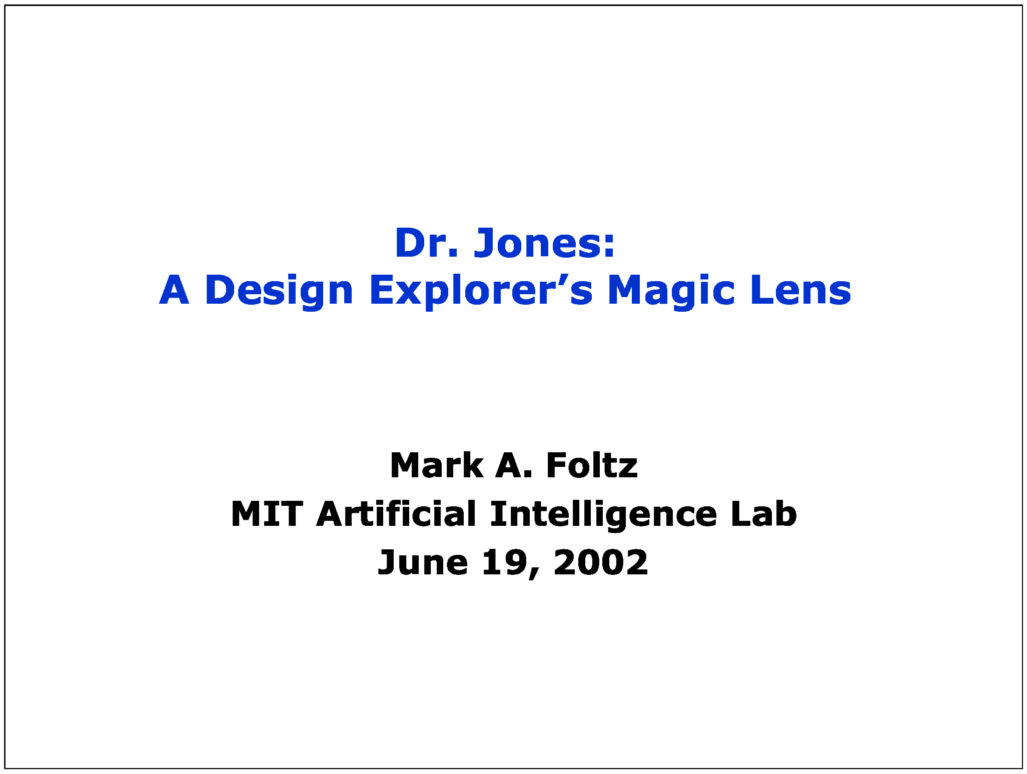
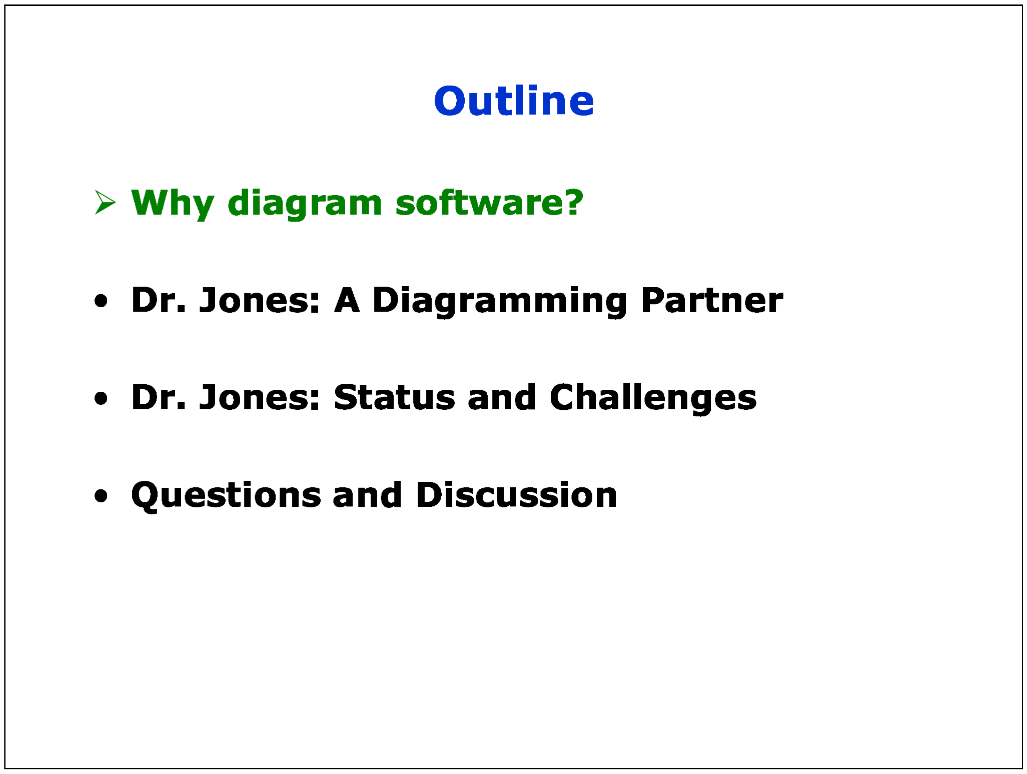


Dr. Jones: A Design Explorer's Magic Lens

Mark A. Foltz
MIT Artificial Intelligence Lab
June 19, 2002

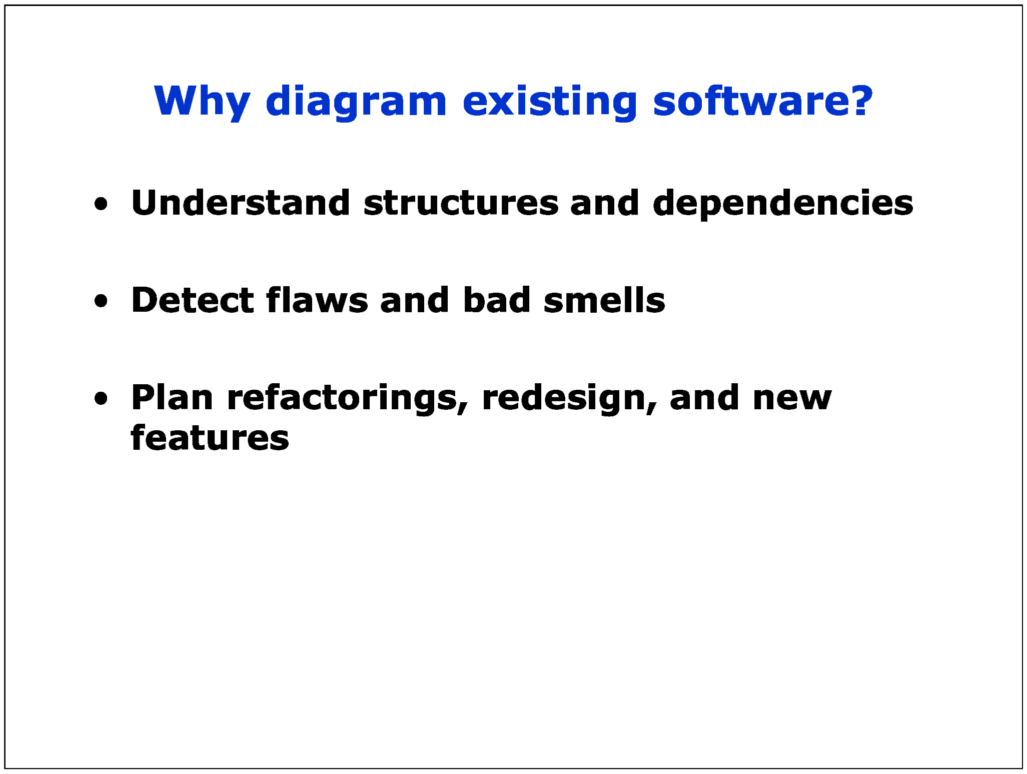


Outline

- ➢ **Why diagram software?**
- **Dr. Jones: A Diagramming Partner**
- **Dr. Jones: Status and Challenges**
- **Questions and Discussion**

The agenda for today's meeting is my progress on thesis research on software diagramming as an aid for the understanding and redesign of existing software.

First I'll motivate the work by describing some of the reasons we diagram software.



Why diagram existing software?

- Understand structures and dependencies
- Detect flaws and bad smells
- Plan refactorings, redesign, and new features

It is commonly accepted that software diagramming is an invaluable aid for software engineers.

It helps them understand their existing software and explain it to others.

It points out problems that are hard to infer from the source code alone.

