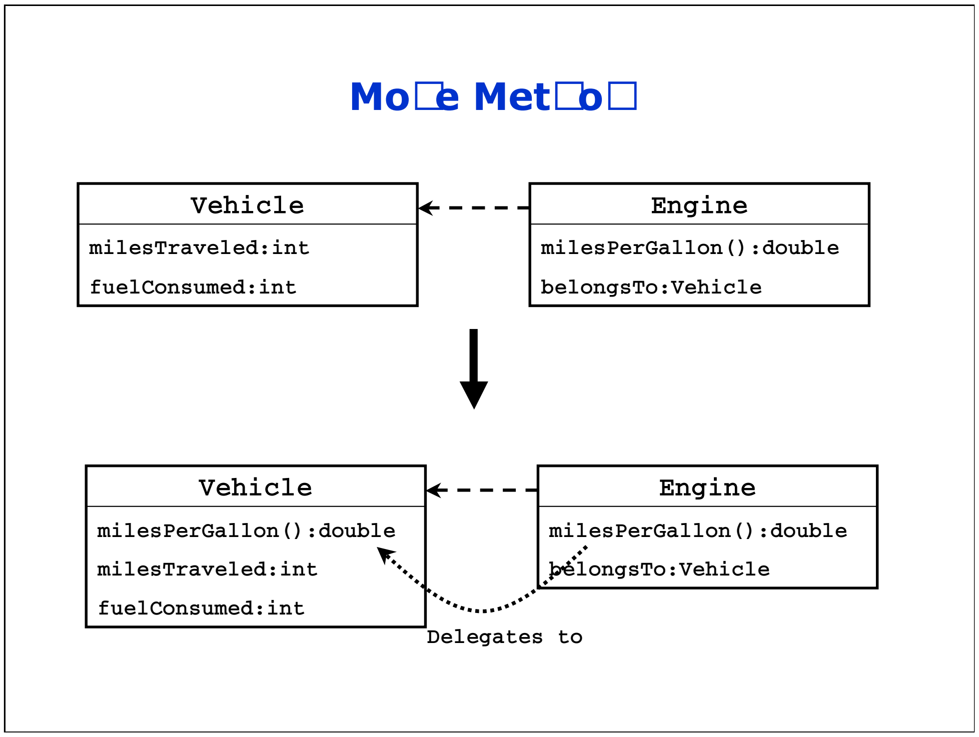


## □ □at is a Refactoring□

- **A structural c□ange t□at i□pro□es progra□ □esign, w□ile □aintaining be□a□ior**
  - **Mo□ing a □et□o□to re□uce coupling**
  - **Extracting a base class wit□co□□on □et□o□s**
  - **Encapsulating □et□o□s in a new □elegate**

Briefly, a refactoring is a structural change to a program that improves its design, without changing its visible behavior.

Common examples are moving a method, generalizing classes to a base class, and encapsulating methods into a delegate.

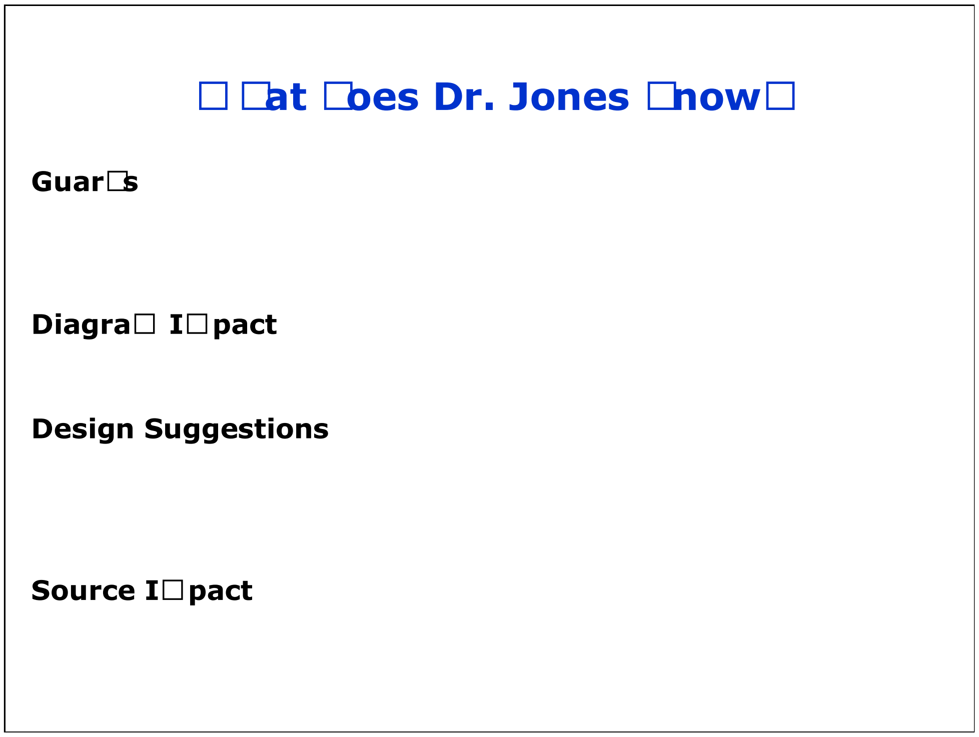


Let’s look at the move method refactoring in detail, to illustrate what Dr. Jones knows about a typical refactoring.

Suppose we decide that the miles per gallon of a vehicle is really a property of a vehicle, and not its engine, to reduce coupling.

We refactor by moving the method to Vehicle, and leaving a skeleton method behind in Engine that delegates to the new location.

What would Dr. Jones need to know to help me plan this refactoring?



## □ □at □oes Dr. Jones □now□

**Guar□s**

**Diagra□ I□pact**

**Design Suggestions**

**Source I□pact**

In Dr. Jones, I represent a refactoring by four pieces of knowledge.

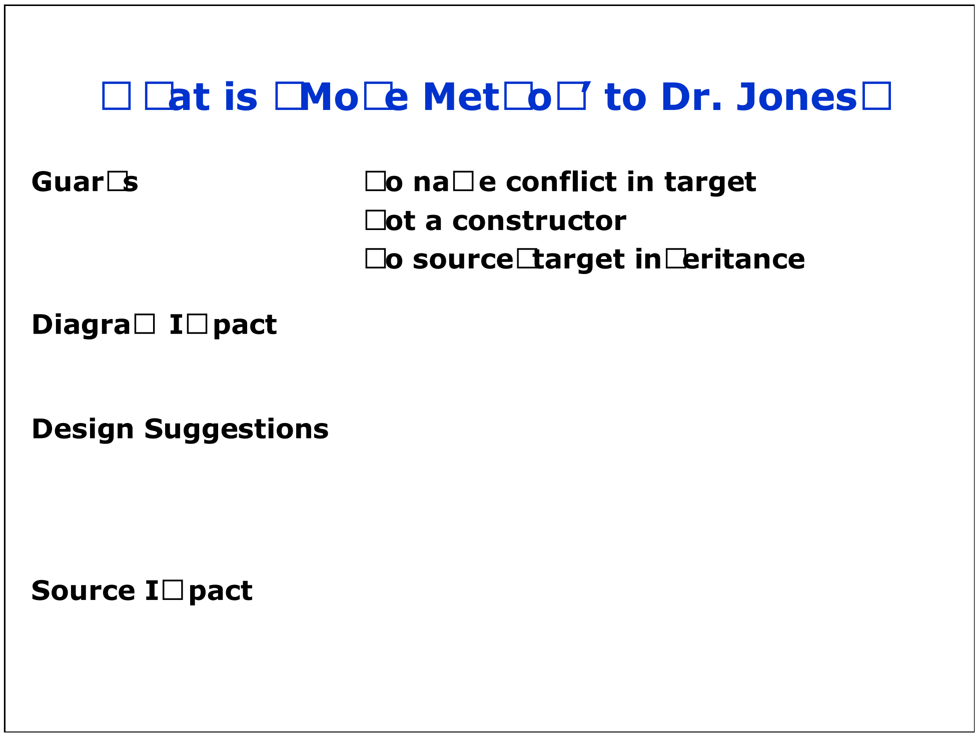
First, what are the obvious reasons not to perform the refactoring (the guards).

Second, how does the refactoring change Dr. Jones' representation of the program design and thus what is shown in its diagrams.

Third, does the refactoring suggest other refactorings that are likely to improve the program design.

