Design Intent:
a Principled Approach to Application Security

Jonathan Aldrich
Associate Professor
School of Computer Science and CyLab
Carnegie Mellon University

CyLab Seminar
September 14, 2009

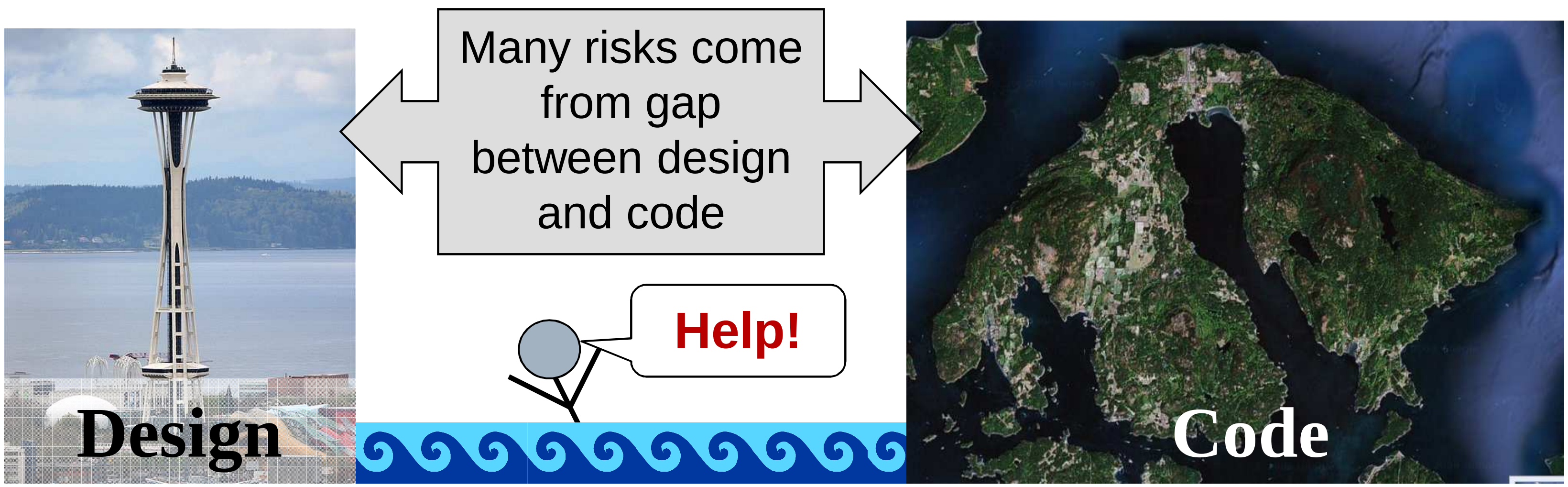
The Design-Code Security Gap

Design-Level Security

- Proactive
  - Security lifecycle
  - Threat modeling
  - Design evaluations
- Risks
  - Lack of detail
  - Missed interactions
  - Out of date

Code-Level Security

- Reactive
  - Security testing
  - Inspection
  - Code scanning tools
- Risks
  - Low-level abstractions
  - Tremendous complexity
  - Incompatible assumptions
  - Lack of context



Example: Concurrency

- Concurrency is increasingly common
  - Multi-core: must parallelize for performance
  - Many domains inherently concurrent
    - control systems, web, …
- Source of new vulnerabilities
  - Time of Check – Time of Use (TOCTOU) errors
- Example: java.util.logging defect
  - Found by CMU's Fluid research team, led by Bill Scherlis

Fluid tool: [Aaron Greenhouse and William L. Scherlis. Assuring and Evolving Concurrent Programs: Annotations and Policy. ICSE 2002,]

Carnegie Mellon CyLab CONFIDENCE FOR A NETWORKED WORLD

Concurrency: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
public class Logger { ...
  private Filter filter;

  public void setFilter(Filter newFilter) ... {
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

}
```

Consider `setFilter()` in isolation

Concurrency: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
public class Logger { ...
  private Filter filter;









  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

Consider **`log()`** in isolation

Concurrency: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
/** ... All methods on Logger are multi-thread safe. */
public class Logger { ...
  private Filter filter;

  /** ...
   * @param newFilter a filter object (may be null)
   */
  public void setFilter(Filter newFilter) ... {
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

Consider class **Logger** in it's entirety!

Concurrency: java.util.logging.Logger

[Source: Aaron Greenhouse]

```java
/** ... All methods on Logger are multi-thread safe. */
public class Logger { ...
  private Filter filter;

  /** ...
   * @param newFilter a filter object (may be null)
   */
  public void setFilter(Filter newFilter)…{
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

1

2

3

Class Logger has a ***race condition***.