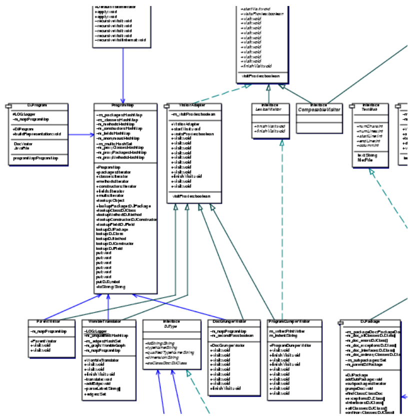
And it is the best way to plan large-scale structural changes in the software, such as refactorings, redesign, and new features.

Assisting software engineerrs in these tasks is the main interest of my research.

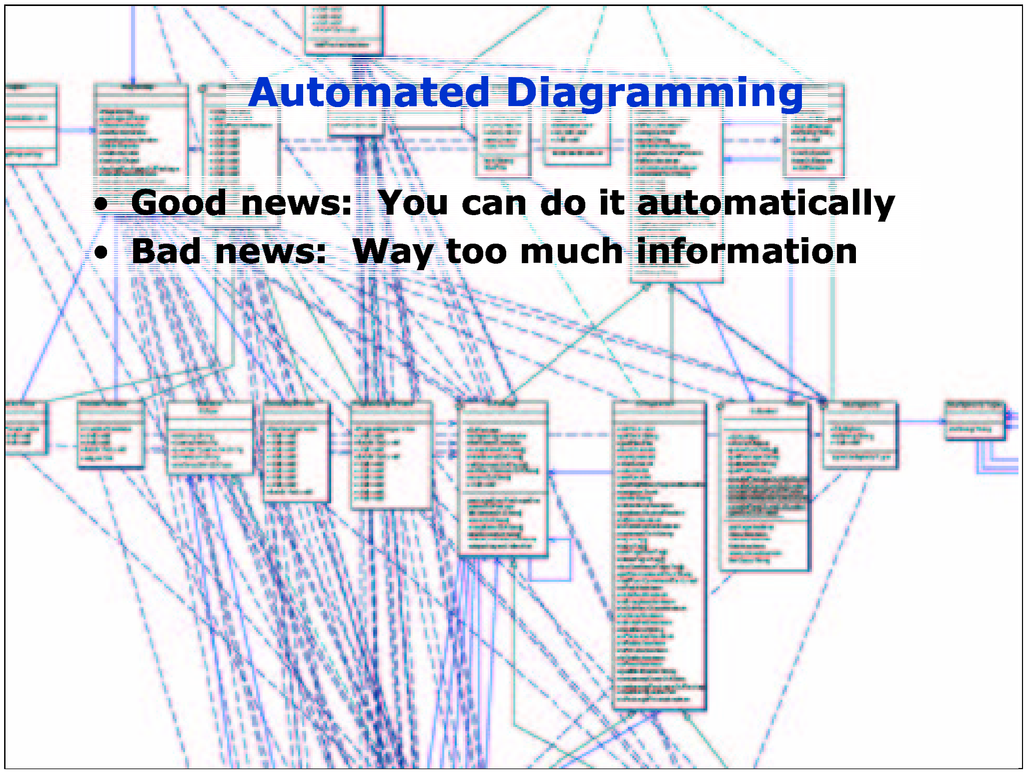
Diagrams let the programmer step back from the code and think abstractly and hypothetically about the design.

Automated Diagramming

- **Good news: You can do it automatically**



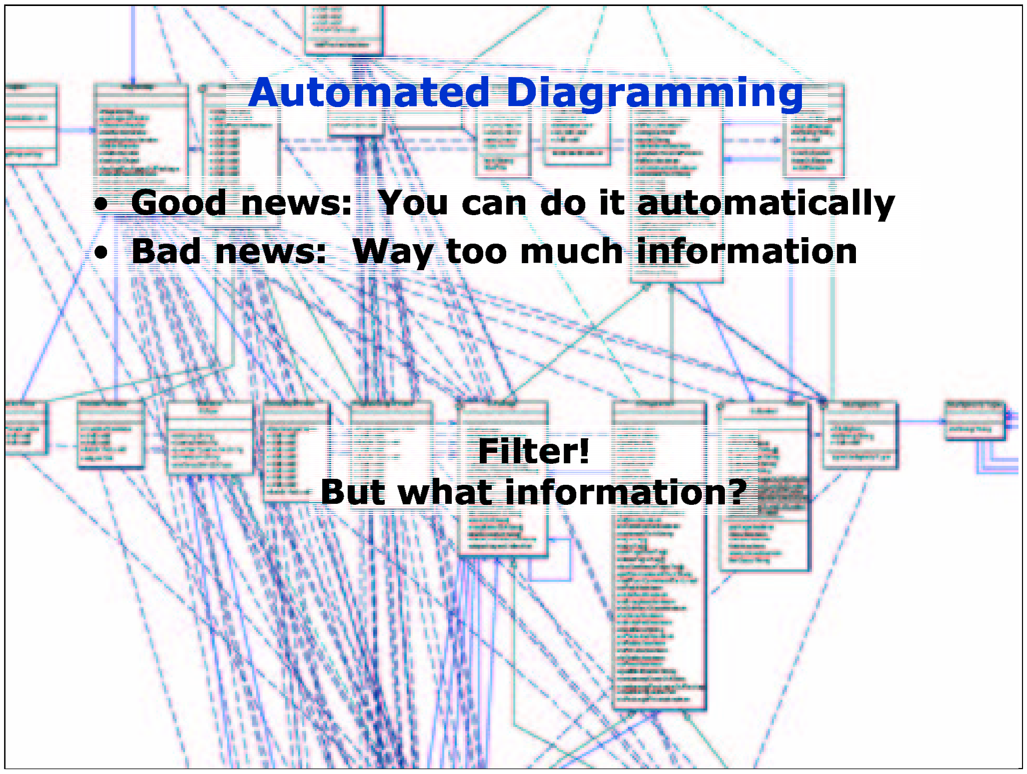
The good news is that reverse engineering existing software into diagrams is not that difficult, and modern IDEs like TogetherSoft and Rational Rose provide this capability.



Automated Diagramming

- **Good news: You can do it automatically**
- **Bad news: Way too much information**

The bad news is that there is often way too much information in these diagrams. Diagramming everything results in large, complex diagrams, like the one you see here, that are not very usable. Moreover, there are so many kinds of relationships in software that's it's hard for the tool to guess which relationships designers need to make design decisions. These one-diagram-fits-all approaches don't seem to work very well.



Automated Diagramming

- Good news: You can do it automatically
- Bad news: Way too much information

Filter!
But what information?

We can reduce these diagrams’ complexity by filtering the information in them.

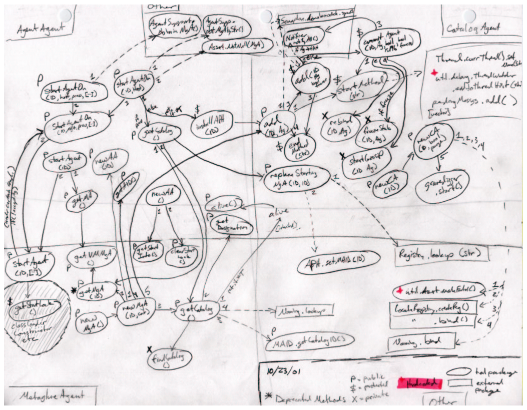
For example, TogetherSoft first shows everything, and then lets the programmer to filter out the parts of the diagram he doesn’t want to see.

But this is not a very natural way of getting to the diagram the programmer wants.

Instead, if the tool knew **why** the programmer wanted the diagram, it could create a diagram with only the relevant information.

I’d like to explore the approach that starts with nothing and tries to show the programmer only what is relevant, instead of starting with everything and asking him to filter out what’s not.

Pen-and-Paper Diagramming

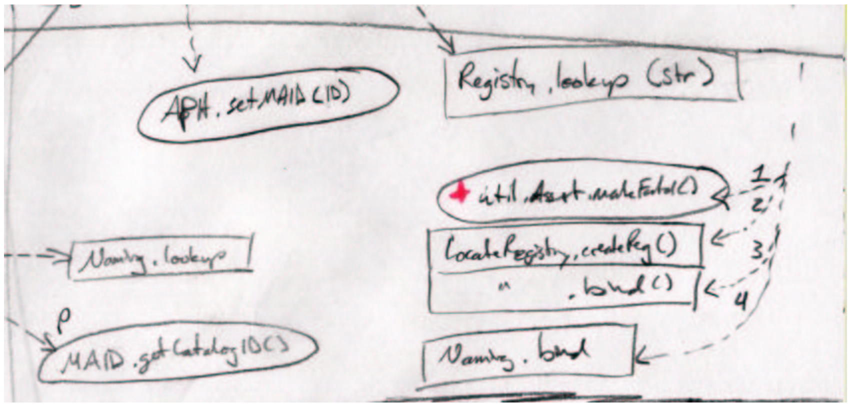


This is similar to what a programmer might do if they didn't have a diagramming tool, and were drawing software diagrams with pen and paper.

