Data Preparation 1

Put Danny's XML .refactorings data into the following tables:

Table: eclipse_dev_tool_usage

refactoring_id	userid	description	project	flags	refactoring_name	timestamp	comment
1	pdobrev	Extract method 'packLater'	org.eclipse...	589830	org.eclipse. jdt.ui.extract. method	1/1/2007	Extract method...

*   make different table with project names?

Table: eclipse_dev_tool_usage_attributes

refactoring_id	attribute	value
1	visibility	2

Hypothesis 13: eclipse devs different from other devs

> The kinds of refactorings done with tools by Eclipse core developers differ from those done by Eclipse users.

Produce a histogram of refactorings from Danny's data set using:

```
SELECT refactoring_name,COUNT(refactoring_name) eclipse_dev_tool_usage
          GROUP BY refactoring_name
```

Produce a similar histogram using Gail Murphy's data set. Compare the results, possibly performing statistical test.

Cross Validation

Compare to a histogram from the Eclipse Usage Collector.

Hypothesis 4: Tool Configuration

> Programmers do not provide configuration information for their refactoring tools frequently.

Needs *eclipse_dev_tool_usage*

1. Divide refactorings into tool type (from 'id' attribute).

2. For each tool type, divide instantiations into the following buckets:
   - No configuration (programmer just accepts refactoring defaults)
   - Configuration by name-only (programmer changes new element names)
   - Other configuration (programmer changes any other configuration)
- Report the results as stacked, 100% bar chart, with 1 bar for each tool.
- The hypothesis predicts that "other configuration" will be quite small for each tool.

Hypothesis 9: Refactoring Grouping

> Refactorings of the same type often occur close together temporally, such as two back-to-back Extract Local Variable refactorings.

Needs *eclipse_dev_tool_usage, eclipse_dev_tool_usage_attributes*

1. Divide refactorings into tool type.
2. Report what percentage of same-kind refactorings appear within a 1-minute window of each other for all groups combined.
3. The hypothesis predicts that most refactorings will appear in windows (this was *almost* true in Murphy's data set).

Cross-validation

This has already been cross-validated with Murphy's data set. If I recall correctly, about 48% of refactorings appeared as part of a group.

Data Preparation 2

Next we need to inspect actual source code to infer refactorings. In preparation, we'll have to find the range of dates for which we have logs, for each developer:

```
SELECT userid,project,MIN(timestamp) as firstRefactoringToolUsage,
        MAX(timestamp) as lastRefactoringToolUsage
        FROM eclipse_dev_tool_usage
        GROUP BY userid,project
```

View: eclipse_dev_date_range

userid	project	firstRefactoringToolUsage	lastRefactoringToolUsage
pdobrev	org.eclipse...	12:00:00 12/1/2007	12:00:00 6/1/2008

Hypothesis 3: decreased refactoring before code release

> The quantity of refactorings decrease a code release approaches.

Use *eclipse_dev_date_range* to find some milestone(s) in the Eclipse release, hopefully both major and minor ones that fall within the refactoring tool range. Create a time plot with *eclipse_dev_tool_usage* with time buckets on the x axis and refactoring counts on the y axis. There may be a more conise way to summarize this, but this will at least let us explore patterns.

Cross-validation

This may be cross-validated with Xing and Stroulia's data set, but may be somewhat unbelievable because they both apply to Eclipse.

Weissgerber's data should be quite promising, though. In fact, I've already begun this in the query "Refactorings Query," without yet looking at release points. It should just require some fiddling.

Data Preparation 3

Then, based on *eclipse_dev_tool_usage*, find which projects each developer committed to, and the commit dates for each project based on CVS history:

Table: eclipse_dev_commits

userid	**timestamp**	**version**	**project**	**file**	**comments**
pdobrev	12:00:31 12/1/2007	1.1	org.eclipse.core	src/edu/.../ Workspace.java	Renamed big class...

*   *eclipse_dev_commits*.project is not necessarily the same as *eclipse_dev_tool_usage*.project (the programmer may have renamed the project after checking it out), but we'll cross that bridge when we come to it

View: eclipse_dev_global_commits

commit_id	**userid**	**timestamp**
1	pdobrev	12:00:31 12/1/2007

We'll also have to figure out how many comments are tagged as refactorings. We can do something clever, or we can just use Ratzinger's 12 or so keywords, again using an SQL query to make two views (SELECT commit_id FROM eclipse_dev_global_commits WHERE ...):

View: eclipse_dev_labeled_commits and *eclipse_dev_unlabeled_commits*

commit_id
1

We then calculate the proportion of unlabeled commits to those that are labeled, say the proportion is 2:1. Then, we'll take a randomly select 20 (userid,timestamp) pairs from *eclipse_dev_labeled_commits* (this number depends on the ratio, as well as what is a reasonable total number to inspect), and take 20*(2:1) = 40 randomly selected from *eclipse_dev_unlabeled_commits*. (Is there a way to do this in SQL? maybe, ask google) This is known as a stratified sample.