Concurrency: java.util.logging.Logger

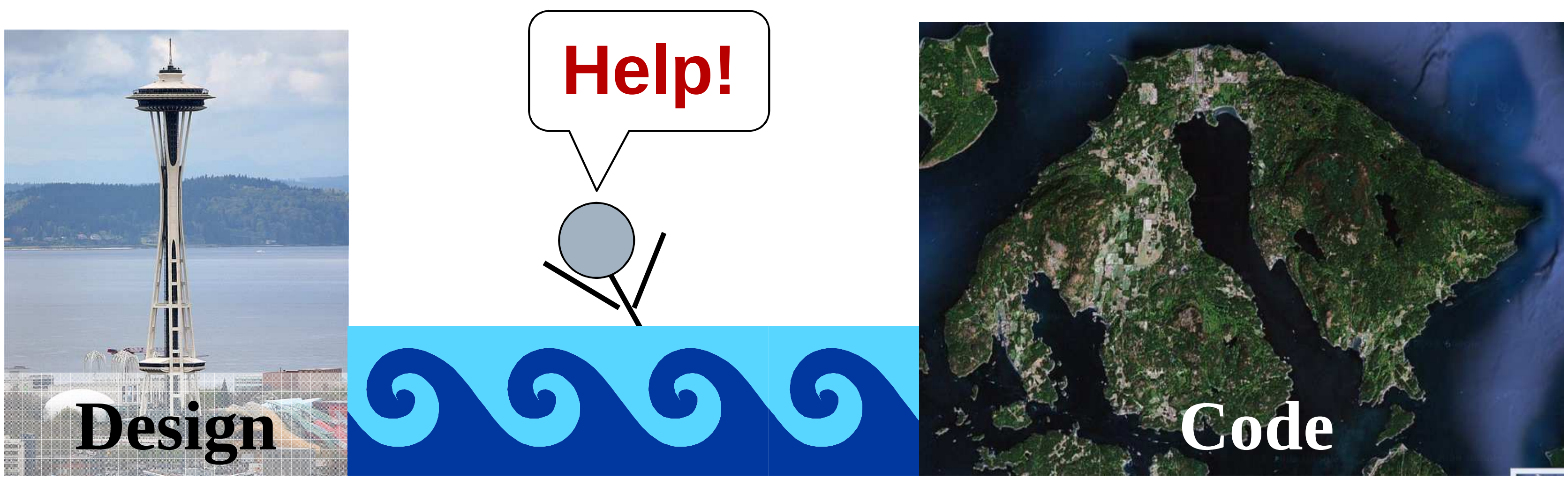
[Source: Aaron Greenhouse]

```
/** ... All methods on Logger are multi-thread safe. */
public class Logger { ...
  private Filter filter;

  /** ...
   * @param newFilter a filter object (may be null)
   */
  public synchronized void setFilter(Filter newFilter)…{
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

Correction: synchronize `setFilter()`

Concurrency Defects are Devilish

- Hard to test for
  - Occur on rare paths
  - Nondeterministic, irreproducible
- Hard to inspect for
  - Non-local inconsistencies
  - Requires model of design



Design Intent: Bridging Design and Code

Design Intent:

Automatically checkable engineering information that describes *how* code meets its quality requirements

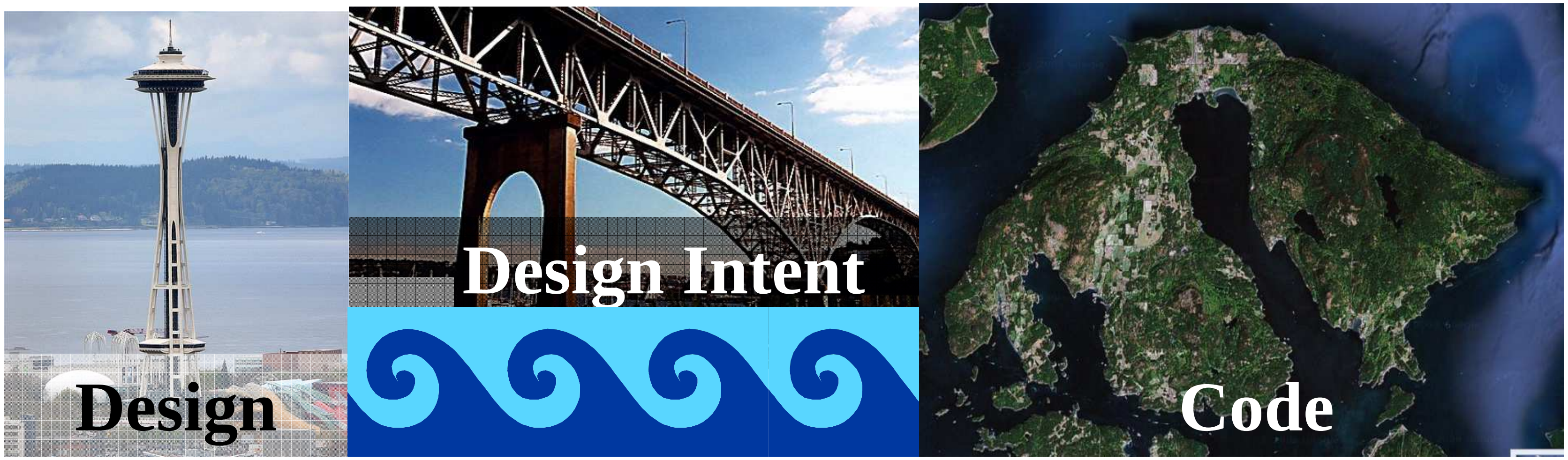
Validates Design against Code

- Keeps design up to date
- Ensures details are not omitted

Ensures Code follows Design

- Avoids inconsistent assumptions
- Raises level of abstraction
- Manages complexity

Bottom line: Assurance that secure design is realized in secure code



Concurrency Design Intent

[Source: Aaron Greenhouse]

```
public void setFilter(Filter newFilter)…{
  if (!anonymous) manager.checkAccess();
  filter = newFilter;
}
```

- How do we know this is a defect?
  - Logger is intended to be used concurrently
  - The Logger instance's lock protects filter
  - The lock may not be currently held
- These are all about design intent!
- How can a tool help the engineer:
  - Discover design intent is needed?
  - Check the design intent against the code?
  - Find the defect?

Concurrency Design Intent

[Source: Aaron Greenhouse]

- Discover design intent is needed
  - Tool observes synchronized statement, asks what state the lock protects

```
public void log(LogRecord record) { ...
  synchronized (this) {
    if (filter != null
        && !filter.isLoggable(record)) return;
  } ...
}
```

- User adds design intent:

```
// @lock FL is this protects filter
public class Logger {
   ...
}
```

Concurrency Design Intent

[Source: Aaron Greenhouse]

- Now the intent is in the source
  - The tool can check consistency between intent and code
  - Tool observes an access to filter
  - filter is protected by lock FL
  - FL is not known to be held
  - Warning: unprotected field access; possible race condition detected