Or perhaps printing out a manually filtered diagram from an IDE and annotating it.

What are the qualities of these pen-and-paper diagrams that make them useful?

Pen-and-Paper Diagramming



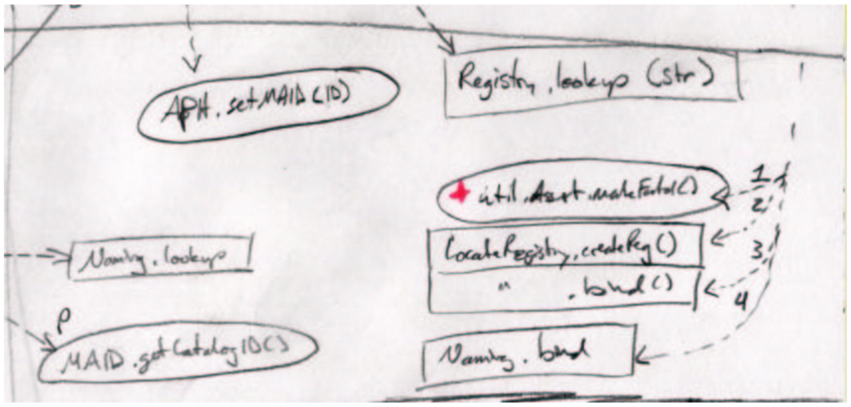
- **Specific to design intentions.**

Unlike the IDE diagrams, pen-and-paper diagrams are very specific to the programmer's design intentions, both in the information they contain and the way it is presented.

Their marks convey facts about the program, problems that need attention, and design intentions to fix and enhance the program.

For example, in this closeup the programmer has denoted hard-coded values with a red star which will need to be fixed later.

Pen-and-Paper Diagramming

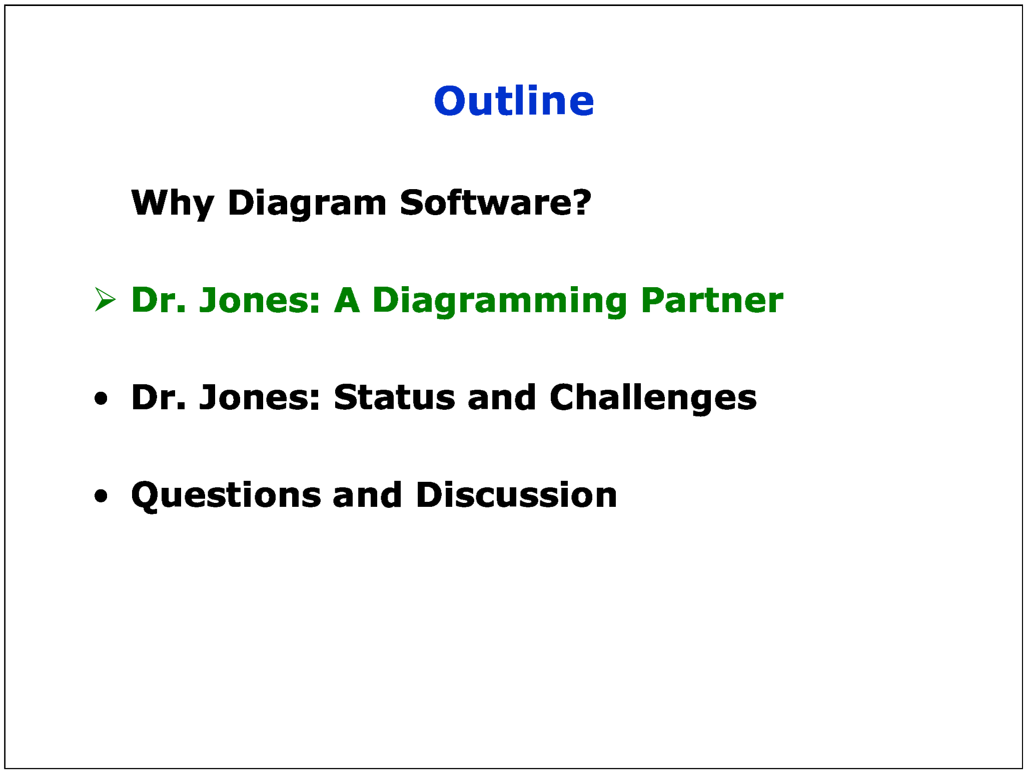


- **Specific to design intentions.**
- **Isolates relevant parts of the program.**
- **Records problems and intentions (to-dos).**
- **Flexible, but static.**

Two key aspects make these diagrams useful. First, they isolate the parts and relationships in the program relevant to the intended changes, which keeps them simple. This makes it easy to visually reason about the current and future state of the software, which has been called ``reasoning in the diagram''. Although the programmer might be aware of other parts of the program which might be impacted, they aren't necessarily represented, again to keep the diagram simple.

The second aspect is that they record problems and intended changes in the program, so they can serve as a visual `to-do' list when later implementing the changes.

These hand-drawn diagrams are very flexible, but take effort to produce, may be an inaccurate, and can't easily represent redesigns (unless the programmer goes to the trouble of drawing multiple diagrams).



Outline

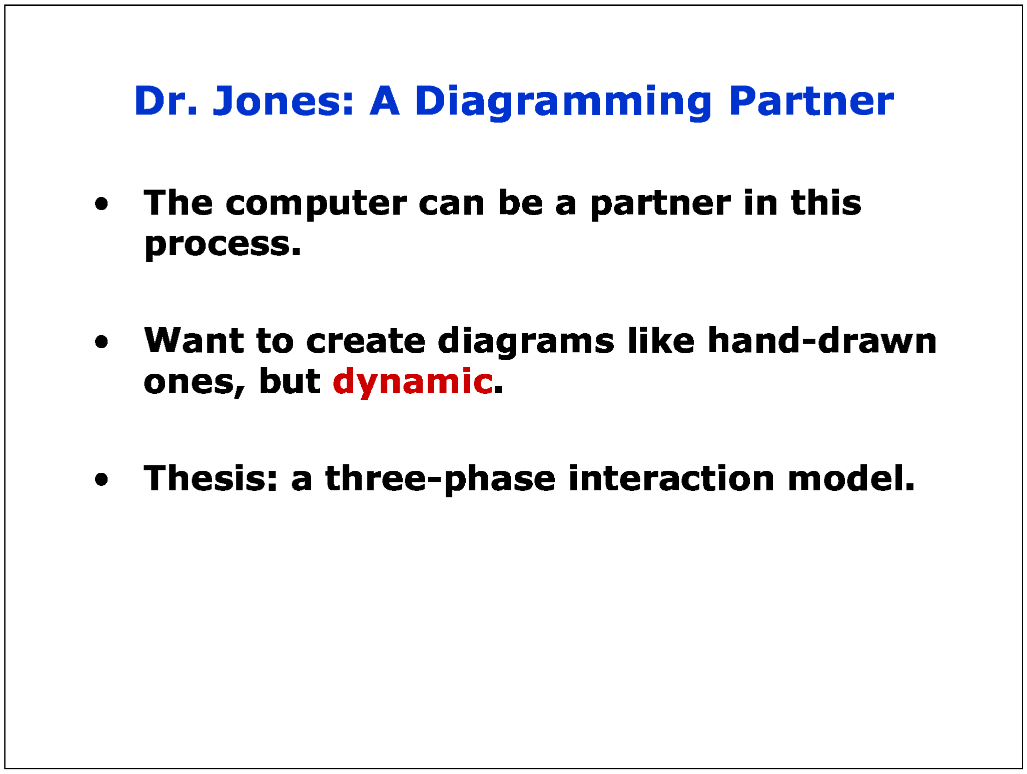
Why Diagram Software?

➢ Dr. Jones: A Diagramming Partner

- Dr. Jones: Status and Challenges
- Questions and Discussion

My research proposes a tool, Dr. Jones, that combines automated diagramming with the focus and task-awareness of hand-drawn diagrams.

Dr. Jones' goal is to collaborate with the programmer to explore the design family of a Java program.