And finally, where are the places in the source that might have to be changed to implement the refactoring.

I'll now examine these four pieces of knowledge in detail for the move method refactoring.



## □ □at is □Mo□e Met□o□′ to Dr. Jones□

**Guar□s**
- **□o na□e conflict in target**
- **□ot a constructor**
- **□o source□target in□eritance**

**Diagra□ I□pact**

**Design Suggestions**

**Source I□pact**

First, Dr. Jones can check for guards – obvious reasons that one wouldn't want to do the refactoring.

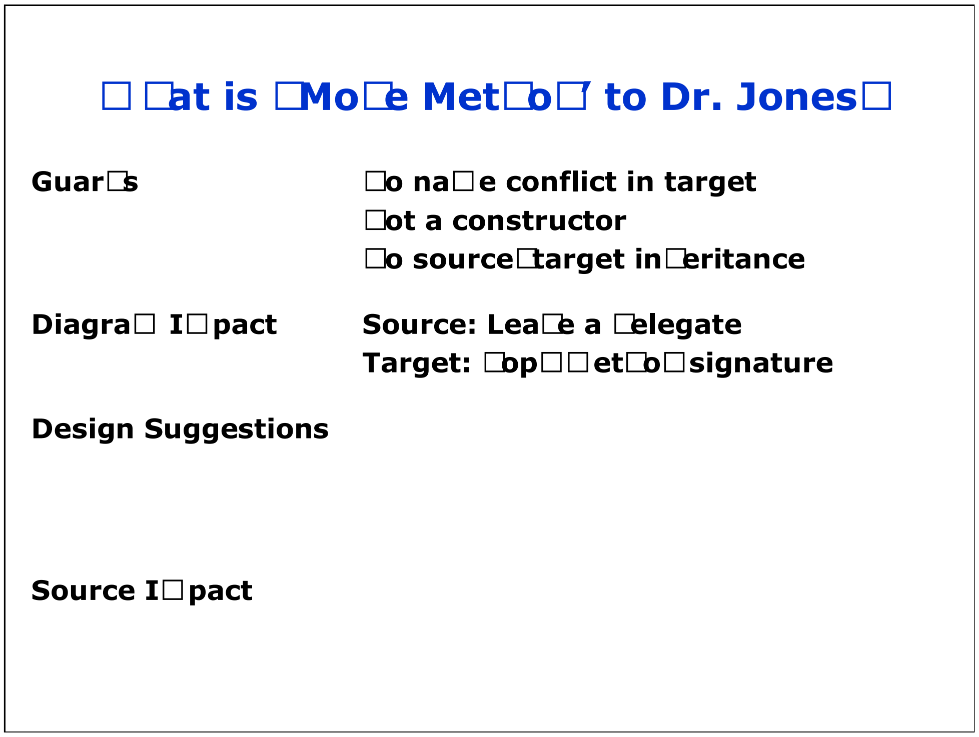
In this case, Dr. Jones can check for name conflicts, and that you're not trying to move a constructor.

Also a different set of rules apply for move method if the source and target classes are related by inheritance.

Note that these`guards' don't completely check that the refactoring preserves the program's behavior (since that would involve much more difficult analyses).

Rather these are more like sanity checks to help the programmer avoid refactoring mistakes.

(Dr. Jones can remind the programmer to check the more difficuly safety conditions when it's time to implement the refactorings. This runs the risk, however, of allowing the programmer to plan unsafe refactorings with Dr. Jones.)

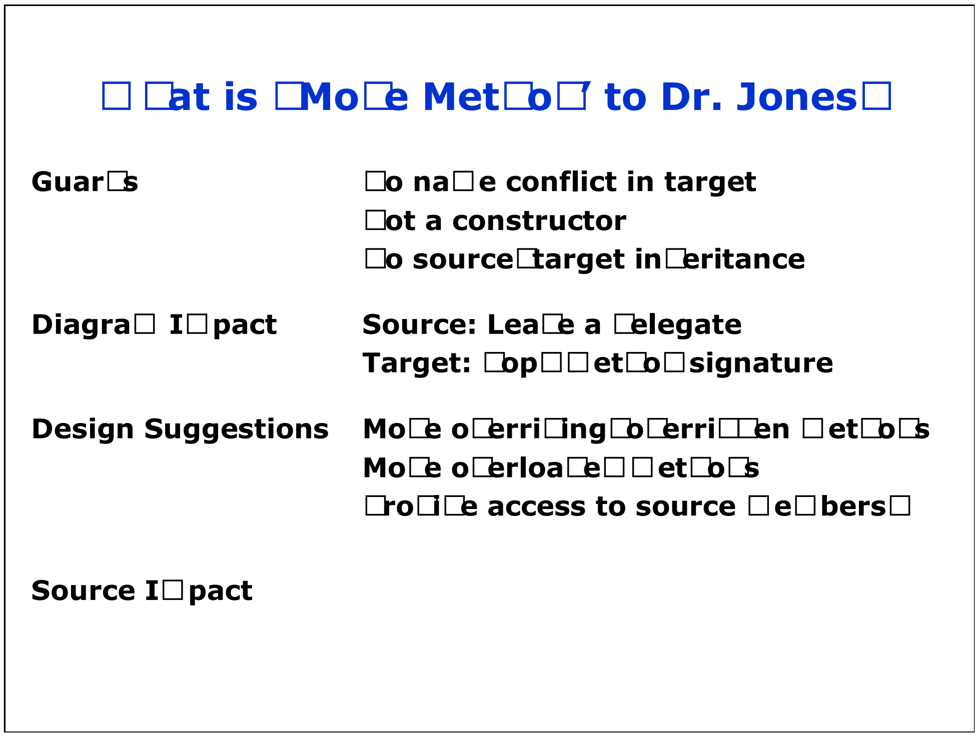


## □ □at is □Mo□e Met□o□′ to Dr. Jones□

| | |
|---|---|
| **Guar□s** | **□o na□e conflict in target**<br>**□ot a constructor**<br>**□o source□target in□eritance** |
| **Diagra□ I□pact** | **Source: Lea□e a □elegate**<br>**Target: □op□□et□o□signature** |
| **Design Suggestions** | |
| **Source I□pact** | |

The impact on the design representation is straightforward (as we saw a few slides ago in the move method example).

Dr. Jones copies the method signature from the source to the target and notes that the old method delegates to the new location.



## □ □at is □Mo□e Met□o□′ to Dr. Jones□

| | |
|---|---|
| **Guar□s** | **□o na□e conflict in target**<br>**□ot a constructor**<br>**□o source□target in□eritance** |
| **Diagra□ I□pact** | **Source: Lea□e a □elegate**<br>**Target: □op□□et□o□signature** |
| **Design Suggestions** | **Mo□e o□erri□ing□o□erri□□en □et□o□s**<br>**Mo□e o□erloa□e□□et□o□s**<br>**□ro□i□e access to source □e□bers□** |
| **Source I□pact** | |

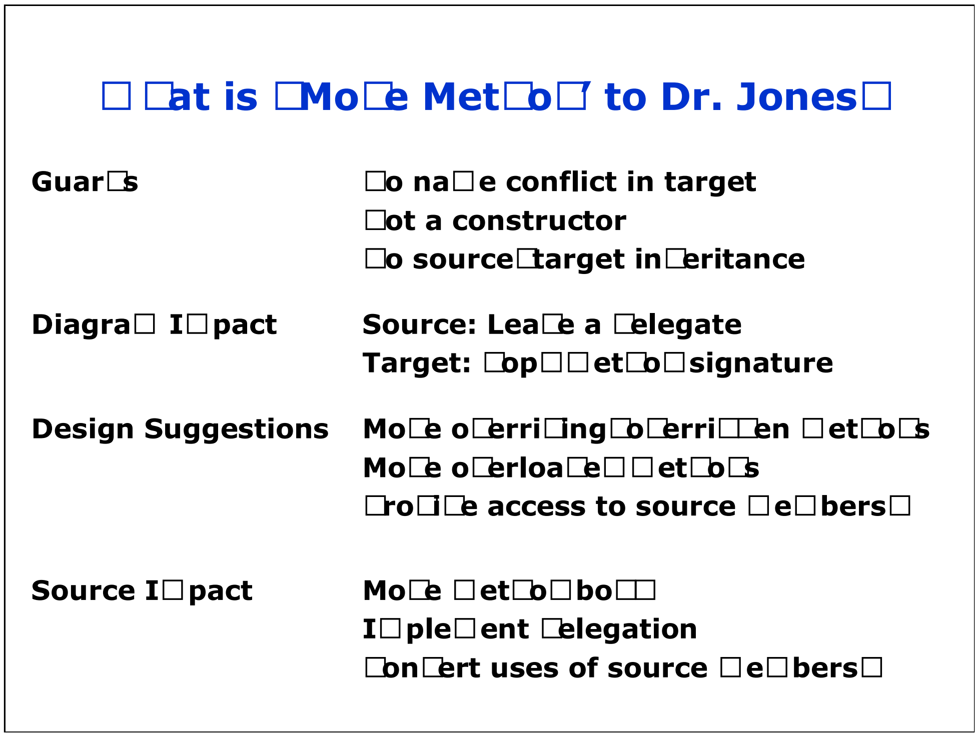
Dr. Jones can make several design suggestions even for this seemingly straightforward refactoring.