```
// @lock FL is this protects filter
public class Logger {
  public void setFilter(Filter newFilter)…{
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }
```

- User can now fix the defect
  - Tool responds that all field accesses are now protected

Concurrency Design Intent

[Source: Aaron Greenhouse]

- Look again: is this always a defect?
- What if the caller is supposed to lock before calling setFilter?
  - That's design intent!

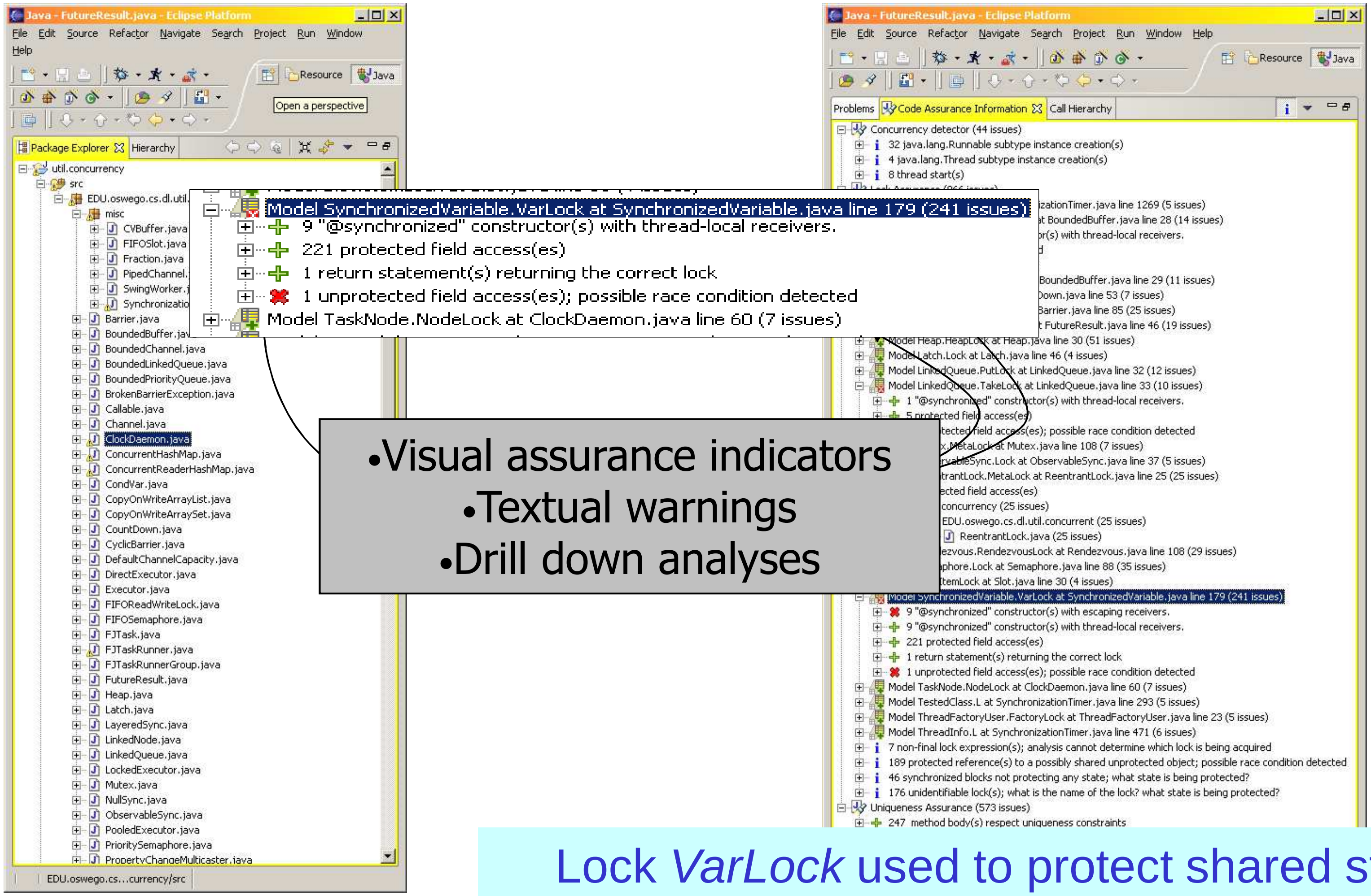
  - Instead of synchronizing, the user can add more design intent:

```
// @lock FL is this protects filter
public class Logger {
  // @requiresLock FL
  public void setFilter(Filter newFilter)…{
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }
```

- Now the tool reports that intent and code are consistent
  - But it warns if setFilter is called without acquiring the lock!

Carnegie Mellon CyLab CONFIDENCE FOR A NETWORKED WORLD

Case Study: java.util.concurrent



Lock *VarLock* used to protect shared state

- 221 protected accesses
- 1 unprotected access

Commercial Case Studies with Fluid

[Source: William Scherlis]

Top-10 Software Vendor

- 300 kloc Java
  - Production code
- 3 days of modeling
  - Both by CMU and vendor
- 25 faults detected
  - Deadlocks and races
  - Fixes checked into codebase

Major Aerospace Vendor

- 250 kloc Java
  - 4 production systems
  - Some already deployed
- Races found in all 4
  - Fixes made to 3 of 4

Developer/Manager Quotes:

It would have been very difficult if not impossible to find these issues without the Fluid analysis

I'm actually considering implementing a policy that you can't add a synchronize to the code without documenting [in Fluid notation] what region it applies to.

Commercial Case Studies with Fluid

[Source: William Scherlis]

Top-10 Software Vendor | **Major Aerospace Vendor**

- 300 kloc J
  - Produc
- 3 days of
  - Both b
- 25 faults c
  - Deadlo
  - Fixes checked into codebase

ava

ction systems

lready deployed

nd in all 4

ade to 3 of 4

Now available commercially as SureLogic's JSure tool

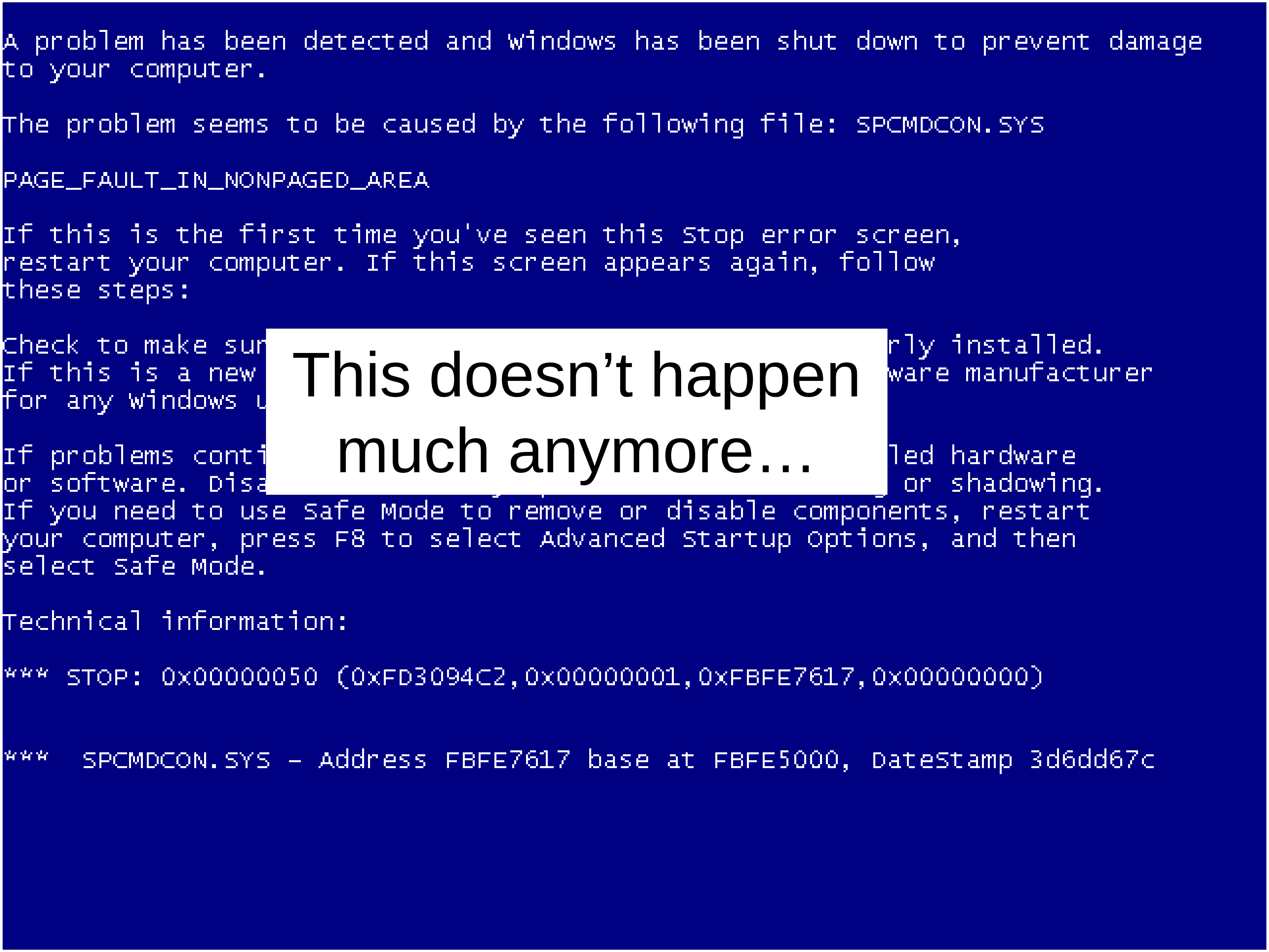
Developer/Manager Quotes:

It would have been very difficult if not impossible to find these issues without the Fluid analysis

I'm actually considering implementing a policy that you can't add a synchronize to the code without documenting [in Fluid notation] what region it applies to.

Another Design Intent Success Story

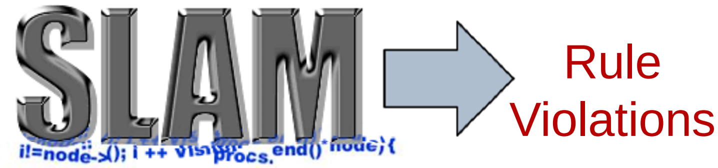
- Not so long ago, Windows was not so reliable…

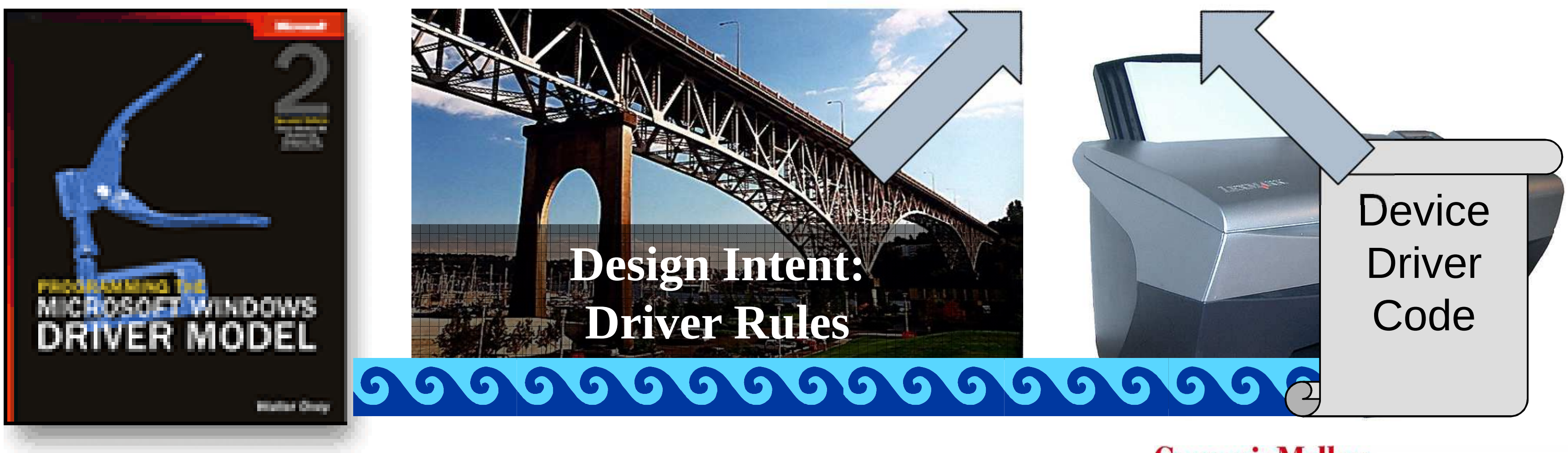


This doesn’t happen much anymore…

Device Driver Design Intent

- Most blue screens came from buggy device drivers
  - Device driver model very complex
  - Gap between model and code
- Design Intent
  - Machine-checkable summary of device driver rules
- SLAM tool
  - Reports inconsistencies





Factors in SLAM’s Success

- Deep math, targeted technology
  - Boolean abstraction, model checking, theorem proving, abstraction refinement
  - Analysis specialized to the problem

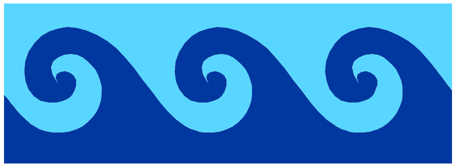
- Design intent
  - Device driver rules

- Tool tuned for scalability, usability
  - SLAM research prototype ➔ Static Driver Verifier
  - Analyzes code separately
  - Shows actionable error trace

- Business case
  - Low cost: only Microsoft writes design intent
  - High benefit: driver certification
    ➔ inclusion in Windows build

Design Intent Today: Languages and Libraries

- C: char*
  - Intent: here's a bunch of memory
  - Append: beware of strcpy…



- Java: char[]
  - Intent: character array with a specific (checked) length
  - Append: ArrayCopy may fail, but does so securely

- Library: String
  - Intent: a string, backed with automatically managed memory.
  - Append: no buffer overruns
  - SQL queries: pasting Strings risks command injection

- Library: PreparedStatement
  - This is a SQL query, ready to accept parameters
  - Resistant to command injection attacks

Design Intent Today: Other Tools

- Microsoft analysis tools
  - Design intent: how big is that buffer?

    void writeToBuf(**__out_ecount(BUF_SIZE)** **char** *buffer)

  - Expressed in Standard Annotation Language (SAL)
  - Checked by PREfast – free download from Microsoft

- FindBugs – open source Java tool
  - Design intent: could that value be null?