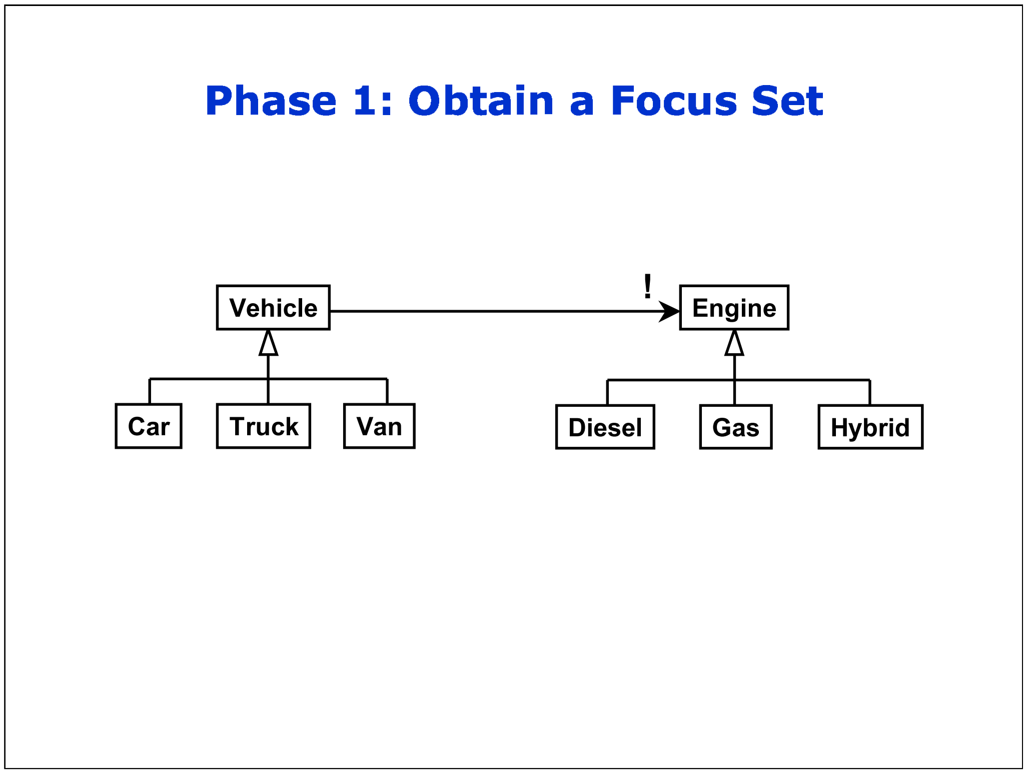


Dr. Jones: A Diagramming Partner

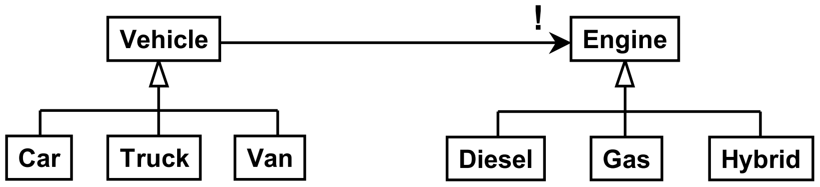
- **The computer can be a partner in this process.**
- **Want to create diagrams like hand-drawn ones, but dynamic.**
- **Thesis: a three-phase interaction model.**

My thesis is that Dr. Jones should play the role of a semi-intelligent diagramming partner in software design exploration. Dr. Jones knows about software structures, how to diagram them, and some of the ways they might be changed, but not enough to make design decisions – that's up to the programmer. Dr. Jones automates the drawing of the diagrams, and, like hand-drawn ones, they contain task-specific information and remind the programmer of the changes he intends to make.

But, unlike hand-drawn diagrams, they are dynamic: they can show redesigns and record multiple design alternatives. I propose a three-phase model of interaction between Dr. Jones and the programmer, which I believe will accomplish this goal.



Phase 1: Obtain a Focus Set

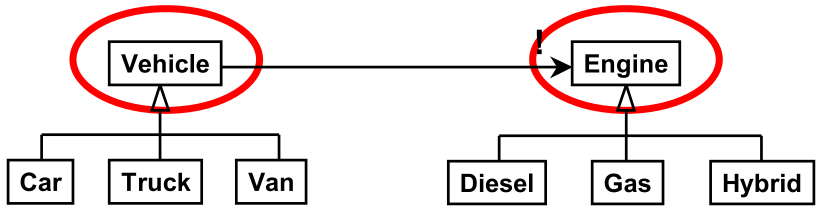


The first part of the interaction lets the programmer to choose what parts of the program he would like to work on while browsing an overview of the program.

This high-level overview of the program's classes doesn't have deep detail, but shows the is-a and has-a relationships among the classes, like a UML object model.

This example is a fragment of the object model for a program that deals with vehicles.

Phase 1: Obtain a Focus Set



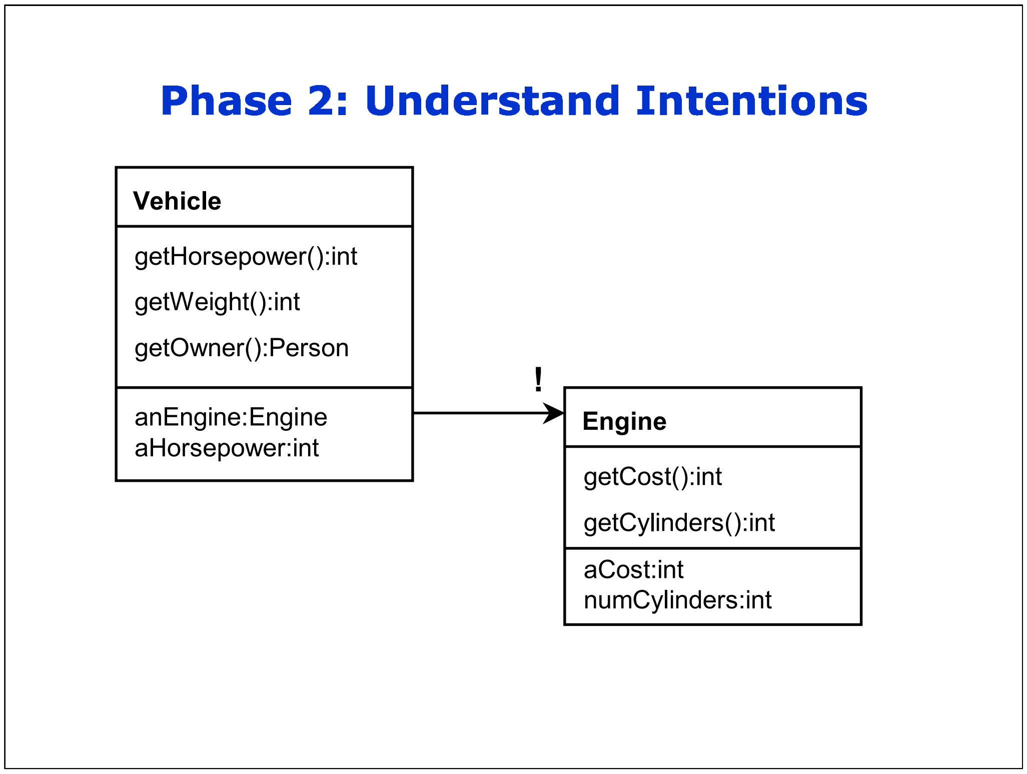
Focus Set = Focus of Attention

The chosen classes constitute the initial focus set for Dr. Jones. The focus set is a small number of program elements at the focus of attention for the programmer and Dr. Jones at any one time. It's Dr. Jones job to maintain and diagram this focus set as the programmer explores designs.

In this case the programmer wants to look at the Vehicle and Engine base classes.

Focus+context views, showing more detail on selected classes, could help the programmer understand the program and make a good choice for this focus set.

Also, the visualization of ``bad smells'' in the code would be useful here.



Phase 2: Understand Intentions

Dr. Jones then diagrams the focus set in a variant of a UML object model. These diagrams contain the structural elements the programmer

might want to manipulate, like classes, methods and properties, as well as dependencies among them, in this case a has-a relation between Vehicle and Engine.