If the method is polymorphic in the source hierarchy, it's likely that the programmer will want to express that polymorphism on the target hierarchy.

So we leave to-dos for the programmer to move the overriding and overridden methods to appropriate places in the target hierarchy.

Also, if the method is overloaded with functions of the same name but different signatures, then the programmer might want to move those as well.

Finally, if the method uses fields or methods in the source class, the programmer will need to provide access to them (I.e., by encapsulating those fields).



## □ □at is □Mo□e Met□o□′ to Dr. Jones□

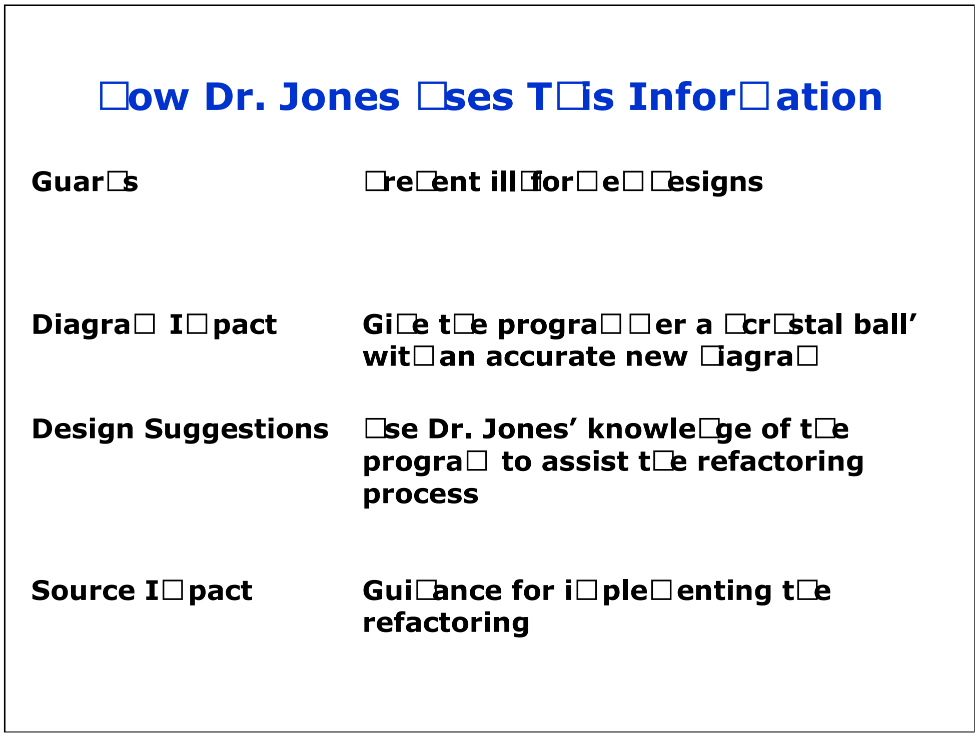
| | |
|---|---|
| **Guar□s** | **□o na□e conflict in target**<br>**□ot a constructor**<br>**□o source□target in□eritance** |
| **Diagra□ I□pact** | **Source: Lea□e a □elegate**<br>**Target: □op□□et□o□signature** |
| **Design Suggestions** | **Mo□e o□erri□ing□o□erri□□en □et□o□s**<br>**Mo□e o□erloa□e□□et□o□s**<br>**□ro□i□e access to source □e□bers□** |
| **Source I□pact** | **Mo□e □et□o□bo□□**<br>**I□ple□ent □elegation**<br>**□on□ert uses of source □e□bers□** |

Finally we can give some guidance to the programmer when he is ready to tackle the source, like moving the method body to the target, implementing the delegation in the source, and converting the uses of source members.

(Dr. Jones builds a cross reference of which methods use which other fields and methods for use in these last two steps.)

Dr. Jones decouples the choices of the refactoring steps to take, and the actual manipulation on the source to implement the refactorings.

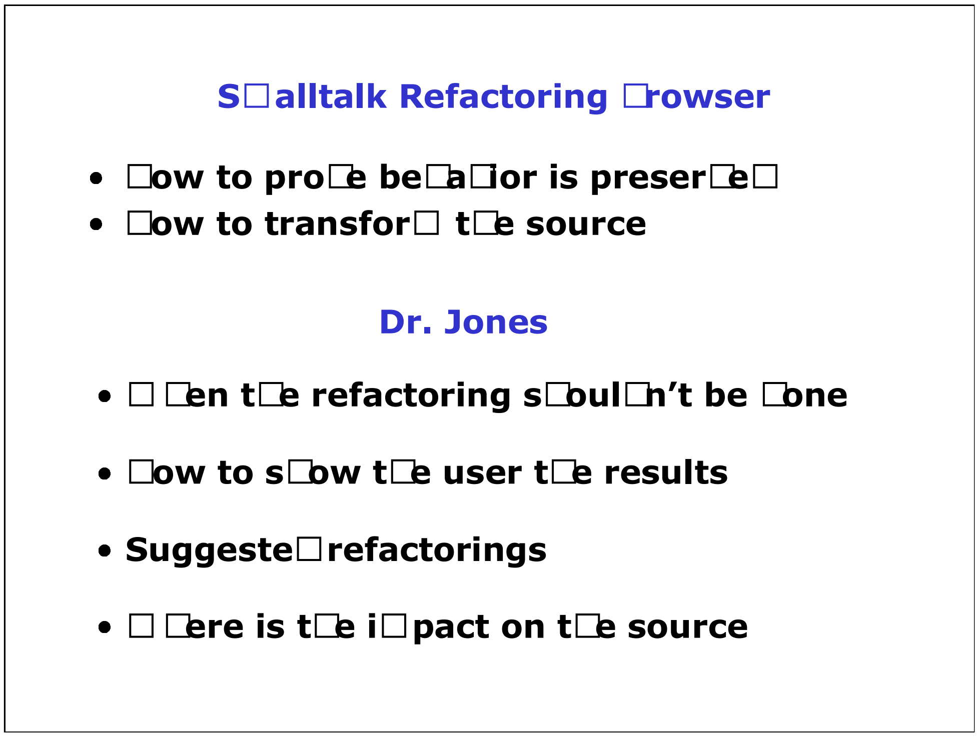
In this way multiple alternatives can be more easily explored, and the hard work of implementing the refactorings undertaken once an alternative is chosen.



## □ow Dr. Jones □ses T□is Infor□ation

| | |
|---|---|
| Guar□s | □re□ent ill□for□e□ □esigns |
| Diagra□ I□pact | Gi□e t□e progra□□er a □cr□stal ball' wit□an accurate new □iagra□ |
| Design Suggestions | □se Dr. Jones' knowle□ge of t□e progra□ to assist t□e refactoring process |
| Source I□pact | Gui□ance for i□ple□enting t□e refactoring |

Dr. Jones uses these four pieces of knowledge together to play its role as a diagramming assistant.