Then, we do the hard work of actually doing the comparison. For a given timestamp *t*, this involves checking out all projects associated with *t* at that timestamp, then the same projects at *t-1*, then comparing the two together. This study could be nicely blinded, so that the person doing the comparison doesn't know whether or not it was labeled as "refactoring."

During the comparison, we will look for refactorings manually, noting roughly where each refactoring originated (so that we can return to refactorings later, if needed):

Table: eclipse_dev_inspected

commit_id	**refactoring_name**	**project**	**file**	**line_number**	**notes**
1	org.eclipse.rename.temp	org.eclipse.core	Workspace	123	This was a little refactoring!
1	org.eclipse.push.up	org.eclipse.jdt.core	BinaryMethod	56	Pushed up to 2nd superclass
2	swap.statements	org.eclipse.core	Workspace	658	Not implemented by any tool

Also, for each commit, we'll note whether a root canal refactoring took place (pure refactoring). If it did, we record nothing, but if it didn't we record the actual line number of some non-refactoring:

Table: eclipse_dev_nonrefactorings

commit_id	**project**	**file**	**semantics_change_line_number**
2	org.eclipse.core	Workspace	658

Hypothesis 5: Low-level Refactorings

Refactorings below the method level account for the majority of refactorings.

Create a new table:

Table: refactorings

refactoring_name	**level**
org.eclipse.push.up	high
swap.statements	low

*we may later want to change level to a ranking, or a category, such as expression, statement, method, class, package, etc.
** add an id, and refer to id in

Create an SQL query that counts the number of low and high level refactorings in *eclipse_dev_inspected*. Report the proportion.

Hypothesis 1: Automated vs. Manual Refactoring (quantity)

> Many refactorings are performed manually, without a refactoring tool, even when one is available.

If we wanted to be tricky, we would try to associate tool-refactorings with inspected-refactorings, but this is probably too much work (however, it would be a good sanity check). Instead, we'll just count the refactorings per commit with tools and in total:

```
ToolRefactoringsPerCommit = COUNT(eclipse_dev_tool_usage) /
                            COUNT(eclipse_dev_global_commits)
TotalRefactoringsPerCommit = COUNT(eclipse_dev_inspected) /
                            COUNT(SELECT UNIQUE commit_id FROM eclipse_dev_inspected)

ToolUsageRatio = ToolRefactoringsPerCommit / TotalRefactoringsPerCommit
```

Hypothesis 2: Automated vs. Manual Refactoring (types)

> The variety of refactorings performed with tools do not reflect the variety performed without tools.

Essentially, we do the same thing as in hypothesis 1, but we separate by tool type. The SQL is more complicated, but the view ends up like this:

View: tool_types

refactoring_name	**is_automated**	**count**
org.eclipse.push.up	true	33
org.eclipse.push.up	false	430
swap.statements	false	16

This can then be summarized as a bar chart.

Hypothesis 10: 'refactor' comment: more refactoring?

> Commits labeled “refactor” do not indicate significantly more refactoring instances than those without the label.

RefactoringsPerLabeledCommit = COUNT(SELECT * FROM
eclipse_dev_labeled_commits,*eclipse_dev_inspected*) /
COUNT(*eclipse_dev_labeled_commits*)

RefactoringsPerUnLabeledCommit = COUNT(SELECT * FROM
eclipse_dev_unlabeled_commits,*eclipse_dev_inspected*)/COUNT(*eclipse_dev_unlabeled_commits*)

For hypotheses 10-12, looking at a histogram instead of a scalar value is probably a more honest representation of the data.

Hypothesis 11: 'refactor' comment: more root canal?

> Commits labeled “refactor” indicate significantly fewer nonrefactoring changes than those without the label.

