APPENDIX A
A CASE BY CASE ANALYSIS OF AD-HOC NODES FOR JHOTDRAW 5.2

Recall that out of the 54 AD-HOC nodes that our algorithm identified for JHtoDraw5.2, we had:

- Nodes that are descendants of full/partial configuration candidate nodes, such as the 'diamonds' of Figure **??**). There are 40 such nodes
- Nodes that are false positives due to a type having two independent ancestors. There were 5 such cases
- Nodes that are 'truly fortuitous', and there are 9 of those.

In this section, we analyze those 9 cases one-by-one.

1) Node 5, whose extent consists of the classes `AttributeFigure` and `PolylineFigure`, both of which are subclasses of `AbstractFigure`. The intent of this nodes includes, 1) the entire API of interface `FigureChangeListener`, which is explained by the fact that both classes of the extent have subclasses that implement the `FigureChangeListener` interface (see Figure **??**), 2) a good subset (but not all) of the API of the interface `Figure`[1], and 3) a number of methods that are present in neither `Figure` nor `FigureChangeListener`. A good number of such methods deal with points, including `Iterator points(), Point getPointAt(int i), int findSegment(int, int), int pointCount(), int splitSegment(int,int), void addPoint(int,int), void insertPointAt(Point,int), void removePointAt(int)`, etc. A closer inspection revealed that these point-related methods are found, unsurprisingly, `PolylineFigure`, but also in a subclass of `AttributeFigure` called `PolygonFigure`. Ethymologically, the two are certainly similar: a polygon figure is certainly a special kind of polyline figure, and that is why both can be constructed/examined as sequences of points/segments. There is, clearly, an overlap between the two classes, which should have been captured at either the type level (using a Java interface) or at the implementation level (through class inheritance, or aggregation).
2) Node 18, with extent {`ChangeConnectionHandle, ConnectionHandle, ConnectionTool`}, and intent {`Connector findConnector(int x, int y, Figure f), Figure findConnectableFigure(int x, int y, Drawing drawing)`}. In this case, both `ChangeConnectionHandle`, and `ConnectionHandle` implement the `Handle` interface. However, the methods of the intent are *not* part of that interface. The class `ConnectionTool` implements the `Tool` interface, but again, the methods of the intent are not part of the interface. The methods of the intent are related to the 'connection' concept– as evident from the class and method names. JHotdraw does have the concepts of `Connector`, which is an object that attaches to a `Figure` and 'knows' the figure's connection point, and `ConnectionFigure`, which is a `Figure` that can connect two other figures (by dealing with their connectors). The two methods of the intent find a connectable figure, within a particular location within a drawing, and for a given figure (e.g. a circle), finds the connector object responsible for a particular position (e.g., the closest one of the eight cardinal connection points of circles/ellipses). It is not clear to us who should be responsible for this *connection-related* behavior. One could argue that finding a connectable figure within a drawing, given a point within the drawing, should be the responsibility of the `Drawing` type, and finding the connector near a given point within a `Figure`, should be the responsibility of `Figure`.
3) Node 19, which is a child of node 18, with extent {`ChangeConnectionHandle, LocatorHandle`}, both of which inherit from `AbstractHandle`, which implements the interface `Handle`, and whose intent includes a subset of the API of `Handle`, along with the two connection-related methods mentioned above (intent of node 18), plus an additional method: `Connector findConnectionTarget(int, int, Drawing)`[2]
4) Node 20, which is also a child of node 18, is again related to the concept of connection, with different combinatorics of intents and extents. Specifically, node 20 has extent {`ConnectionHandle, ConnectionTool`}, and the intent {`ConnectionFigure createConnection(), Connector findConnector(int x, int y, Figure f), Figure findConnectableFigure(int x, int y, Drawing drawing)`}.
5) Node 21, with extent {`StorableInput, StorableOutput`}, and intent {`void`

1. Both classes inherit from `AbstractFigure`, which implements `Figure`, and which provides *many* default implementations for the methods of `Figure`. That is why only a subset of the API of `Figure` shows up in the intent