## S□alltalk Refactoring □rowser

- □ow to pro□e be□a□ior is preser□e□
- □ow to transfor□ t□e source

## Dr. Jones

- □ □en t□e refactoring s□oul□n't be □one
- □ow to s□ow t□e user t□e results
- Suggeste□refactorings
- □ □ere is t□e i□pact on t□e source

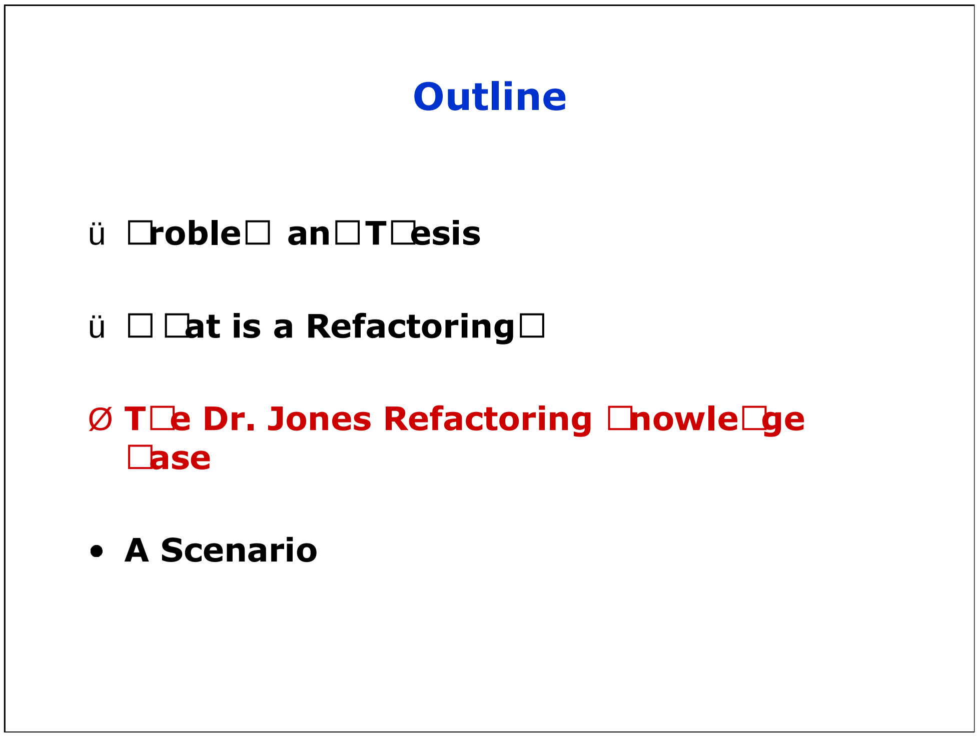
I'd like to compare Dr. Jones' knowledge about refactoring to that of another research project, the Smalltalk Refactoring Browser developed at UIUC.

The Refactoring Browser was primarily concerned with giving the user a safe and reliable tool – the user could trust it to know when a refactoring is behavior-preserving, and if so to transform the source correctly.

Dr Jones on the other hand has knowledge that will give the user visual feedback on the new designs generated by refactoring, prevent bad refactorings, and suggest`follow-up' refactorings.

These kinds of knowledge haven't been explicitly considered before in a refactoring tool, and I believe my specifications of it represents a contribution to refactoring research.

I also believe these two bodies of knowledge are complementary, and a tool that integrates, for example, design diagnosis, design exploration and source transformation would be a more complete solution and a fruitful direction for future work.



## Outline

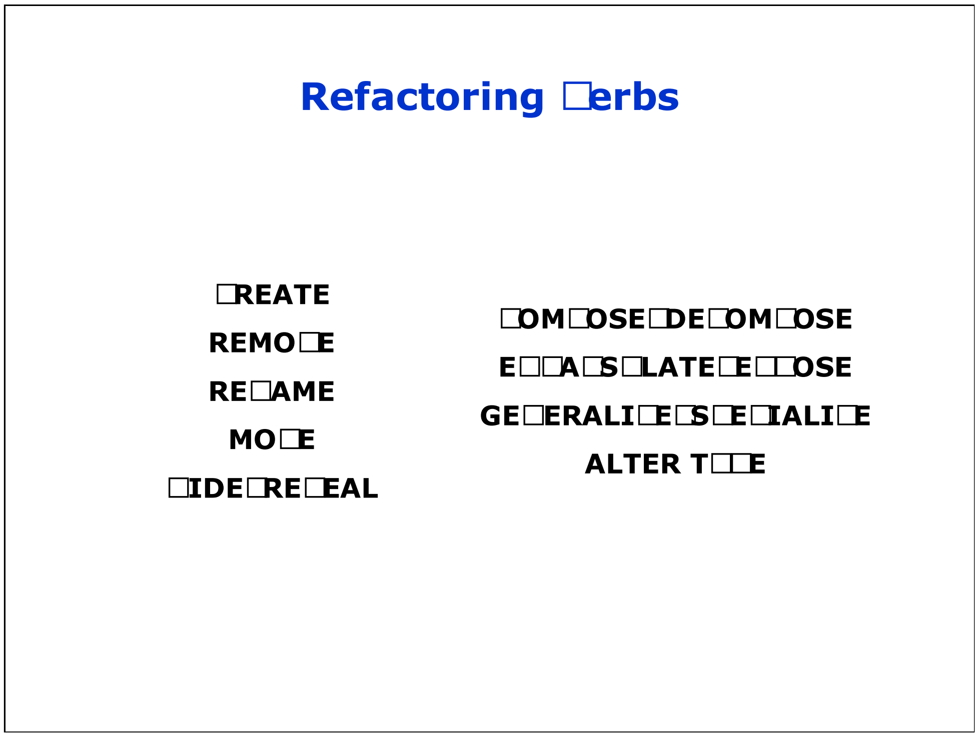
ü □roble□ an□T□esis

ü □ □at is a Refactoring□

Ø T□e Dr. Jones Refactoring □nowle□ge □ase

• A Scenario

Next I am going to give an overview of what refactorings are included in the knowledge base.



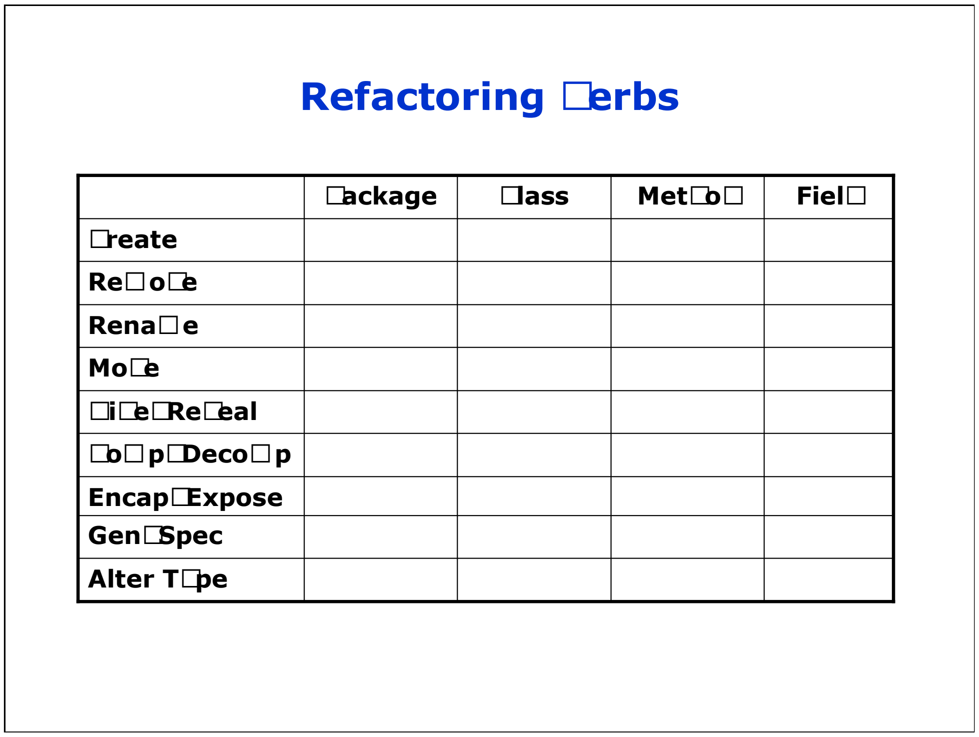
## Refactoring □erbs

□REATE
REMO□E
RE□AME
MO□E
□IDE□RE□EAL

□OM□OSE□DE□OM□OSE
E□□A□S□LATE□E□□OSE
GE□ERALI□E□S□E□IALI□E
ALTER T□□E

The KB is structured around a set of refactoring verbs that can be applied to the major program elements in Java.