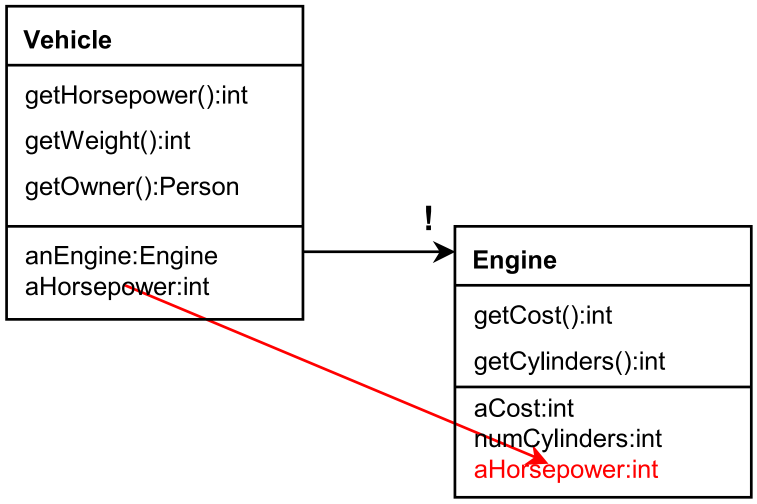
The next phase of the interaction is for the programmer to tell Dr. Jones what he wants to do to the program.

This is the crucial step in the interaction, because how well Dr. Jones understands these intentions will determine how well it can diagram the software.

Fortunately, there is a common way emerging for describing how to change programs (I.e. fowler's refactorings).

Dr. Jones will be able to interpret intentions based on a vocabulary of these refactorings (to be described later).

Phase 2: Understand Intentions



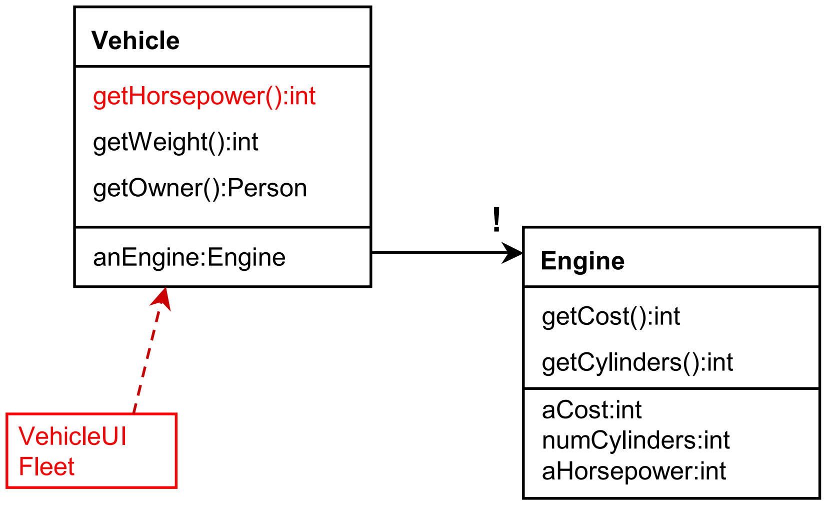
Here, the programmer decides that ``horsepower'' is really a property of an Engine, not a Vehicle, and indicates the Move Field refactoring.

I haven't committed to a UI modality for this kind of interaction; a conventional WIMP technique like drag-and-drop would work, although I'd also like to explore gestural or multimodal techniques, keeping with the metaphor of interacting with a pen-and-paper diagram.

(Of course this brings up all the complexities of recognition-based user interfaces.)

Dr Jones records this intention and …

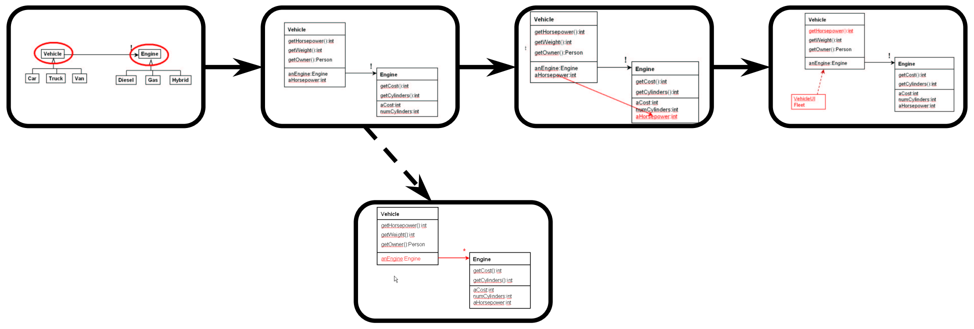
Phase 3: Update Focus Set, Diagram



… transforms the diagram to show the effect of the refactoring, updates the focus set, and diagrams this new focus set. The focus set now includes the participants in the refactoring and its indirect effects, which are program elements that may require further changes to produce a valid program. Here, moving horsepower to Engine causes getHorsepower() in Vehicle to refer to a missing field, and in turn clients that depend on getHorsepower() may need further change. (The likely solution is to delegate getHorsepower() to the anEngine object, so clients won't have to change.)

It's important to note is that Dr. Jones doing `virtual refactoring.' It's transforming the diagram, not the program – it's a design exploration tool, not a software transformation tool. That's because that actually doing the transformation may involve asking the programmer lots of questions he doesn't want to worry about when thinking about design.

Dr. Jones: A Diagramming Partner



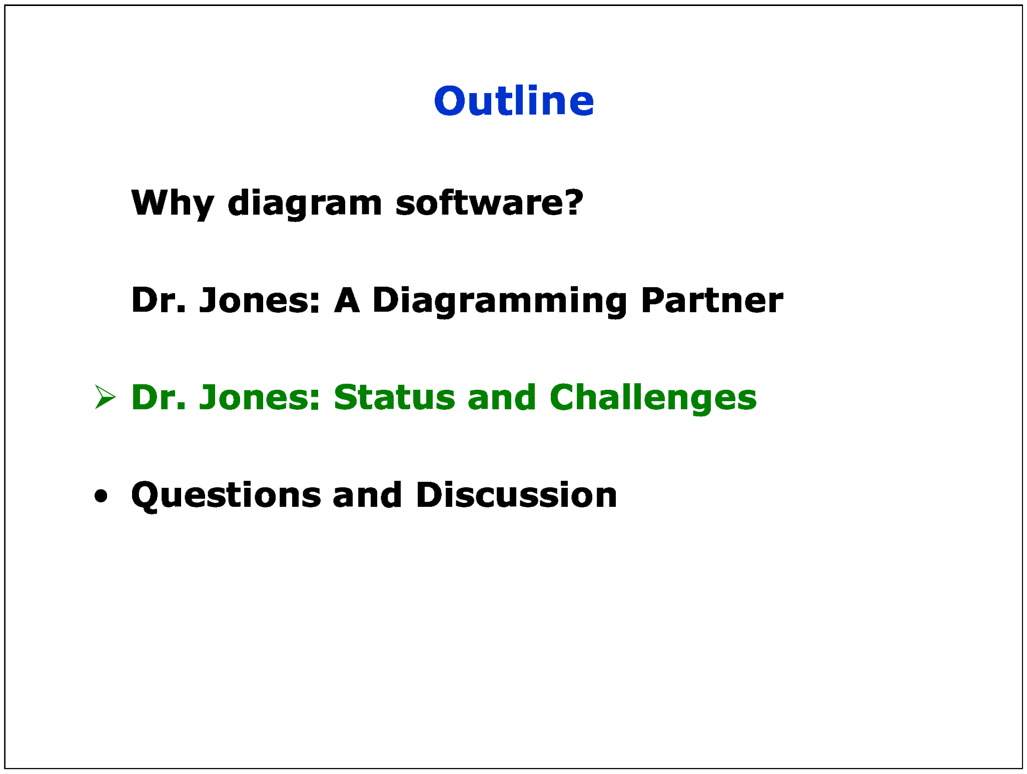
To summarize, Dr. Jones collaborates with the programmer in diagramming a program and its design alternatives.

It maintains the focus set of program elements throughout a design dialogue with the programmer.

The focus set contains only the relevant elements at each step, so the diagrams stay simple.

These focus sets can be kept in a history so that the programmer can revisit previous steps and branch to explore alternatives.

This could be an opportunity for design rationale capture, although that's not what I'm focusing on in this thesis.



Outline

Why diagram software?