```
interface Map<K,V> {
    public @CheckForNull V get(@NonNull K key);
```

Design Intent: More Effective Tools

- Findbugs study
  - Added design intent regarding null
  - Small increase in false warning rate
    - From near zero to 10-20%
  - Major increase in % real null pointer bugs found
    - From 0-30% to 50-80%

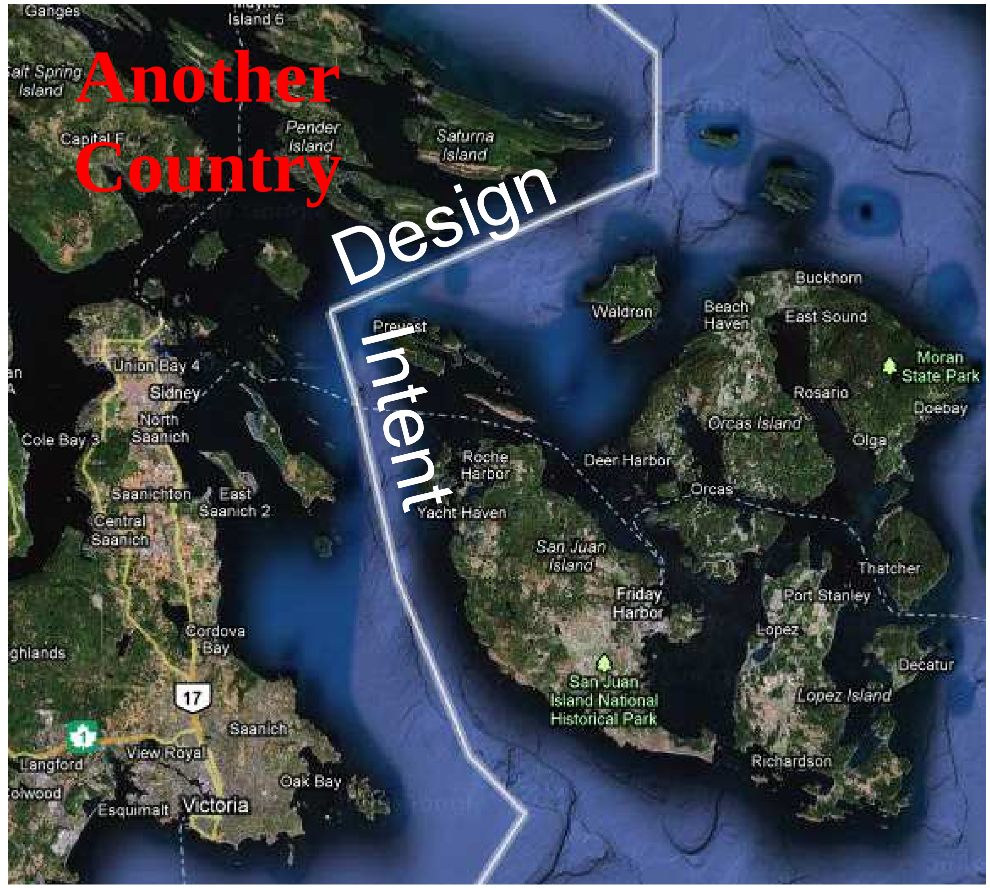
[David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. PASTE 2005.]

Design Intent: More Effective QA

- Testing
  - Shows boundaries of normal behavior
    - Aids in constructing appropriate test suites
    - Facilitates robustness testing
- Inspection
  - Provides context so developers can check consistency of assumptions
    - Who locks the lock? Could this be null? Design intent documents.

Design Intent and the Supply Chain

- Supply chain
  - Multiple sources
  - Differential trust
  - New vulnerabilities
- Design intent role
  - Captures richer interface
  - Example: who acquires the lock?
- Benefits
  - Avoids inconsistent assumptions