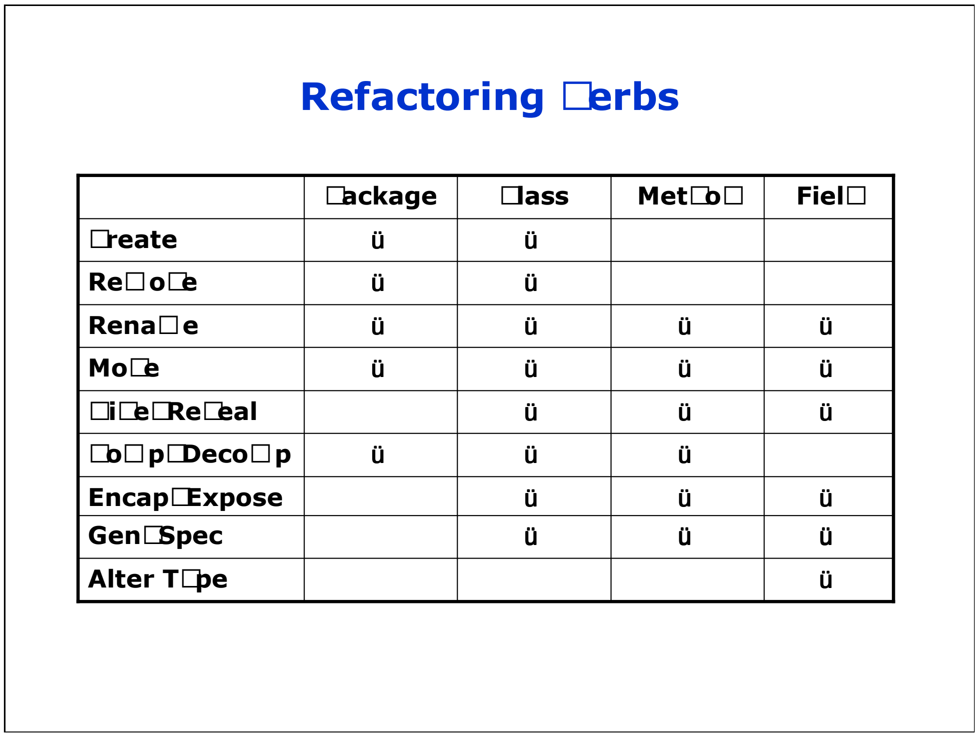
This vocabulary was motivated by the desire to have a economical number of actions that the user can apply to elements of the diagram, instead of a flat list that would have to be learned and remembered.



## Refactoring □erbs

| | □ackage | □lass | Met□o□ | Fiel□ |
|---|---|---|---|---|
| □reate | | | | |
| Re□o□e | | | | |
| Rena□e | | | | |
| Mo□e | | | | |
| □i□e□Re□eal | | | | |
| □o□p□Deco□p | | | | |
| Encap□Expose | | | | |
| Gen□Spec | | | | |
| Alter T□pe | | | | |

The vocabulary also sets up a space of possible refactorings whose cases can be filled in for a specific language (in this case Java).

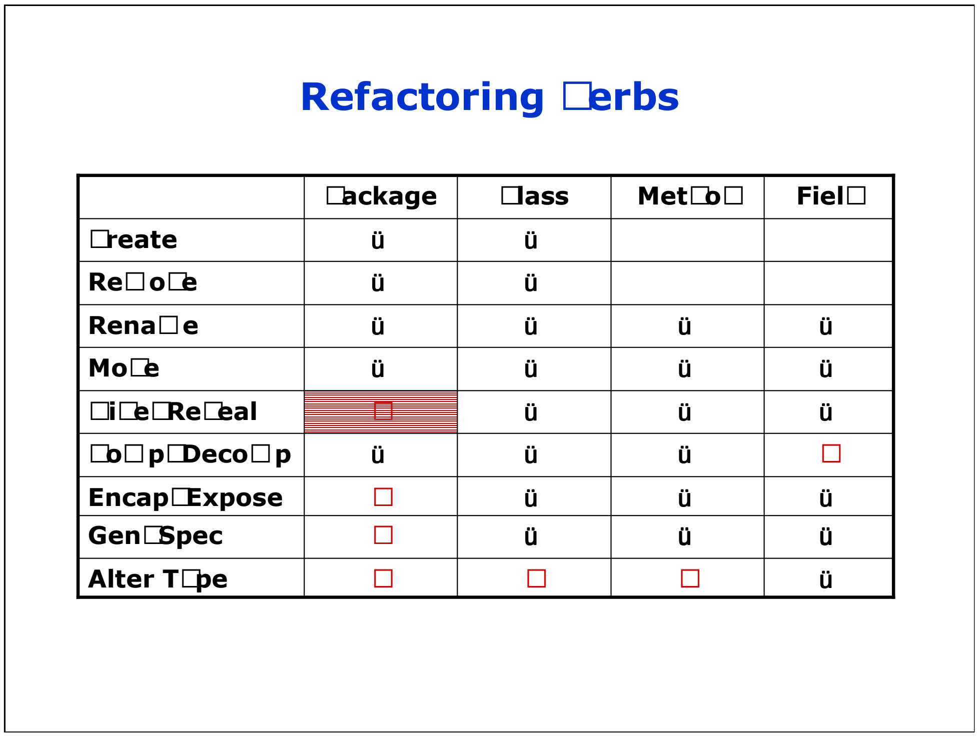


## Refactoring □erbs

| | □ackage | □lass | Met□o□ | Fiel□ |
|---|---|---|---|---|
| **□reate** | ü | ü | | |
| **Re□o□e** | ü | ü | | |
| **Rena□e** | ü | ü | ü | ü |
| **Mo□e** | ü | ü | ü | ü |
| **□i□e□Re□eal** | | ü | ü | ü |
| **□o□p□Deco□p** | ü | ü | ü | |
| **Encap□Expose** | | ü | ü | ü |
| **Gen□Spec** | | ü | ü | ü |
| **Alter T□pe** | | | | ü |

I have specified entries in the KB for each of these check marks in a semi-formal language.

Most of the missing marks are cases that don’t make sense in Java, [next slide]



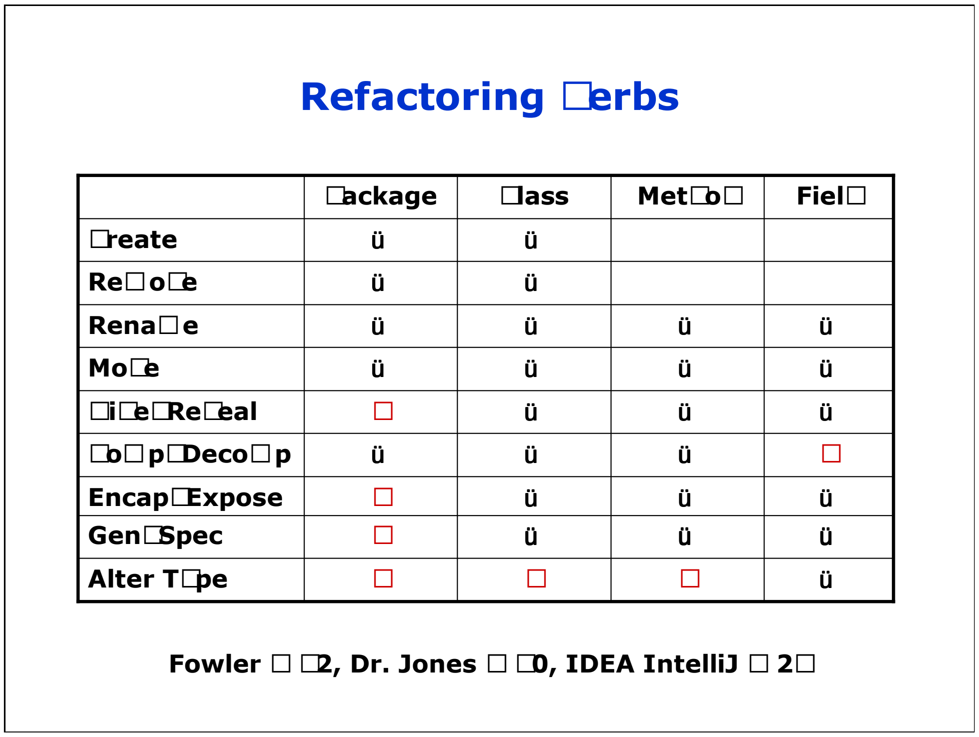
## Refactoring □erbs

| | □ackage | □lass | Met□o□ | Fiel□ |
|---|---|---|---|---|
| □reate | ü | ü | | |
| Re□o□e | ü | ü | | |
| Rena□e | ü | ü | ü | ü |
| Mo□e | ü | ü | ü | ü |
| □i□e□Re□eal | □ | ü | ü | ü |
| □o□p□Deco□p | ü | ü | ü | □ |
| Encap□Expose | □ | ü | ü | ü |
| Gen□Spec | □ | ü | ü | ü |
| Alter T□pe | □ | □ | □ | ü |

I.e. hiding a package.