2. Normally, as we go down the lattice, extents are reduced ('subsetted') and intents are increased ('supersetted'). By going from node 18 to 19, indeed, the intent was increased. But how is the extent of node 19 is a subset of that of node 18? Well, before the *purge*, where we remove the non-minimal nodes from the extent, they were: the original extent of node 18 was {`Handle, Tool, AbstractHandle, AbstractTool, ConnectionHandle, ChangeConnectionHandle, ConnectionTool, LocatorHandle`}. Purging the non-minimal classes in this extent eliminated `Handle, Tool, AbstractHandle, AbstractTool` *and* `LocatorHandle`, which is a superclass of `ConnectionHandle`, because *none* of these types (interfaces or classes) defined *locally* the methods {`Connector findConnector(int x, int y, Figure f), Figure findConnectableFigure(int x, int y, Drawing drawing)`}. Thus, by going down from node 18 to 19, we removed the classes `ConnectionTool` and `ConnectionHandle`, the latter having as a side effect the fact that `LocatorHandle` is now minimal–and thus, was not purged from the extent

map(Storable storable)}. The two classes of the extent are used to serialize/deserialize storable (graphical) objects. Their APIs preserve identity (i.e. multiple references to the same object), and they do that by recording/registering encountered objects, and that is the purpose of the map(Storable) method. Whether that is worth a separate concept, is another matter.

6) Node 24, with extent {DrawingChangeEvent, FigureChangeEvent}, and intent {Rectangle getInvalidatedRectangle()}. Interestingly, both DrawingChangeEvent and FigureChangeEvent inherit from java.util.EvenObject, but both have two constructors each, have two methods each, one is the intent(), and the other is getDrawing()/getFigure(), and both have fRectangle as a data member. Intrestingly, there are *also* similarities between the class CompositeFigure and the interface Drawing. If we were to somehow factor out the common aspects of CompositeFigure and Drawing[3], we could probably use the same change event object type for both.

7) Node 110, with extent {DrawApplet, DrawApplication}, and whose intent consists of 16 methods [4]. Note that both DrawApplet and DrawApplication implement the interface DrawingEditor, which has only methods {Drawing drawing(), DrawingView view(), Tool tool(), void toolDone(), void showStatus(String), void selectionChanged(DrawingView)}. Further, there is a third class that implements DrawingEditor, which is JavaDrawViewer, which implements only two extrra methods, beyond what is the interface. In fact, we do have a node, whose extent is {DrawApplet, DrawApplication, JavaDrawViewer, DrawingEditor}, which was categorized by our tool as FULL_EXTENT_FULL_BEHAVIOR_EXPLICIT_INTERFACE_IMPLEMENTATIONS. So, in this case, it seems that there were far more commonalities between DrawApplet and DrawApplication than was captured by DrawingEditor– ten (10) extra methods, to be precise! In fact, in terms of functionalities both DrawApplet and DrawApplication represent drawing applications, the main difference being that one is packaged as a Java applet, to be executed within a browser applet container, whereas the other can execute as a standalone Java desktop application. Thus, in this case, it would have made sense to create another 'intermediary abstraction', as shown in Figure **??**.

8) Node 128, with extent {JavaDrawApp, NetApp, NothingApp, PertApplication} and intent {void createTools(JToolBar palette), void main(String[] args)}. The classes are all descendants of DrawApplication, which implements void createTools(JToolBar palette). We would have had a case of FULL_EXTENT_FULL_BEHAVIOR_EXPLICIT_CLASS_SUBCL with DrawApplication part of the extent, if it weren't for the method void main(String[] args) which, in each case, creates an instance of the application and opens it. We cannot really say that this is an interesting feature. In fact, the use of method void main(String[] args) is itself questionable in this case[5].

9) Node 135, with extent {MDI_InternalFrame, DrawingChangeEvent}, and intent {Drawing getDrawing()}. The two classes are not 'conceptually' related, by any stretch of the imagination: MDI_InternalFrame represents an internal frame in a drawing application that has many 'internal windows', whereas DrawingChangeEvent represents events that can occur within drawings. Here, they were 'assembled' because both have accessors to the drawing at hand. Two heuristics could have eliminated this node, 1) the number of methods in the intent, or 2) the fact that the (single) method is an accessor. We have decided against using either filter because, 1) there are some interesting single method-behaviors (e.g., the java interface Comparable), and 2) getters may retrieve *computed* attributes, which can be domain meaningful. This is one case where either filter would have been helpful.

APPENDIX B
A CASE BY CASE ANALYSIS OF AD-HOC NODES FOR JREVERSEPRO

Let us now look at the ADHOC nodes:

1) *Node 0*: this node, with extent {jreversepro.common.JJvmOpcodes, jreversepro.common.KeyWords}, has an intent of *seventy nine methods*. A look at the types JJvmOpcodes and KeyWords shows that they are both *interfaces* consisting *solely of constants*[6], and they are 'implemented' by

3. We could define an interface called CompositeFigure, as a subtype of Figure, and define a separate class that implements CompositeFigure, and extends AbstractyFigure. Then, we could have Drawing extend the (now) *interface* Drawing.