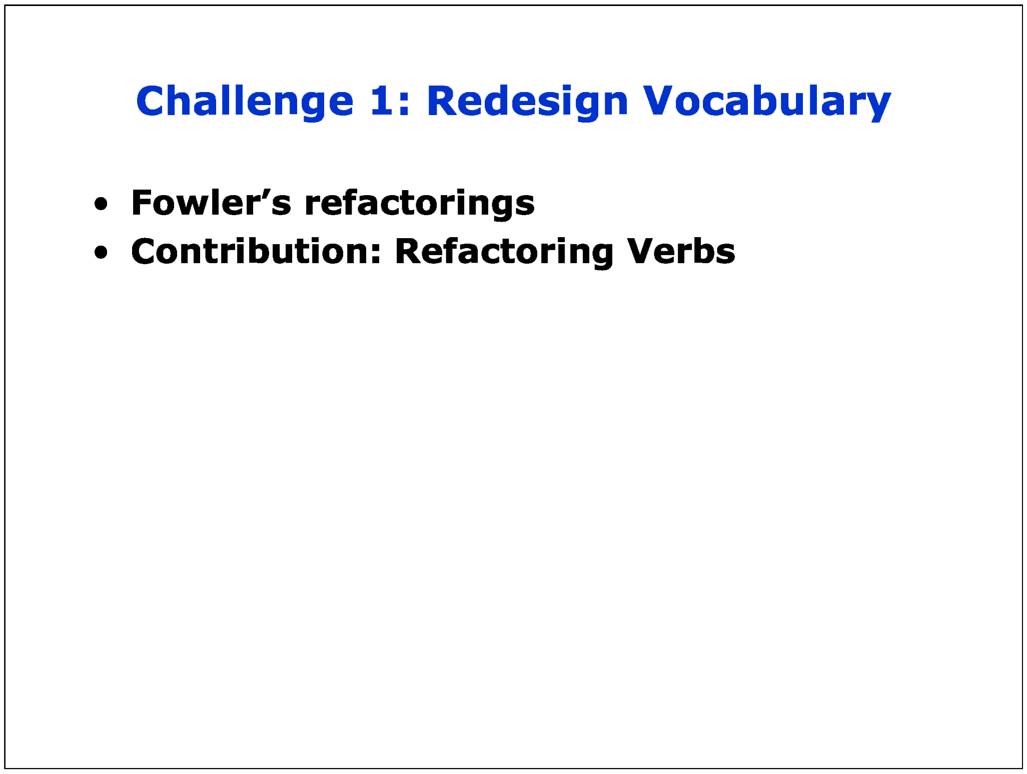
Dr. Jones: A Diagramming Partner

➢ Dr. Jones: Status and Challenges

- Questions and Discussion

Now I will discuss some of the specific research challenges in realizing Dr. Jones and the progress of a prototype implementation.

Each of the challenges points to a contribution this research intends to make.



Challenge 1: Redesign Vocabulary

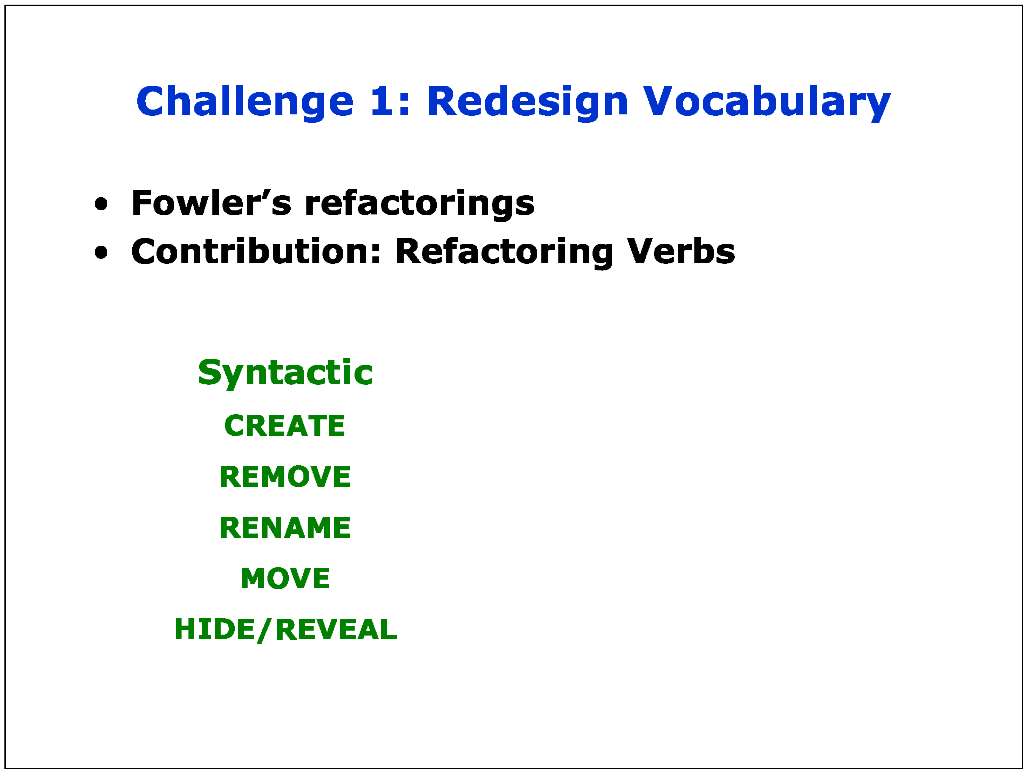
- **Fowler’s refactorings**
- **Contribution: Refactoring Verbs**

The first challenge is finding a way for the programmer to describe to Dr. Jones how he wants to change the program.

My starting point is Martin Fowler’s list of 72 common ways object oriented programs can be refactored.

I’ve tried to take his list and find a group of underlying ``refactoring verbs’’ that cover many of the cases he describes.

Each of these verbs can be applied to multiple kinds of program structures, resulting in a more economical vocabulary.



Challenge 1: Redesign Vocabulary

- **Fowler's refactorings**
- **Contribution: Refactoring Verbs**

Syntactic

CREATE

REMOVE

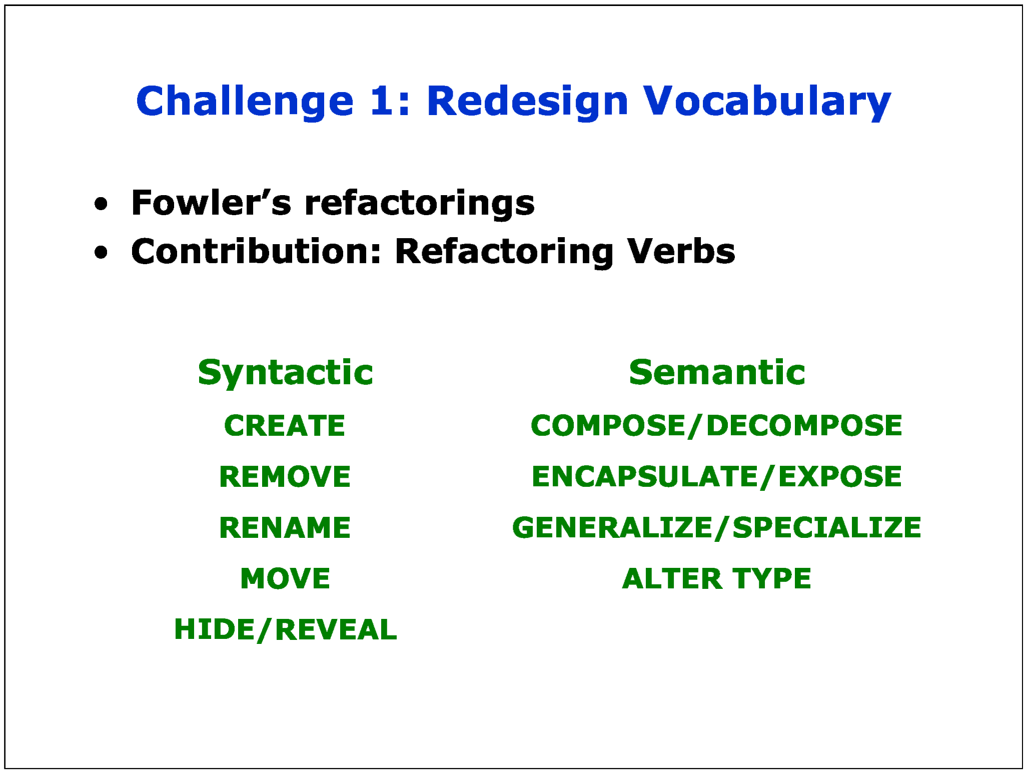
RENAME

MOVE

HIDE/REVEAL

The first set of verbs are syntactic – they are concerned with the lexical and nominal relationships among program structures.

Hide and reveal manipulate visibility modifiers, I.e. public, protected, private.