Creating and removing fields and methods aren’t included because they don’t seem to be in the spirit of behavior preservation, and adding new functionality is a separate concern.

The intention is to let programmers naturally express typical sequences of refactorings they would use in practice.



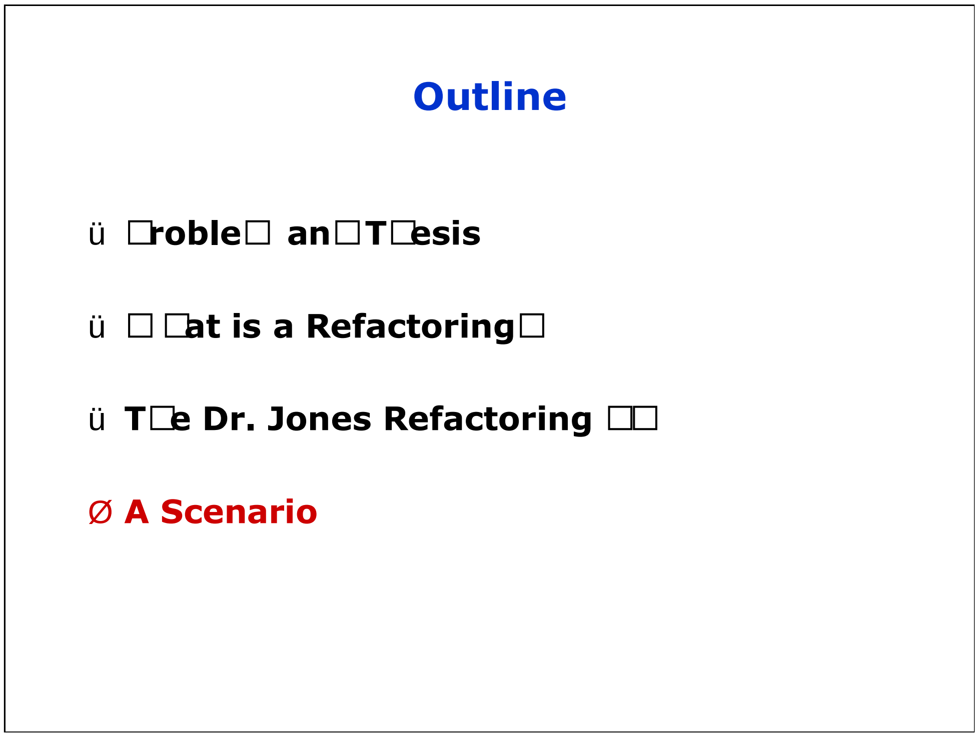
## Refactoring □erbs

| | □ackage | □lass | Met□o□ | Fiel□ |
|---|---|---|---|---|
| □reate | ü | ü | | |
| Re□o□e | ü | ü | | |
| Rena□e | ü | ü | ü | ü |
| Mo□e | ü | ü | ü | ü |
| □i□e□Re□eal | □ | ü | ü | ü |
| □o□p□Deco□p | ü | ü | ü | □ |
| Encap□Expose | □ | ü | ü | ü |
| Gen□Spec | □ | ü | ü | ü |
| Alter T□pe | □ | □ | □ | ü |

**Fowler □ □2, Dr. Jones □ □0, IDEA IntelliJ □ 2□**

We can compare the coverage of Dr. Jones to the catalog in Fowler’s 1999 book and a leading refactoring CASE tool.

Although we’re comparing apples and oranges, in terms of expressiveness, Dr. Jones has a significant fraction of the Fowler’s refactorings collected from practice and more than a source-transformation-only CASE tool.



## Outline

ü □roble□ an□T□esis

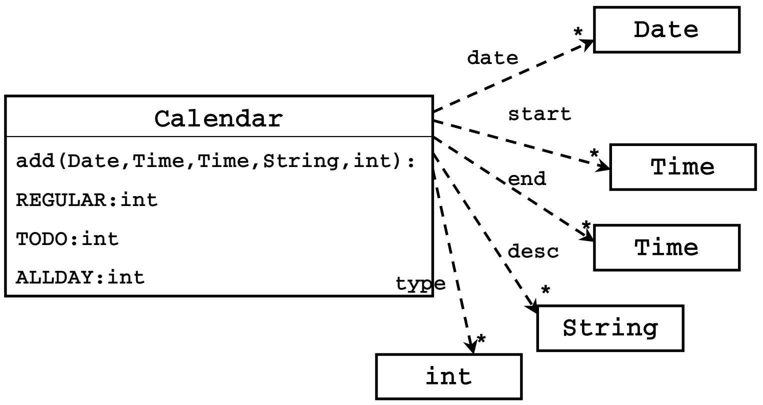
ü □ □at is a Refactoring□

ü T□e Dr. Jones Refactoring □□

Ø A Scenario

To bring the two main parts of the talk together I'll present a scenario that shows Dr. Jones' body of knowledge in action. simulte

## Scenario



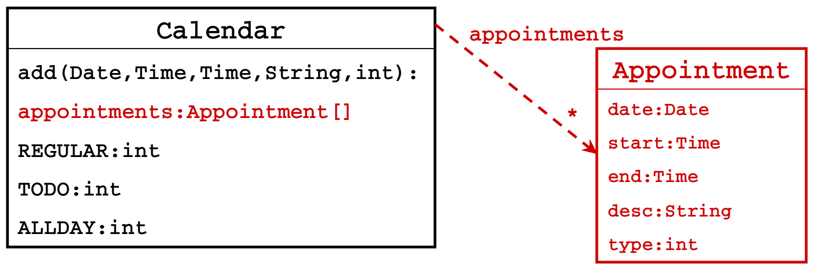
We start with a class that keeps a calendar of appointments.

The information for each appointment is kept in fields of arrays (one for the date, one for the start time, etc.)

Appointments can be made in three types: regular, to-dos, and and all-day (indicated by a numeric type code).

The programmer would like to refactor this to create an extensible abstraction for an Appointment.

## Scenario



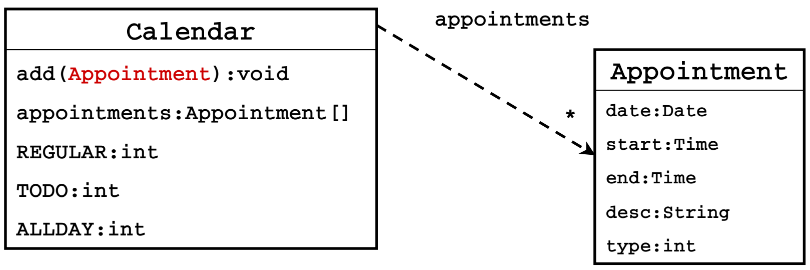
**Encapsulate Fiel□s**

The first step is to encapsulate the array fields into a new Appointment class.

Dr. Jones would ask the user to name the new class, and to choose a container for the aggregation (in this case an array).

It would then change the program representation as necessary and diagram the new design, including replacing the multiple aggregation edges with a single new one.

## Scenario



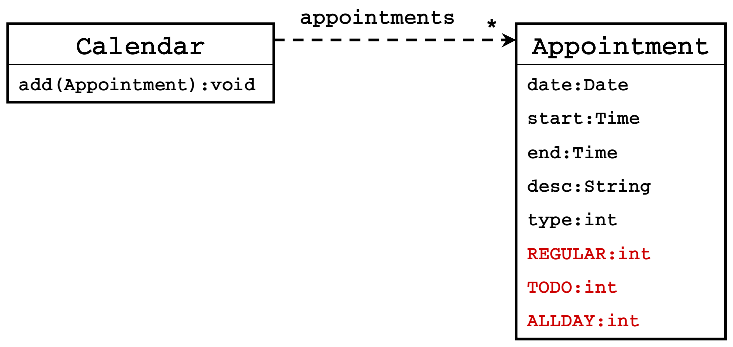
**Encapsulate □ara□eters**

Now that we have an Appointment class, it makes sense to replace the multiple parameters to add() with a single Appointment parameter.

The user does this with the encapsulate parameters refactoring.