  - Direct assurance of supplied components
  - Compositional analysis



Putting Analysis Tools into Practice

- Adopt incrementally
  - Start with early adopters, small teams
  - Use as champions in rest of organization
- Prioritize tools
  - Focus on high value defects, with low false warning rates
    - context dependent:
      unused variable assignments a major bottleneck at eBay
  - Turn off less useful checkers
- Ensure developer buy-in, then build into process
  - Example: check-in gates, QA handoff standards
  - Ensure there's a way to override enforcement
- Support the tool

[Sources: Manuvir Das (Microsoft) and Ciera Jaspan (eBay)]

Design Intent: Emerging Research

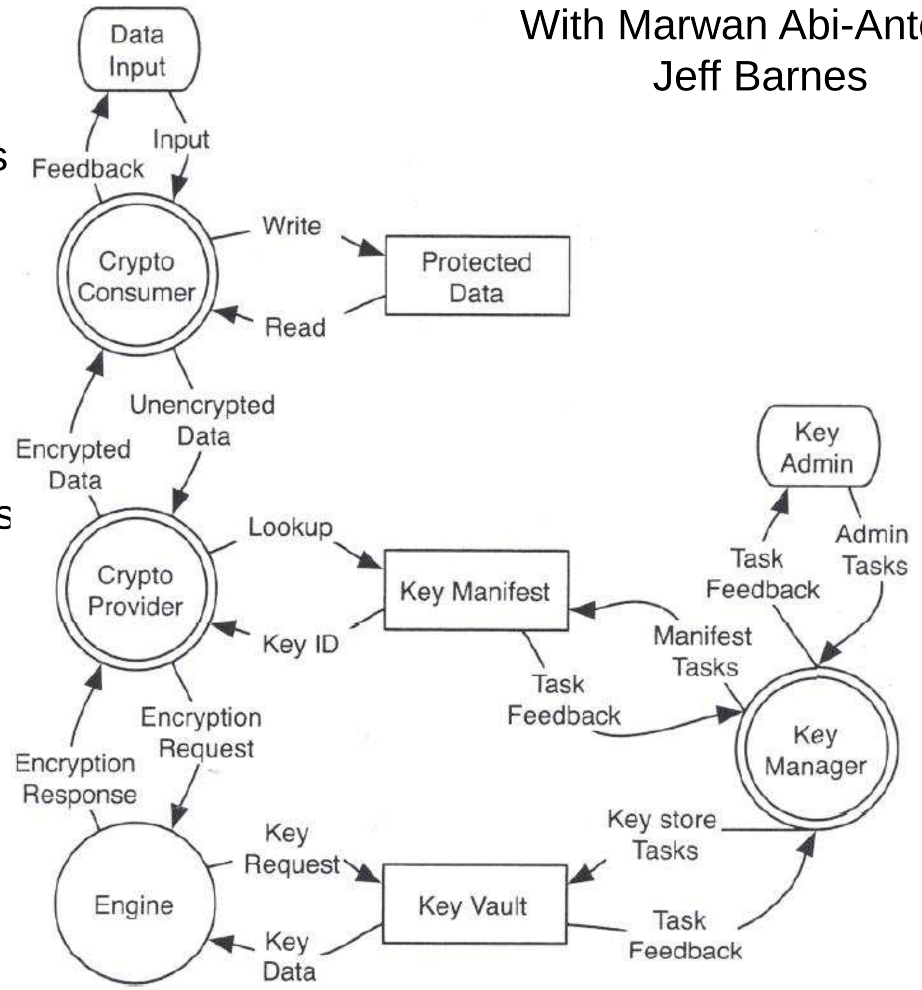
- Two current projects
  - Architectural information flow
  - Component interaction protocols

Information Flow: Threat Modeling

- Data flow diagrams
  - Processes
  - Data sources, sinks, and flows
  - Trust boundaries
- Security review
  - Completeness
  - Potential threats
- Key Microsoft process
  - Vista: 1400 data flow diagrams