Challenge 1: Redesign Vocabulary

- **Fowler's refactorings**
- **Contribution: Refactoring Verbs**

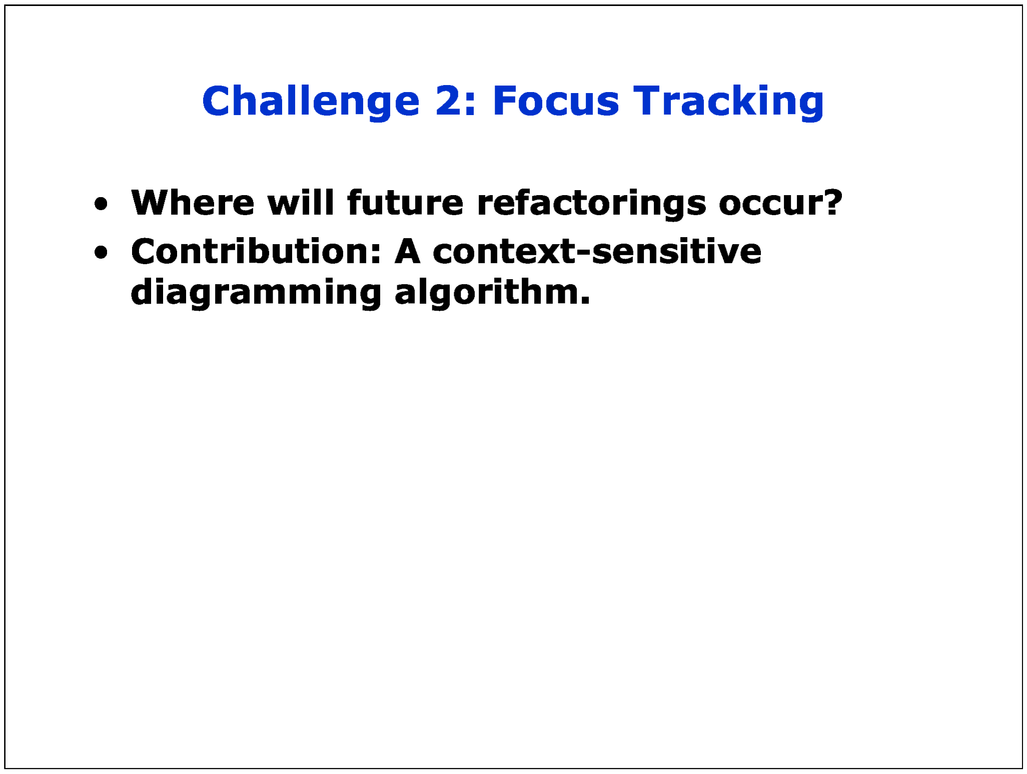
Syntactic	Semantic
CREATE	COMPOSE/DECOMPOSE
REMOVE	ENCAPSULATE/EXPOSE
RENAME	GENERALIZE/SPECIALIZE
MOVE	ALTER TYPE
HIDE/REVEAL	

The second set are semantic – they deal with the set of concepts modeled by the program design.

However, the distinction between syntactic and semantic is more organizational than formal at this point.

The majority of Fowler's refactorings can be restated in terms of one or more of these verbs, and the resulting verb-plus-nouns representation of design change more naturally fits into the interaction model I have proposed.

These concepts also can apply to other programming languages, although specific meaning is language-dependent.



Challenge 2: Focus Tracking

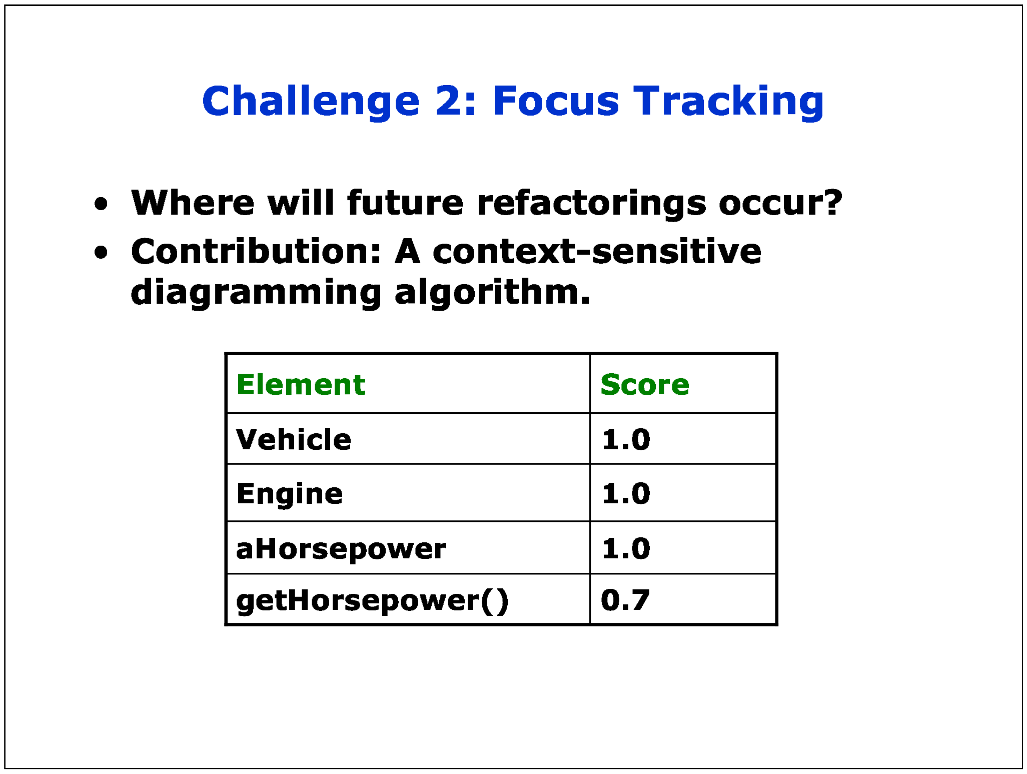
- **Where will future refactorings occur?**
- **Contribution: A context-sensitive diagramming algorithm.**

The next challenge is to track the programmer's focus of attention as he modifies the design.

Each new focus set should let the programmer see the results of the last change. These places are likely spots for where the next refactoring will occur.

To derive a good focus set, Dr. Jones will have to reason about where the programmer's next design moves might take place.

One way is fpr Dr. Jones to recognize or be told that a multi-step refactoring is taking place (I.e., applying a design pattern like the Observer pattern).

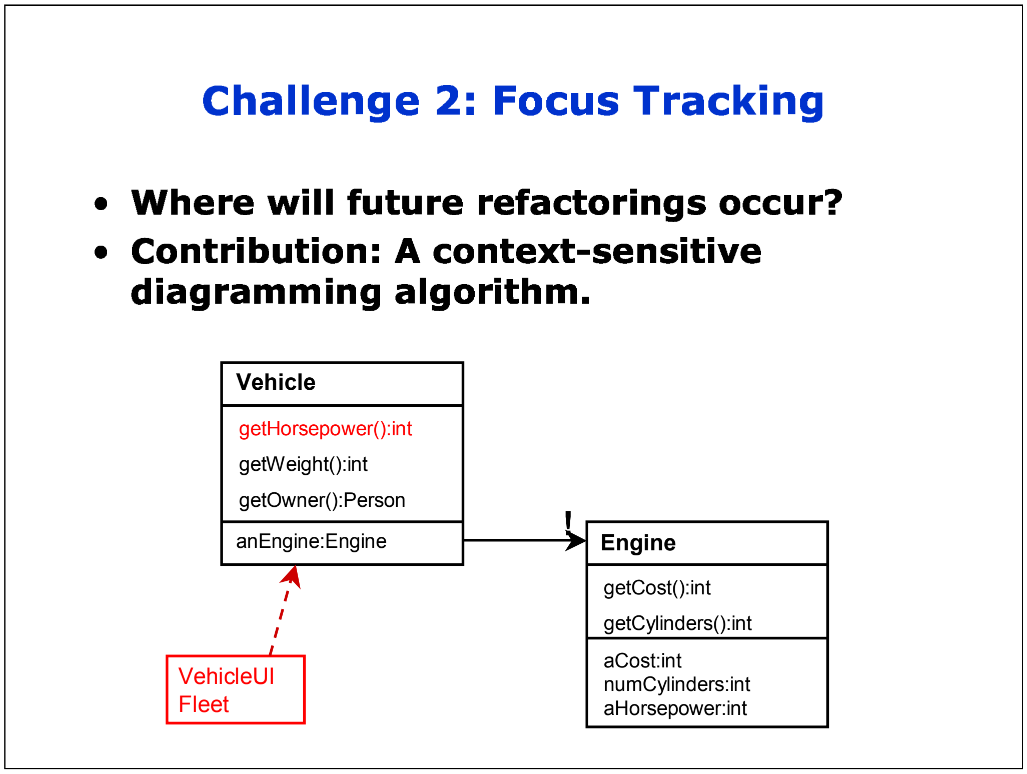


Challenge 2: Focus Tracking

- **Where will future refactorings occur?**
- **Contribution: A context-sensitive diagramming algorithm.**

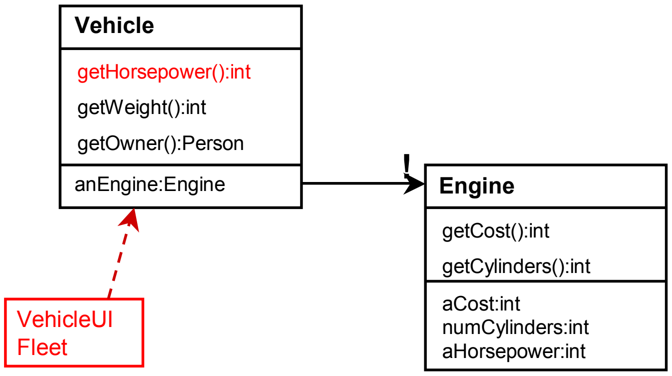
Element	Score
Vehicle	1.0
Engine	1.0
aHorsepower	1.0
getHorsepower()	0.7

Another way is to use the heuristic that a refactoring's effects will be likely sites for further refactoring. My current algorithm sketch has each refactoring specifying weights for related program elements and dependencies, reflecting their relative importance to the programmer. Each element can be rendered at certain levels of detail, and elements with higher weights are given more screen space. Screen space is allocated to maximize a weight*space score.



