The CLOSER: Automating Resource Management in Java

Isil Dillig  Thomas Dillig
Computer Science Department
Stanford University

Eran Yahav  Satish Chandra
IBM T.J. Watson Research Center

ISMM 2008
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
  - Operating system resources: Files, sockets, ...
Motivation

Operating System Resources

```java
public void transferData()
{
    Socket s = new Socket();
    s.connect(...);
    ...
    s.close();
}
```
Motivation

Operating System Resources

```java
public void transferData()
{
    Socket s = new Socket();
    s.connect(...);
    ...
    s.close();
}
```
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
  - Operating system resources: Files, sockets, ...
  - Window system resources: Fonts, colors, ...
## Motivation

### Window System Resources

```java
public void draw()
{
    Font f = new Font();
    ...
    f.dispose();
}
```
Motivation

Window System Resources

```java
public void draw()
{
    Font f = new Font();
    ...
    f.dispose();
}
```
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
  - Operating system resources: Files, sockets, ...
  - Window system resources: Fonts, colors, ...
  - Application specific resources: Listeners, model view control pattern, ...
Motivation

Application Specific Resources

```java
public class SomeView {
    private SomeListener l;
    private WorkbenchWindow w;

    public void createPartControl(Composite parent) {
        l = new Listener(this);
        w.addPerspectiveListener(l);
    }

    public void dispose(){
        w.removePerspectiveListener(l);
    }
}
```
Motivation

Application Specific Resources

```java
public class SomeView {
    private SomeListener l;
    private WorkbenchWindow w;

    public void createPartControl(Composite parent) {
        l = new Listener(this);
        w.addPerspectiveListener(l);
    }

    public void dispose()
    {
        w.removePerspectiveListener(l);
    }
}
```
Generalized Definition of Resource

A resource \( r \) is an instance of any type \( C \) whose specification has the following requirement:

If a method \( m \) is called with \( r \) as the receiver or parameter, then a matching method \( m' \) must be called after the last use of \( r \). We call \( m \) the obligating method and \( m' \) the fulfilling method.
Generalized Definition of Resource

Definition of a Resource

A resource $r$ is an instance of any type $C$ whose specification has the following requirement:

- If a method $m$ is called with $r$ as the receiver or parameter
Generalized Definition of Resource

Definition of a Resource

A resource \( r \) is an instance of any type \( C \) whose specification has the following requirement:

- If a method \( m \) is called with \( r \) as the receiver or parameter
- Then a matching method \( m' \) must be called after the last use of \( r \).
A resource $r$ is an instance of any type $C$ whose specification has the following requirement:

- If a method $m$ is called with $r$ as the receiver or parameter
- Then a matching method $m'$ must be called after the last use of $r$.

We call $m$ the **obligating** method and $m'$ the **fulfilling** method.
Existing Approaches and Their Drawbacks

- Manual Resource Management
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...
Existing Approaches and Their Drawbacks

- [Viewers] LabelProvider disposed twice
- [Actions] StepIntoSelectionActionDelegate "leaks" editor ...
- Leakage: system menu not disposed for Tab...
- [ViewMgmt] Saveable parts are leaked on perspective close.
- Widget Disposed Exception when importing breakpoint
- [Contributions] Leakage: PluginActionContributionItem not...
- Progress view leaks X resources on Linux Ubuntu
- Memory Leak in ActionSetManager
- ShowViewMenu leaks 4 images for each show
- [Contributions] NPE in PopupMenuExtender.dispose
- New Presentation leaks colors

328 bugs found.
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- Finalization
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- Finalization
  - In current JVM implementations, program might run out of non-memory resources before finalizers are called
  - Asynchronous with respect to last use point
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- Finalization
  - In current JVM implementations, program might run out of non-memory resources before finalizers are called
  - Asynchronous with respect to last use point

- Language Based Solutions
Existing Approaches and Their Drawbacks

- **Manual Resource Management**
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- **Finalization**
  - In current JVM implementations, program might run out of non-memory resources before finalizers are called
  - Asynchronous with respect to last use point

- **Language Based Solutions**
  - e.g., weak references: works where premature disposal is not detrimental, but not a general solution
Goals of Our Approach

- Impose minimal burden on the programmer.
Goals of Our Approach

- Impose minimal burden on the programmer.

- Should not be limited to a fixed-class of resources.
Goals of Our Approach

- Impose minimal burden on the programmer.
- Should not be limited to a fixed-class of resources.
- Should not restrict programming patterns.
Challenges

- Resources may be shared.
Challenges

A Font object is shared between two Window objects:
Challenges

- Resources may be shared.