  - Claim: 50% vulnerability reduction

- The **Big Question**: does reality match the diagram?

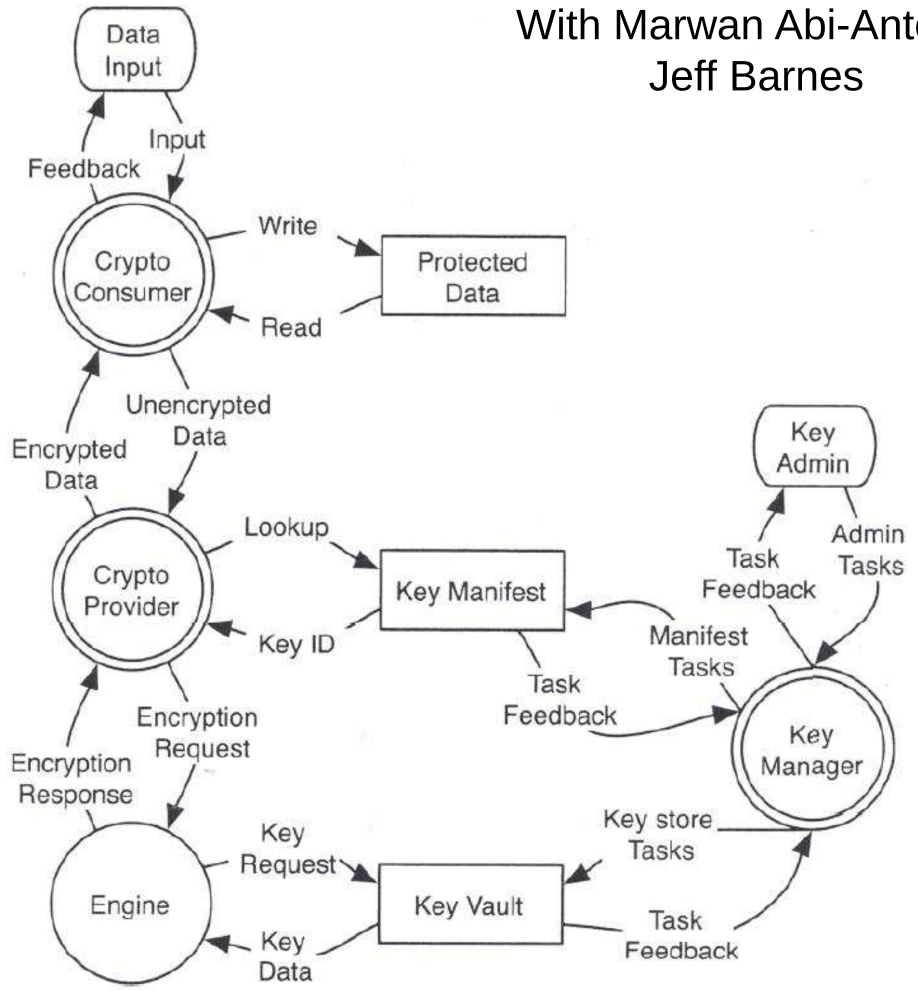
With Marwan Abi-Antoun, Jeff Barnes



Information Flow Case Study: CryptoDB

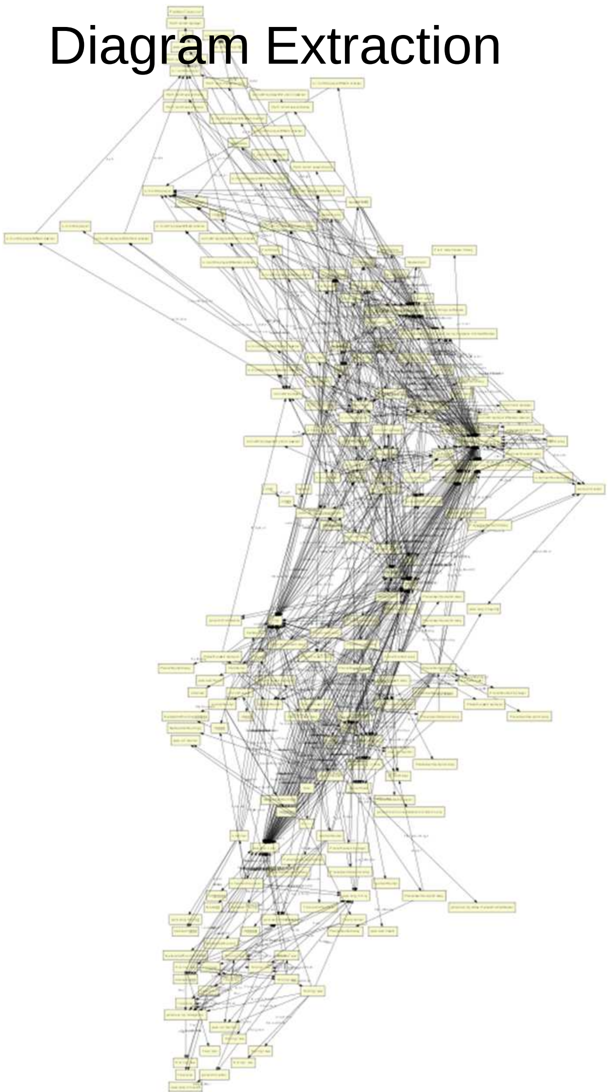
With Marwan Abi-Antoun, Jeff Barnes

- CryptoDB
  - Secure database system
  - From Kenan's textbook
  - Includes cryptography, access control
  - Java implementation
- Level 1 data flow diagram shown
- Study: extract flows
  - Associate code with elements
    - Comment-like program annotations
  - Generate as-built architecture
  - Compare to data-flow diagram

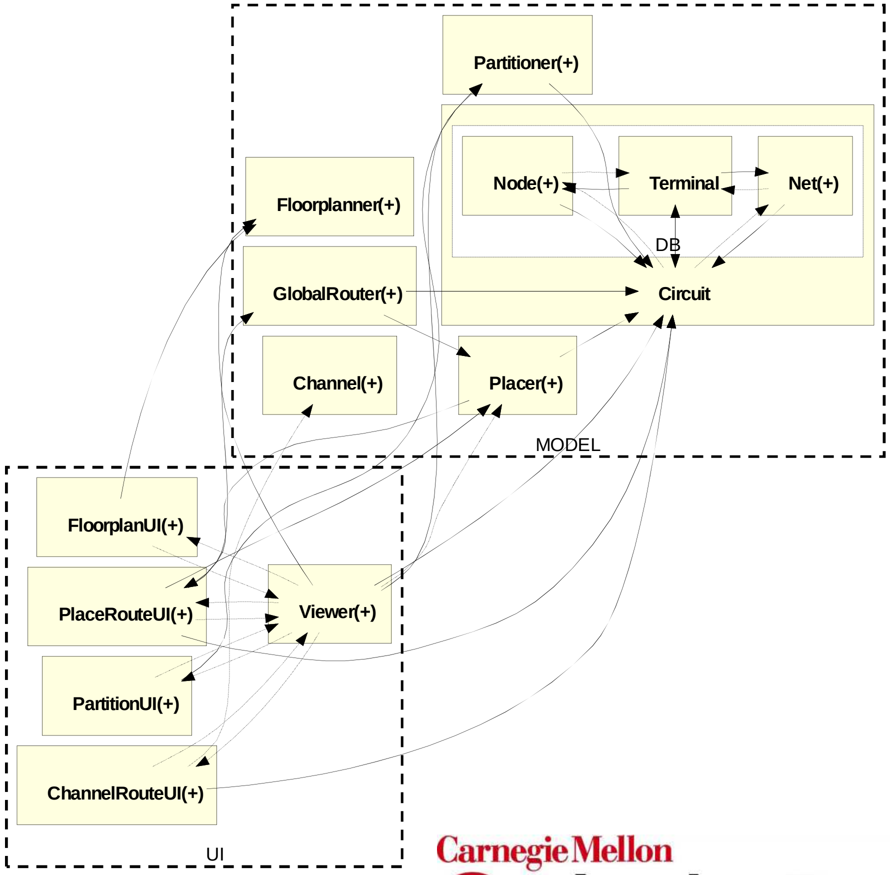


Key Challenge: Managing Complexity

Naïve Object Diagram Extraction



Design Intent-Based Extraction



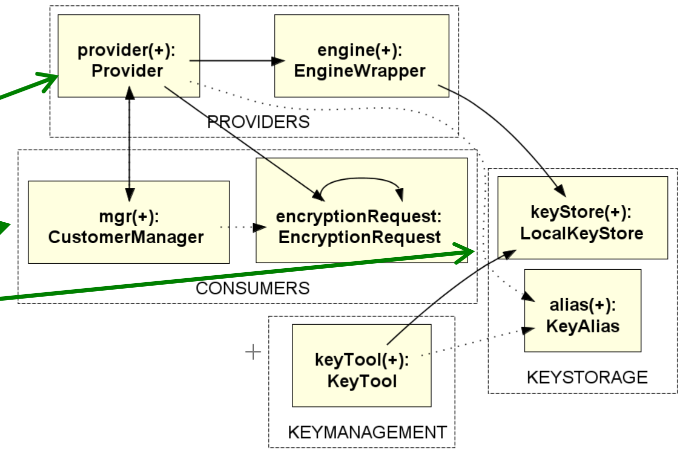
Carnegie Mellon CyLab
CONFIDENCE FOR A NETWORKED WORLD

Architectural Design Intent

With Marwan Abi-Antoun

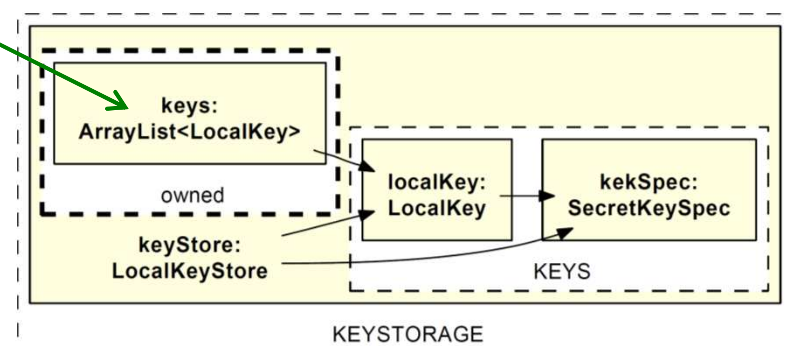
- Labeled groups
  - @Domain: Put in logical part of architecture