Challenge 2: Focus Tracking

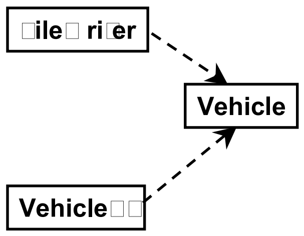
- Where will future refactorings occur?
- Contribution: A context-sensitive diagramming algorithm.



It's a variant of the bin packing problem, but the final size of the diagram is dependent on the graph layout algorithm, which makes it harder.

Challenge 3: The Crystal Ball Problem

- **Clear picture at the beginning …**
- **But it begins to get cloudy.**



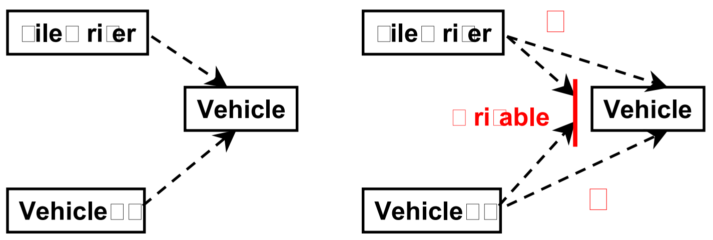
The third challenge is what I call the crystal ball problem. At first, Dr. Jones can get a clear idea of the program's structure by examining the program text.

But as the programmer makes changes, the design evolves away from the original program and it becomes more and more difficult to predict the dependency structure of the program. This is because some refactorings can result in multiple possible dependency structures.

For example, supposed the programmer wanted to decompose the Vehicle class into Writable, an interface for objects that can be written to files.

Challenge 3: The Crystal Ball Problem

- Clear picture at the beginning …
- But it begins to get cloudy.



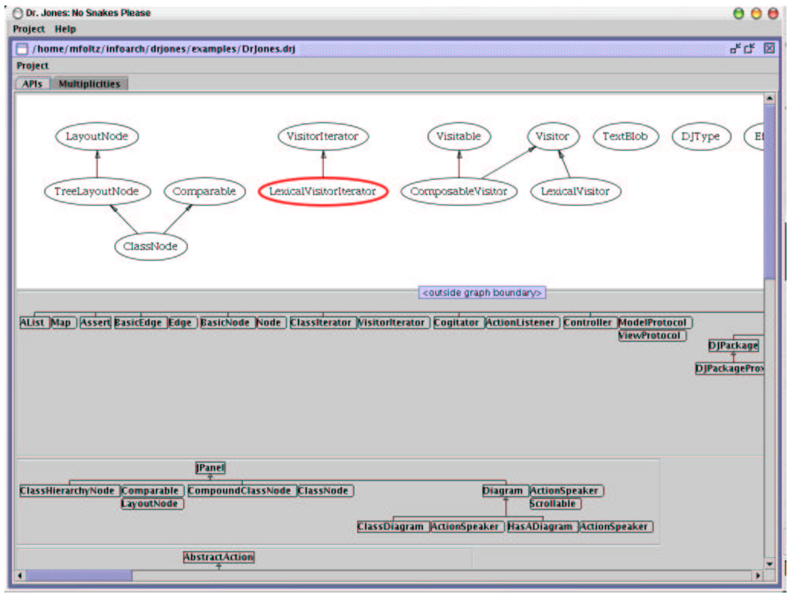
Now every existing Vehicle dependency has the choice of using the old Vehicle interface, or the new Writable interface.

We could ask the programmer to make these choices explicitly, or assume that things should stay the way they are.

I don't have a good solution to this problem -- yet.

At least, the programmer could alternate between Dr. Jones and a source-code editor to implement changes periodically, then refresh Dr. Jones.

Dr. Jones: Status



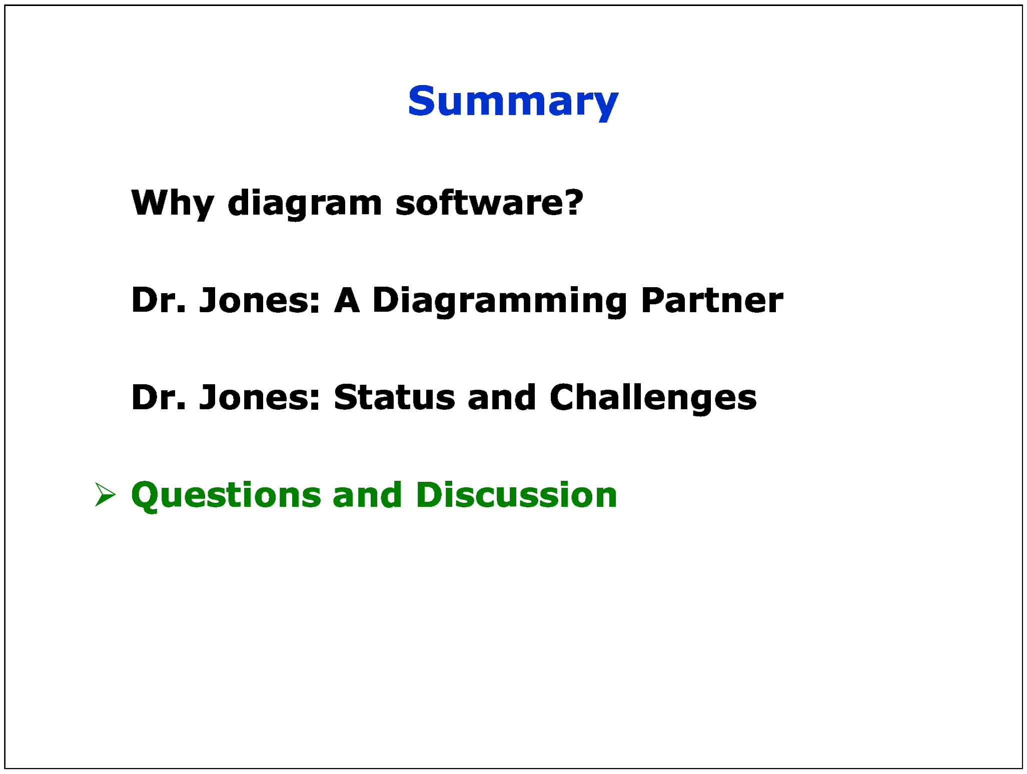
These three challenges form the gist of my next lines of research.

For now, I'd like to briefly show the Dr. Jones prototype and talk a little about its implementation.

Dr. Jones can draw an object model of most Java programs. More detailed nformation is available to it, but only class names are shown in the UI.

Information from the JavaDoc documentation generator and Allison's Superwomble is merged into a unified representation.

Visitor classes traverse this representation to generate diagrams of the program. Some are created with layout engines and components I've written, others use the graphviz and grappa tools from at&t research.



Summary

Why diagram software?

Dr. Jones: A Diagramming Partner

Dr. Jones: Status and Challenges

➢ Questions and Discussion

To summarize:

I claim that diagramming software is useful for understanding and planning structural changes, but

current tools use a one-diagram-fits all approach without asking the programmer what they want to do to the program.

Dr Jones collaborates with the programmer in exploring the program's design by understanding an economical vocabulary of refactorings.

By doing so, it can create focused, relevant and simple diagrams.

I haven't wrote out a formal research plan, but my medium-term goals would be to put in writing my answers to challenges 1, 2, and 3 (as formally as is needed) while working on the implementation of Dr. Jones.