  - Consequence: It is not always possible to determine the correct dispose point of the resource purely statically.
Challenges

- Resources may be shared.
  - Consequence: It is not always possible to determine the correct dispose point of the resource purely statically.

- Traditional notion of reachability is not adequate for reasoning about resource lifetimes.
The reference from the Observed object to the Listener is a **non-interest link**:

![Diagram showing the relationship between Observer, Listener, and Observed]

- Observer
- Listener
- Observed

The CLOSER: Automating Resource Management in Java
Challenges

- Resources may be shared.
  - Consequence:
    It is not always possible to determine the correct dispose point of the resource purely statically.

- Traditional notion of reachability is not adequate for reasoning about resource lifetimes.
  - Consequence:
    Refined notion of reachability = Interest Reachability
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
Overview of Our Approach

class WorkbenchWindow {

    private Listener l;

    @Obligation(obligates = 'removePerspectiveListener', resource=1)
    public void addPerspectiveListener(Listener l);

    ...

}
Overview of Our Approach

class WorkbenchWindow {
    private Listener l;

    @Obligation(obligates = ‘removePerspectiveListener’, 
                resource=1)
    public void addPerspectiveListener(Listener l);

    ...
}

Overview of Our Approach

class WorkbenchWindow {

    private Listener l;

    @Obligation(obligates = ‘removePerspectiveListener’,
                resource=1)
    public void addPerspectiveListener(Listener l);

    ...

}
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links
Overview of Our Approach

class WorkbenchWindow {

    @NonInterest
    private Listener l;

    @Obligation(obligates = ‘‘removePerspectiveListener’’,
                resource=1)
    public void addPerspectiveListener(Listener l);

    ...

}
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
  - and later automatically synthesizes dispose methods.
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
  - and later automatically synthesizes dispose methods.

- CLOSER statically analyzes resource lifetimes to identify how and where each resource should be disposed.
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
  - and later automatically synthesizes dispose methods.

- CLOSER statically analyzes resource lifetimes to identify how and where each resource should be disposed.

- CLOSER automatically inserts any appropriate resource dispose calls into source code.
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the resource interest graph (RIG).
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the resource interest graph (RIG).

**Resource Interest Graph**

An RIG for a method $m$ at a given point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where:

- $V$ is a finite set of abstract memory locations
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the **resource interest graph (RIG)**.

An RIG for a method $m$ at a given point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where:

- $V$ is a finite set of abstract memory locations
- $E$ is a set of directed edges between these locations
Resource Interest Graph

To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the resource interest graph (RIG).

An RIG for a method $m$ at a given point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where:

- $V$ is a finite set of abstract memory locations
- $E$ is a set of directed edges between these locations
- $\sigma_V$ is a mapping from abstract memory locations to a value in 3-valued logic, identifying whether that location may, must, or must-not be a resource
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the **resource interest graph (RIG)**.

**Resource Interest Graph**

An RIG for a method \( m \) at a given point is a tuple \( \langle V, E, \sigma_V, \sigma_E \rangle \) where:

- \( V \) is a finite set of abstract memory locations
- \( E \) is a set of directed edges between these locations
- \( \sigma_V \) is a mapping from abstract memory locations to a value in 3-valued logic, identifying whether that location may, must, or must-not be a resource
- \( \sigma_E \) is a mapping from edges to a boolean value identifying whether that edge is an interest or non-interest edge
Example RIG

```java
public class BufferPrinter {
    
    public BufferPrinter(Buffer buf) {
        this.buf = buf;
        this.listener =
            new BufferListener(this);
        buf.addListener(listener);
        this.socket = new Socket();
        socket.connect();
    }
}
```
public class BufferPrinter {
    ... 
    public BufferPrinter(Buffer buf) {
        this.buf = buf;
        this.listener =
            new BufferListener(this);
        buf.addListener(listener);
        this.socket = new Socket();
        socket.connect();
    }
}

\[ \sigma_E(e) = 1 \]
\[ \sigma_E(e) = 0 \]
\[ \sigma_V(A) = ? \]
\[ \sigma_V(B) = 1 \]
\[ \sigma_V(C) = 1 \]
\[ \sigma_V(D) = ? \]
\[ \sigma_V(e) = 1 \]
\[ \sigma_V(e) = 0 \]

Example RIG
A class $T$ is a higher-level resource if:

There exists a field $l$ of some instance of $T$ such that:

$$\sigma_{V}(l_{f}) \subseteq 1 \sigma_{E}(l_{T} \times f \rightarrow l_{f}) = true$$

If $T$ is inferred to be a higher-level resource, $T$'s constructor becomes an obligating method and the dispose method synthesized by CLOSER becomes the corresponding fulfilling method.
A class $\mathcal{T}$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $\mathcal{T}$
A class $\mathcal{T}$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $\mathcal{T}$
- such that $\sigma_V(l_f) \sqsupseteq 1$
Higher-Level Resource

A class $T$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $T$
- such that $\sigma_V(l_f) \supseteq 1$
- $\sigma_E(l_T \times f \rightarrow l_f) = \text{true}$
A class \( \mathcal{T} \) is a higher-level resource if:

- there exists a field \( l_f \) of some instance of \( \mathcal{T} \)
- such that \( \sigma_V(l_f) \supseteq 1 \)
- \( \sigma_E(l_T \times f \rightarrow l_f) = \text{true} \)

If \( \mathcal{T} \) is inferred to be a higher-level resource,
A class $T$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $T$
- such that $\sigma_V(l_f) \supseteq 1$
- $\sigma_E(l_T \times f \to l_f) = \text{true}$

If $T$ is inferred to be a higher-level resource,

- $T$’s constructor becomes an obligating method
A class $T$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $T$
- such that $\sigma_V(l_f) \supseteq 1$
- $\sigma_E(l_T \times f \to l_f) = \text{true}$

If $T$ is inferred to be a higher-level resource,

- $T$’s constructor becomes an obligating method
- and the `dispose` method synthesized by CLOSER becomes the corresponding fulfilling method.
Higher-Level Resource Example

\[
\sigma_E(e) = 1 \\
\sigma_E(e) = 0
\]

\[
\sigma_v(B) = 1 \\
\sigma_v(C) = 1 \\
\sigma_v(D) = 0
\]
Higher-Level Resource Example

\[ \sigma_v(B) = 1 \]
\[ \sigma_v(C) = 1 \]
\[ \sigma_v(D) = 0 \]
CLOSER disposes of a resource in one of three ways:

1. **Strong static dispose**: Dispose resource directly by calling fulfilling method. No checks necessary.
2. **Weak (conditional) static dispose**: Checks whether the resource's obligating method was called before disposing it.
3. **Dynamic dispose**: Requires keeping a run-time “interest-count” needed whenever CLOSER infers that resource may be shared.
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
CLOSER disposes of a resource in one of three ways:

- Strong static dispose
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- Weak (conditional) static dispose
  - Checks whether the resource’s obligating method was called before disposing it.

- Dynamic dispose
  - Requires keeping a run-time “interest-count” needed whenever CLOSER infers that resource may be shared.
CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource’s obligating method was called before disposing it.
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource’s obligating method was called before disposing it.

- **Dynamic dispose**
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource’s obligating method was called before disposing it.

- **Dynamic dispose**
  - Requires keeping a run-time “interest-count”
  - Needed whenever CLOSER infers that resource may be shared.
Solicitors

CLOSER proves a resource is unshared if it can identify a unique solicitor for it.
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique solicitor for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$. 
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique **solicitor** for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$. 
CLOSER proves a resource is unshared if it can identify a unique solicitor for it.

If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$. 
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique **solicitor** for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$.
- CLOSER infers a solicitor by:
  - First computing a set of solicitor candidates from the resource interest graph for each point in the program.
  - Then by doing data flow analysis to ensure that the inferred solicitor candidates “agree” at every program point.
CLOSER proves a resource is unshared if it can identify a unique **solicitor** for it.

- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$.

CLOSER infers a solicitor by:

- First computing a set of **solicitor candidates** from the resource interest graph for each point in the program.
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique solicitor for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$.
- CLOSER infers a solicitor by:
  - First computing a set of solicitor candidates from the resource interest graph for each point in the program
  - Then by doing data flow analysis to ensure that the inferred solicitor candidates “agree” at every program point.
Inference of Solicitors

To compute a solicitor candidate for resource \( r \):
Inference of Solicitors

To compute a solicitor candidate for resource $r$:

- CLOSER first computes a set of paths $\mathcal{P} = \langle l, f_1 \circ \ldots \circ f_n, \text{May}/\text{Must} \rangle$ that reach $r$. 