4. The intent is {Drawing createDrawing(), Drawing drawing(), DrawingView view(), StandardDrawingView createDrawingView(), Tool createSelectionTool(), Tool tool(), ToolButton createToolButton(String iconName, String toolName, Tool tool), void endAnimation(), void initDrawing(), void paletteUserOver(PaletteButton button, boolean inside), void paletteUserSelected(PaletteButton button), void selectionChanged(DrawingView view), void setSelected(ToolButton button), void setTool(Tool t, String name), void startAnimation(), void toolDone()}

5. if it used to test the corresponding classes, there are better ways of doing it

6. JJvmOpcodes includes the hexadecimal value for all the JVM operation codes, whereas KeyWords lists the string constants of the various Java keywords

the classes that need those constants. It so happens in this case that four classes need those constants (`JDecompiler`, `JInstruction`, `IRunTimeFrame`, `JSwitchTable`). Accordingly, the methods of such classes (and their subclasses) appeared to 'have occurred twice', and hence the sizable intent (79 methods). This is obviously a false positive, and it should have been filtered in one of two ways:

- This is another case of 'multiple inheritance', whereby some class `C` implements two different interfaces, and thus appears within the two type hierarchies having the interfaces as their root, this misleading our algorithm into 'believing' that we have a case of two independent occurrences of the behavior. In this case, we have four such classes, and hence the 79 methods.
- As a general rule, we should have ignored interfaces that contain constants only. In this case, the `implements` relationship between a class and such interfaces is just a 'trick' to make the constant values visible/shared, but there is no behavior per se, to be shared.

2) *Node 1*: this node, with extent {`jreversepro.common.JJvmOpcodes`, `jreversepro.revengine.JBranchEntry`}, has an intent with two methods: {`String getOpr1()`, `int getTargetPc()`}. Interestingly, the method `String getOpr1()` is implemented by both `JBranchEntry` and *a class that implements* `JJvmOpcodes` (class `JInstruction`), and `int getTargetPc()`. Idem for `int getTargetPc()`, which is implemented by `JBranchEntry` and *another class that implements* `JJvmOpcodes` (class `JRunTimeFrame`). Notwithstanding the spuriousness of the candidate nodes induced by `JJvmOpcodes`, we have here a case where a 'feature' is not localized in a single class, but appears in two different classes. As far as this node is concerned, the feature does not appear to be interesting.
3) *Node 2, 3, 4, 5, 6, and 7*: Similar to node 0, all of these nodes have extents consisting of a set of *constant-only* interfaces[7], which are implemented by some of the same classes, leading to rather large intents. These are all false-positive which could/have been filtered, as explained for node 0.
4) *Node 8*: not very interesting. The extent is {`jreversepro.reflect.JException`, `jreversepro.revengine.JBranchEntry`}, with intent the single method `int getStartPc()`, which returns the beginning position ('program counter') of the construct. It is somewhat surprising that the concept of 'start program counter' for block-type constructs appears only for `Exception` (for the various catch blocks) or `jreversepro.revengine.JBranchEntry`, which handles `while` and `if` block statements. Note that the `JCaseEntry` class, which represents the 'processing blocks' of cases, has a slightly different concept, called 'target' (for target program counter), which plays the role of start program counter for the case block. Indeed, in a `switch` statement, we can have several cases 'cascading' to the same processing block, as shown below:

```
case 12:
case 13:
case 18:
<<do Something>> // target refers to this posit
```