Since Dr. Jones knows where the add() method is called in the original program, it can tell the programmer where to change the calling syntax later.

## Scenario



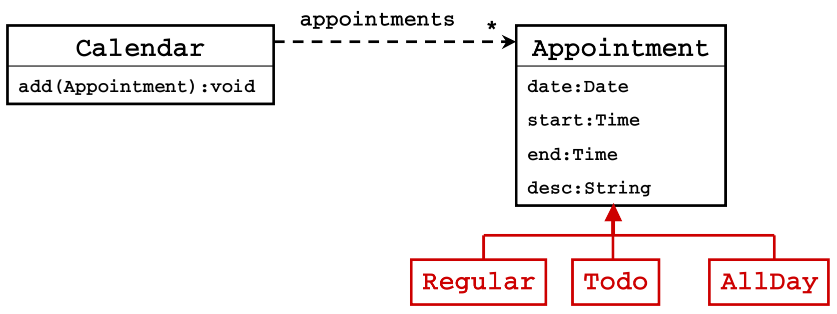
**Mo□e Fiel□s**

Now the user would like to make the appointment-type-specific behavior explicit in the class hierarchy.

The user prepares for this by moving the type code fields to the Appointment class.

[ include something about moving methods in Calendar here, or come up with an example. ]

## Scenario



**Encapsulate T□pe □o□e to Subclasses**

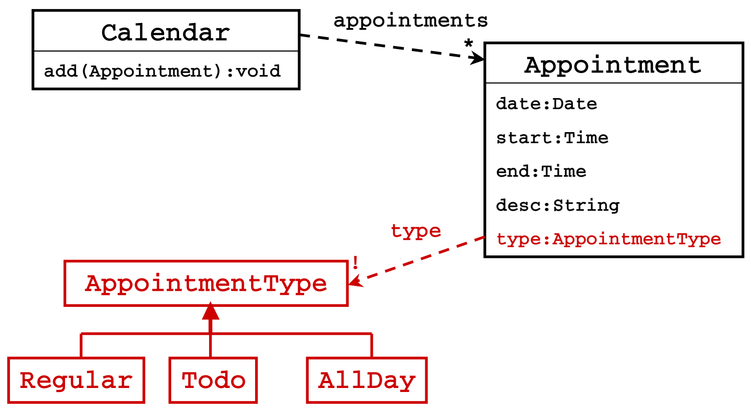
The user can then encapsulate the type codes of Appointment into subclasses.

Things look pretty good, until she realizes that the user of the calendar might want to change the type of an appointment.

Objects can’t change class in Java, so this creates a problem.

Here she can use Dr. Jones’ ability to explore alternatives to back up and try a different refactoring.

## Scenario



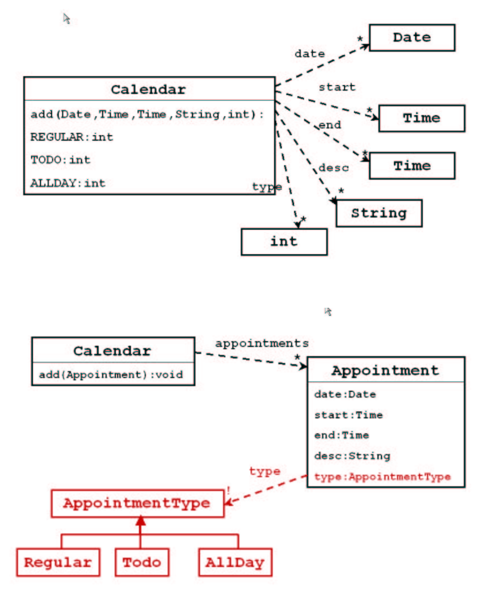
**Oops □Encapsulate T□pe □o□e to State**

Encapsulating the type codes in a separate class avoids this probem.

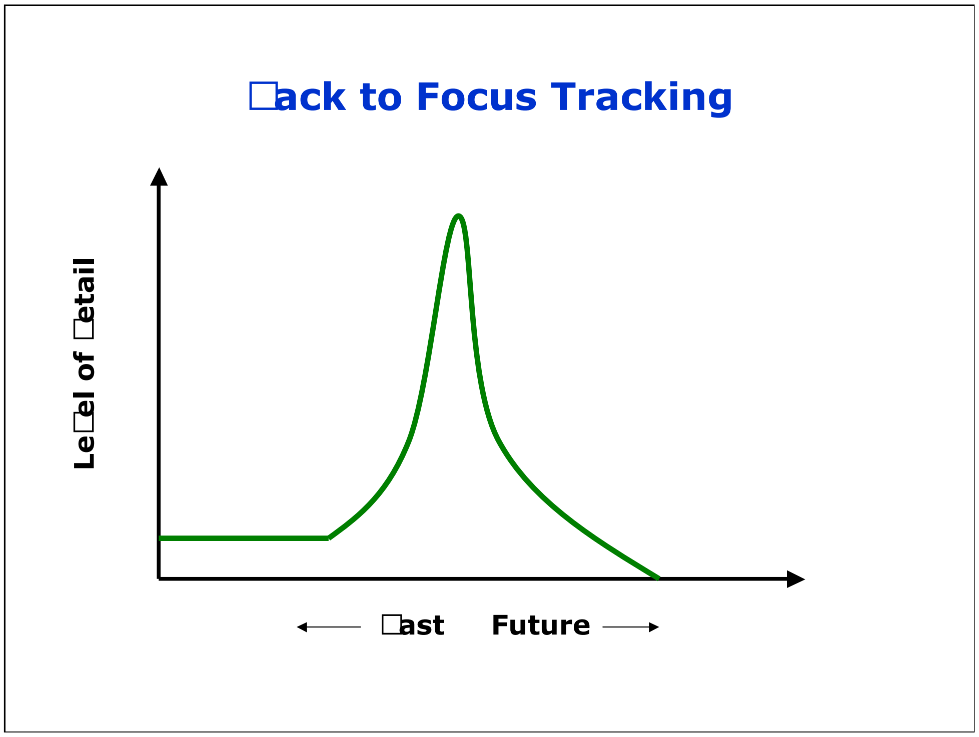
An Appointment can change its type dynamically by reassigning its AppointmentType instance.

We can compare this design to the original to see the improvement [flip slides].

## Scenario



[ slide copied here to illustrate improvement in design ]



## Back to Focus Tracking

Before concluding, I wanted to bring the talk back to the issues that got me started looking at refactoring, that of keeping the diagrams simple and relevant while redesigning software.

This is the goal of focus tracking, and I see it as one of Dr. Jones' major payoffs to the user.

The KB I've described will be a major component of the focus tracking mechanism, since it knows what program elements are involved in the current refactoring and which ones are likely to be further refactored.

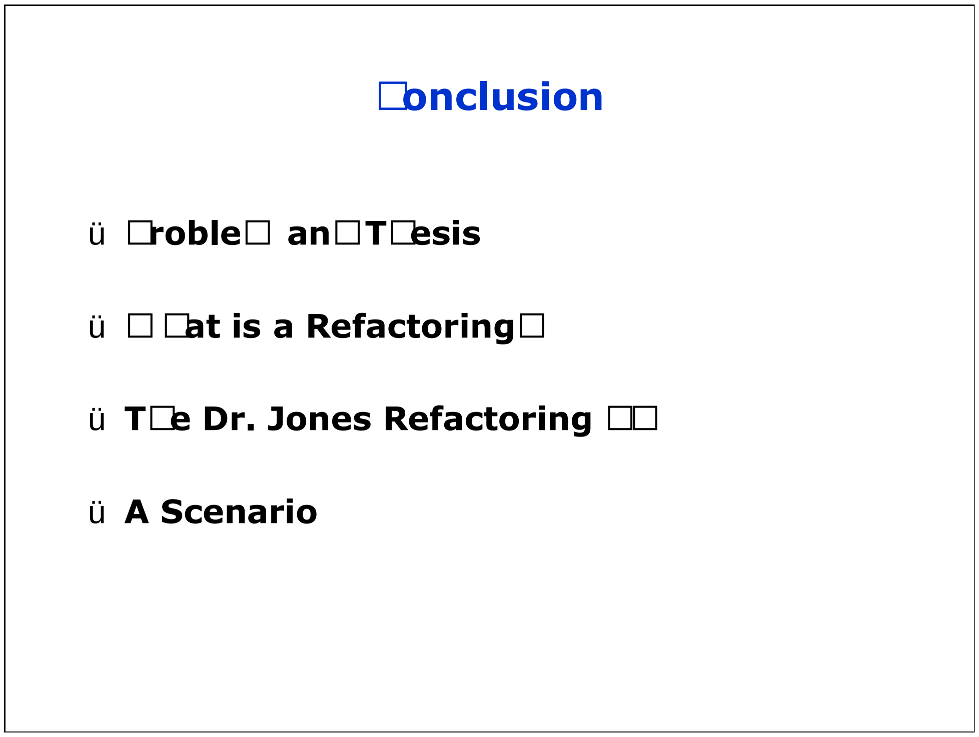
The focus tracking mechanism will use this information to render the elements at appropriate levels of detail, shown here.