Inference of Solicitors

To compute a solicitor candidate for resource $r$:

- CLOSER first computes a set of paths
  \[ P = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle \] that reach $r$

- It then applies a set of unification rules to determine the existence of a canonical path $l.f_1\ldots f_n$ that may safely be used to dispose $r$
Inference of Solicitors

To compute a solicitor candidate for resource \( r \):

- CLOSER first computes a set of paths
  \[
P = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle
\]
  that reach \( r \)

- It then applies a set of unification rules to determine the existence of a canonical path \( l.f_1\ldots f_n \) that may safely be used to dispose \( r \)

- If such a unique path exists, then \( l.f_1\ldots f_n \) is designated as a solicitor candidate for \( r \)
Inference of Solicitors

To compute a solicitor candidate for resource \( r \):

- CLOSER first computes a set of paths
  \( \mathcal{P} = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle \) that reach \( r \)

- It then applies a set of unification rules to determine the existence of a canonical path \( l.f_1 \ldots f_n \) that may safely be used to dispose \( r \)

- If such a unique path exists, then \( l.f_1 \ldots f_n \) is designated as a solicitor candidate for \( r \)

- If the inferred solicitor candidates for \( r \) are consistent, then \( r \) is disposed through the cascading series of dispose calls initiated by \( l\text{.dispose()} \), invoked after the last use point of \( l \)
Solicitor Example

Inferred solicitor for R:

- toolBar
- button.dispose()
- image.dispose()
Solicitor Example

Inferred solicitor for R:

```
toolBar.button
```
Inferred solicitor for R:

```
toolBar.button
```

Image disposed via call chain:
Solicitor Example

- Inferred solicitor for R:
  ```java
  toolBar.button
  ```

- Image disposed via call chain:
  ```java
  toolBar.dispose()
  ```
Solicitor Example

▷ Inferred solicitor for R:

```
toolBar.button
```

▷ Image disposed via call chain:

```
toolBar.dispose()
  ↓
button.dispose()
```
Solicitor Example

▷ Inferred solicitor for R:
  ```java
toolBar.button
  ```

▷ Image disposed via call chain:
  ```java
toolBar.dispose()
  ↓
button.dispose()
  ↓
image.dispose()
  ```
Implementation

- Static Analysis:
  - Builds on IBM WALA framework for analysis of Java byte code
Implementation

- Static Analysis:
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit
Implementation

- **Static Analysis:**
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- **Dynamic Instrumentation:**
  - Does not rely on modifying the JVM
Implementation

- **Static Analysis:**
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- **Dynamic Instrumentation:**
  - Does not rely on modifying the JVM
  - A `Manager` class keeps dynamic interest counts
Implementation

- **Static Analysis:**
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- **Dynamic Instrumentation:**
  - Does not rely on modifying the JVM
  - A `Manager` class keeps dynamic interest counts
  - The modified source code calls static methods of the `Manager`
Implementation

- Static Analysis:
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- Dynamic Instrumentation:
  - Does not rely on modifying the JVM
  - A Manager class keeps dynamic interest counts
  - The modified source code calls static methods of the Manager

- CLOSER appears transparent to the programmer
  - The programmer can inspect and change the code instrumented by CLOSER
We applied CLOSER to automate resource management of an SWT Showcase Graphics Application.
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
- Uses 67 different resources
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
- Uses 67 different resources
- Reasonably complex resource management logic
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
- Uses 67 different resources
- Reasonably complex resource management logic
- Manually removed all resource management code
### Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>
### Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- User annotates only 5 resources.
- CLOSER infers all the remaining 62 resources.
## Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>
Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- Missing dispose call in the original code was a resource leak.
- Programmer forgot to dispose a Transpose (resource in SWT).
More weak dispose calls because CLOSER is path-insensitive.

Inserts redundant null-checks even though one already exists.
private void paint() {
    if(image == null) {
        if(image!=null){
            image.dispose();
        }
        image = new Image(...);
    }
}
**Case Study, Continued**

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- No shared resources in the application.
- CLOSER successfully identified all resources as unshared.
Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- CLOSER doesn’t cause code bloat or substantial runtime overhead.
- And it is correct by construction.
Related Work

**DeLine, R., and Fahndrich, M.**
Enforcing high-level protocols in low-level software.

**Guyer, S., McKinley, K., and Frampton, D.**
Free-Me: a static analysis for automatic individual object reclamation.

**Heine, D. L., and Lam, M. S.**
A practical flow-sensitive and context-sensitive c and c++ memory leak detector.

**Blanchet, B.**
Escape analysis for object oriented languages. application to Java™.
In *OOPSLA* (Denver, 1998).

**Boehm, H.**
Destructors, finalizers, and synchronization.