and hence, the concept of 'target' [program counter] may be more appropriate than 'start' [program counter].
5) *Node 10*, with extent the two interfaces {`jreversepro.common.JJvmOpcodes`, `jreversepro.revengine.JReverseEngineer`}, and whose intent consists of the union of the APIs of the two classes that implement the two interfaces, namely `JDecompiler` and `JDisassembler`. This is another false positive caused by a combination of 'multiple inheritance' and constant-only interfaces (`JJvmOpCodes`).
6) *Node 11*, with extent {`jreversepro.JAwtFrame`, `jreversepro.JCmdMain`, `jreversepro.JMainFrame`}, and intent {`void main(String[] aArgs)`}. The classes of the extent correspond to the three different ways that the program can be invoked: a) as a command line, b) through an AWT gui (main panel a subclass of `java.awt.Frame`), and c) through a SWING gui (main panel a subclass of `java.swing.JFrame`). The three 'main' classes share the `void main(String[] aArgs)` method.
7) *Node 12*, this node, a child of node 11, has an extent consisting of the GUI main classes, {`jreversepro.JAwtFrame`, `jreversepro.JMainFrame`}, and thus has a much bigger intent consisting of {`void addListeners()`, `void appClose()`, `void copyText()`, `void cutText()`, `void formatTitle(String aFileName)`, `void initAppState()`, `void main(String[] aArgs)`, `void openFile()`, `void reverseEngineer(File aFile)`, `void saveFile()`, `void saveProperties()`, `void showAbout()`, `void showFontDialog()`, `void viewPool()`}. Clearly, there is 'an abstraction' here that was missed by the designers. They could have created an interface that represents a 'graphical invocation application', with two implementations, one for AWT-style applications, and one for Swing-style applications.
8) *Node 13*, with extent the two interfaces {`jreversepro.common.AppConstants`, `jreversepro.common.KeyWords`}, and

7. different subsets of {`jreversepro.common.JJvmOpcodes`, `jreversepro.common.KeyWords`, `jreversepro.revengine.BranchConstants`, `jreversepro.runtime.OperandConstants`}

whose intent consists of the *union* of the APIs of the two classes that implement both interfaces, namely `jreversepro.JAwtFrame` and `jreversepro.JMainFrame`. This is another false positive due to a combination of 'multiple inheritance' and constant-only interfaces.

9) *Node 14*, with extent {`jreversepro.revengine.JBranchTable`, `jreversepro.revengine.JCollatingTable`}, and intent the single method `void sort()`. We could look for semantic similarities between the classes of the extent (and there are), but the intent is due simply to the fact that both contain collections of 'branches' [8] that can be sorted.
10) *Node 15*, with extent {`jreversepro.awtui.JCustomListPanel`, `jreversepro.awtui.JErrorDlg`, `jreversepro.gui.JCustomListPanel`, `jreversepro.gui.JErrorDlg`} and intent {`void addComponents()`}, captures the fact that these classes are aggregate graphical objects.
11) *Nodes 16, 17, 18, 21, 24, and 25* have extents consisting of two graphical classes each, one inheriting from an AWT class, and the other inheriting from the corresponding SWING class, but essentially implementing the same functionality, as was the case for node 12. In fact, in all 3 cases, the classes have the same name, but belong to different packages. The SWINg version often takes advantage of the more advanced functionalities (for example, scrollable text areas and such), but they otherwise implement the same API. That common API could have been captured by Java interfaces, albeit small ones (between 1 and 4 methods)[9].
12) *Node 19*, with extent {`jreversepro.reflect.JConstantPoolEntry`, `jreversepro.reflect.JField`, `jreversepro.runtime.Operand`}, and intent `String getValue()`. The three classes represent 'things' that have a data type, and a value. This method returns the string representation of that value. The type, for the case of `jreversepro.reflect.JConstantPoolEntry`, is represented in a different way from the other two because constant pool entries do not only refer to data objects, but they can also refer to methods or classes. This is a mildly interesting concept, but it was caught, nonetheless.
13) *Node 20*, which is a child of node 19, does capture the data type commonality, by excluding `jreversepro.reflect.JConstantPoolEntry` from the extent. Indeed, it has the extent {`jreversepro.reflect.JField`, `jreversepro.runtime.Operand`} and intent {`String getDatatype()`, `String getValue()`}.
14) *Node 22*, with an extent {`jreversepro.revengine.JBranchComparator`, `jreversepro.revengine.JCaseComparator`}, and intent {`int compare(Object o1, Object o2)`}, simply rediscovered the `java.util.Comparator` interface, which is implemented *explicitly* by the two classes of the extent. However, because `java.util.Comparator` was not defined in the same project, it was excluded from our analysis[10].
15) *Node 26*, with extent {`jreversepro.reflect.JMember`, `jreversepro.runtime.JLocalEntry`}, and intent {`String getName()`} discovered things that have a name, i.e. a class member (field or method) or an entry in the local symbol table of a method (local variable or label). Mildly interesting concept.

8. A `java.util.List` for one, and an array for the other
9. The duplicated classes are found in packages `jreversepro.gui.*` and `jreversepro.awtui.*`
10. Actually, the `Comparator` interface includes also the `boolean equals(Object obj)` method. However, because that method is defined outside of the project, it was also ignored