For instance, it could show some historical context by showing elements refactored in the past at a low level of detail.

The currently refactored elements will get the highest level of detail.

Likely future refactorings will also get more detail, but since we can't predict the user's next actions exactly, the drop off is quick.

The specifics of this mechanism remain future work in my research.



## Conclusion

ü **Problem and Thesis**

ü **What is a Refactoring?**

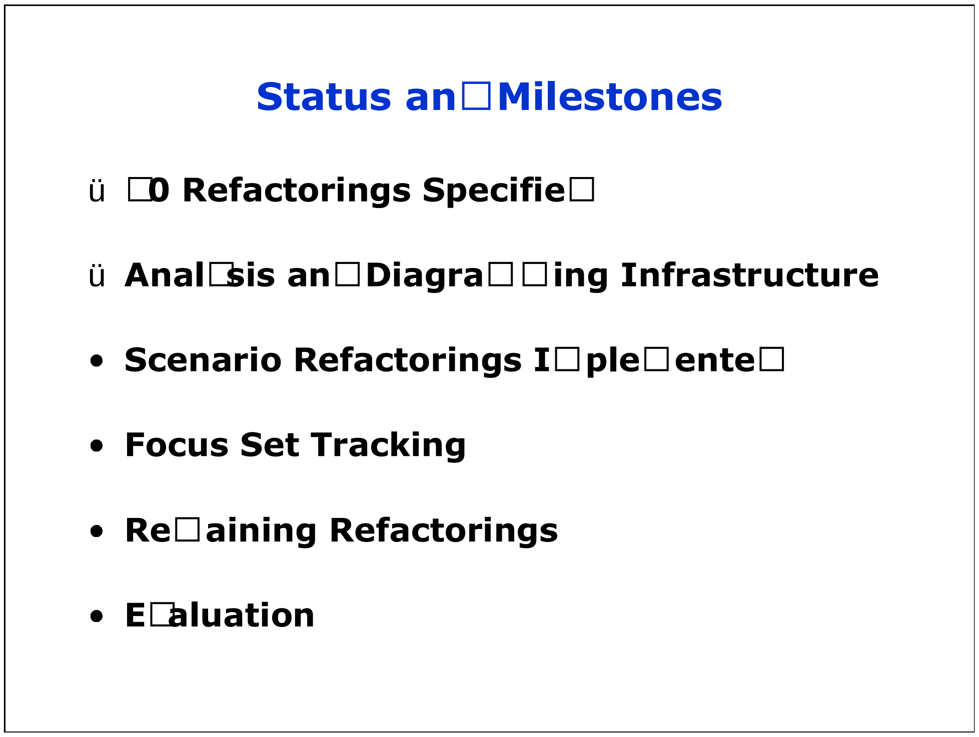
ü **The Dr. Jones Refactoring [illegible]**

ü **A Scenario**

Today I've presented an overview of my research progress on Dr. Jones, an interactive refactoring tool for Java programs.

I've described the four kinds of design exploration knowledge I have specified for each of Dr. Jones' 50 refactorings.

And I've also described a scenario that will drive the next phases of my research.



## Status an□Milestones

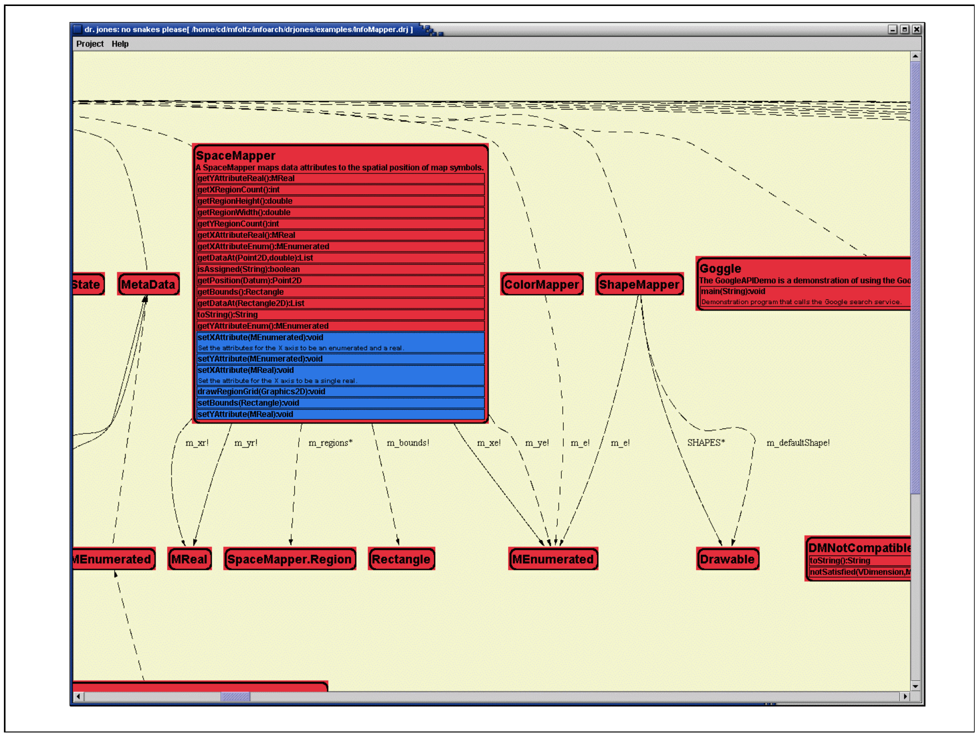
- ü □0 Refactorings Specifie□
- ü Anal□sis an□Diagra□□ing Infrastructure
- Scenario Refactorings I□ple□ente□
- Focus Set Tracking
- Re□aining Refactorings
- E□aluation

The tasks I've completed are the specification of 50 refactorings for Dr. Jones in a semi-formal language, and the infrastructure to analyze and flexibly diagram existing Java programs.

Next I plan to implement the set of refactorings used in the scenario and understand what focus tracking would be for the scenario.

This in turn will drive work on a more general focus tracking mechanism, and the implementation of the remaining refactorings I have specified.

In the final phase of my research I want to evaluate Dr. Jones by obtaining user reactions and feedback.


dr. jones: no snakes please[ /home/cd/mfoltz/infoarch/drjones/examples/InfoMapper.drj ]
Project
Help
SpaceMapper
A SpaceMapper maps data attributes to the spatial position of map symbols.
getYAttributeReal():MReal
getXRegionCount():int
getRegionHeight():double
getRegionWidth():double
getYRegionCount():int
getXAttributeReal():MReal
getXAttributeEnum():MEnumerated
getDataAt(Point2D,double):List
isAssigned(String):boolean
getPosition(Datum):Point2D
getBounds():Rectangle
getDataAt(Rectangle2D):List
toString():String
getYAttributeEnum():MEnumerated
setXAttribute(MEnumerated):void
Set the attributes for the X axis to be an enumerated and a real.
setYAttribute(MEnumerated):void
setXAttribute(MReal):void
Set the attribute for the X axis to be a single real.
drawRegionGrid(Graphics2D):void
setBounds(Rectangle):void
setYAttribute(MReal):void
State
MetaData
ColorMapper
ShapeMapper
Goggle
The GoogleAPIDemo is a demonstration of using the Goo
main(String):void
Demonstration program that calls the Google search service.
m_xr!
m_yr!
m_regions*
m_bounds!
m_xe!
m_ye!
m_e!
m_e!
SHAPES*
m_defaultShape!
MEnumerated
MReal
SpaceMapper.Region
Rectangle
MEnumerated
Drawable
DMNotCompatible
toString():String
notSatisfied(VDimension,M

Before breaking for discussion I'll give you a tour of the prototype's current capabilities.