LabeledNonRefactorings = SELECT * FROM *eclipse_dev_labeled_commits* ,
eclipse_dev_nonrefactorings

UnlabeledNonRefactorings = SELECT * FROM *eclipse_dev_unlabeled_commits* ,
eclipse_dev_nonrefactorings

RootCanalRefactoringsPerLabeledCommit =
(COUNT(*eclipse_dev_inspected*.commit_id) -
COUNT(LabeledNonRefactorings.commit_id)) / COUNT(*eclipse_dev_labeled_commits*)

RootCanalRefactoringsPerUnLabeledCommit =
(COUNT(*eclipse_dev_inspected*.commit_id) -
COUNT(LabeledNonRefactorings.commit_id)) / COUNT(*eclipse_dev_unlabeled_commits*)

FlossRefactoringsPerLabeledCommit =
COUNT(LabeledNonRefactorings) / COUNT(*eclipse_dev_labeled_commits*)

FlossRefactoringsPerUnLabeledCommit =
COUNT(UnlabledNonRefactorings) / COUNT(*eclipse_dev_unlabeled_commits*)

Hypothesis 12: 'refactor': more high level refactorings?

> Commits labeled “refactor” contain proportionally more global and fewer local refactorings than those without the label.

Note: I think here we meant that refactorings span multiple versions, but what's described here doesn't test that.

HighLevelRefactoringsPerLabeledCommit = COUNT(SELECT * FROM *refactorings* , *eclipse_dev_inspected*, *eclipse_dev_labeled_commits* WHERE level='high') / COUNT(*eclipse_dev_labeled_commits*)

HighLevelRefactoringsPerUnLabeledCommit = COUNT(SELECT * FROM *refactorings* , *eclipse_dev_inspected*, *eclipse_dev_unlabeled_commits* WHERE level='high') / COUNT(*eclipse_dev_unlabeled_commits*)

Data Preparation 4

Observe programmers refactorings, either by looking over their shoulder or by video/screen capturing their IDE. While doing so, we count the number of refactorings (in a systematic, repeatable way) the programmers perform, dividing refactorings into tool-based vs. automated, to create the following table:

Table: observed_refactorings

userid	**time**	**refactoring_name**	**used_tool**
mr.ed	8/15/2008 10:00:01PST	org.eclipse.extract.method	no

We also need to capture when they commit and which modules:

Table: observed_commits

userid	**time**	**module**
mr.ed	8/15/2008 10:05:00PST	org.company.product

Later, another researcher (for blindness) goes through the CVS commits (assumedly we have access!), and figures out what refactorings occurred. Produce two tables in form of *eclipse_dev_inspected* and eclipse_dev_nonrefactorings.

Open Questions:

- Do we have multiple observers?
- Do we do live or recorded?
- How many programmers do we observe?
- For how long?
- Emerson needs to update his Human-Subjects Review Committee documents. It has a waiver of review. Can Danny and Chris be added to it, and will they be 'covered' under it?

Hypothesis 6: many refactorings get squished by CVS

Refactorings hidden by other changes account for a substantial proportion of refactorings.

The results of the observed refactorings and refactorings in CVS are compared and contrasted, looking for refactorings that were detected observationally and in CVS.

Hypothesis 7: direct observation yields few results

Direct programmer observation yields few refactoring results per researcher hour.

We compare just the raw refactorings observed in refactorings observed vs refactorings in CVS.

Cross Validation

This hypothesis can be tested using just Gail Murphy's data set. We know when people did refactoring. Now we just need to aggregate time points into time periods (is there an SQL query to do this?) and measure the length of those periods for each developer, and compare that to refactorings performed in those periods to get a refactorings-per-hour measurement.

Hypothesis 8: programmers' refactoring recall

Programmers do not recall most manual refactorings, but do recall most refactoring with tools, in retrospect.

After observing the programmers and recording when they refactor, we ask them in a survey about what they recall about refactoring. Produce a another table, then compare against directly observed refactorings.

A few open questions:

- How do we compute the baseline of refactorings?
    - By live observation?
    - By video observation?
    - By looking through their CVS commits?
- Do we need to ask them live, or could we just call developers up at the end of a work day and say "tell me about the refactoring that you did today"?

Hypothesis 14: refactoring is frequent

Refactoring is performed frequently.

This is essentially a characterization of every data set we can get our hands on. I think refactorings out of CVS and observed refactorings are the most reliable sources for this hypothesis, though.

Hypothesis 15: Refactoring-to-Edit Ratio

Refactorings account for a high proportion of total program changes.

In the Murphy data set, we've got records of edits and tool-based refactorings. However, it's not clear what an "edit" means, exactly. The recorded edit event is probably too fine grained.

A more promising approach would be to go back through the CVS comparisons, and look at every character in the diffs. If the character can be directly accounted for by a refactoring, then we put that character into the "refactoring bucket," and the "edit bucket" otherwise. Then, it's just a matter of counting how many items are in the buckets.