```
class Main {
    @Domain("PROVIDERS") Provider provider;
    @Domain("CONSUMERS") CustomerManager mgr;
    @Domain("KEYSTORAGE") LocalKeyStore keyStore;
}
```



- Data structure encapsulation
  - OWNED: Hide data objects within high-level abstractions

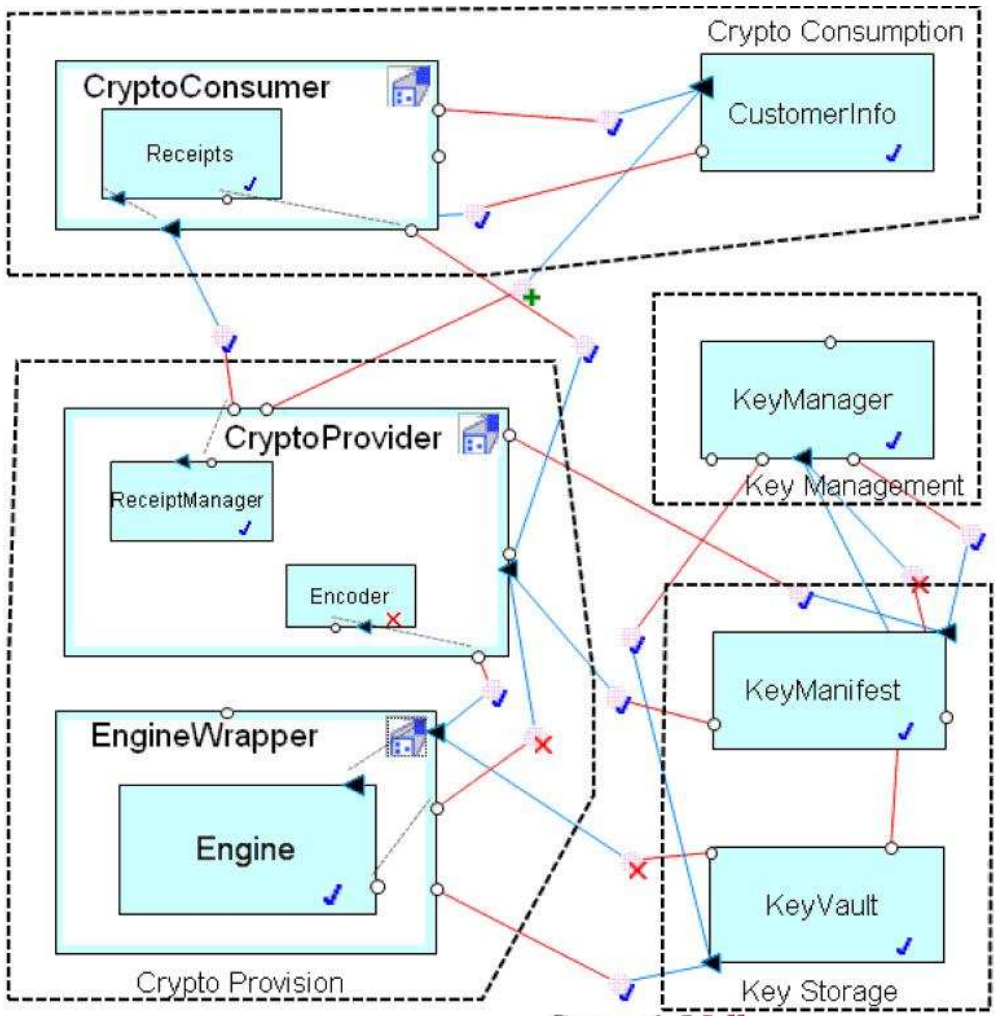
```
class LocalKeyStore {
    @Domain("OWNED<KEYS>") List<LocalKey> keys;
}
```



CryptoDB Case Study Results

[Abi-Antoun & Barnes, ASE '10]

- Comparison non-trivial
  - Names in code differ from diagram
  - Multiple design components merged into one
- Diagrams mostly consistent
  - A few differences marked with X (missing) or + (added)
- Conformance analysis easily found injected defects



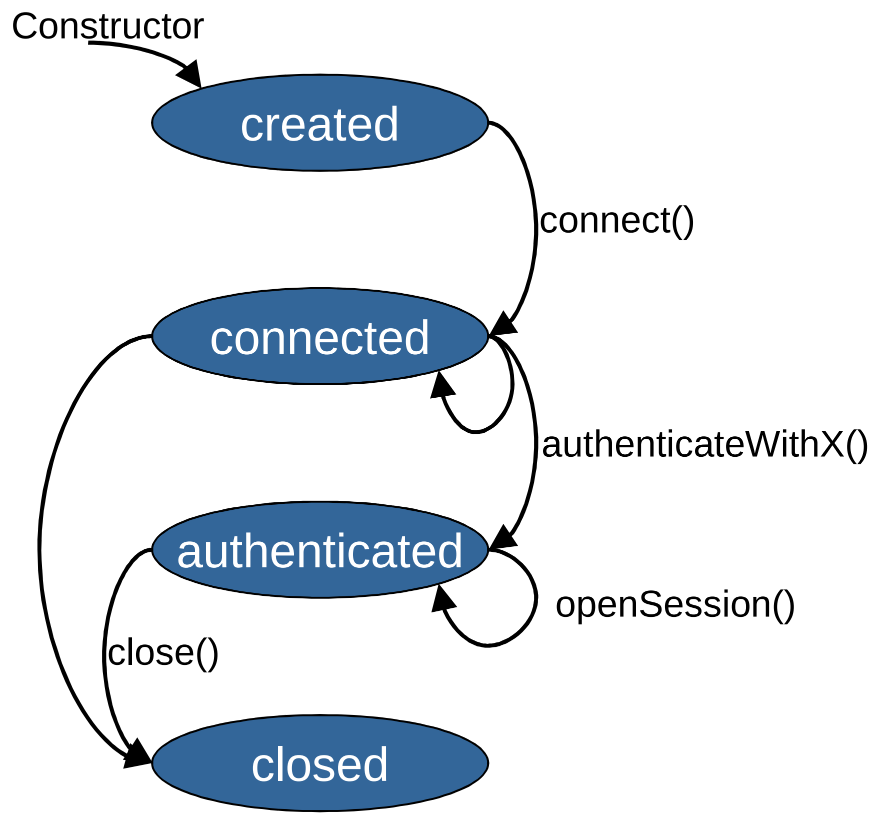
Carnegie Mellon CyLab CONFIDENCE FOR A NETWORKED WORLD

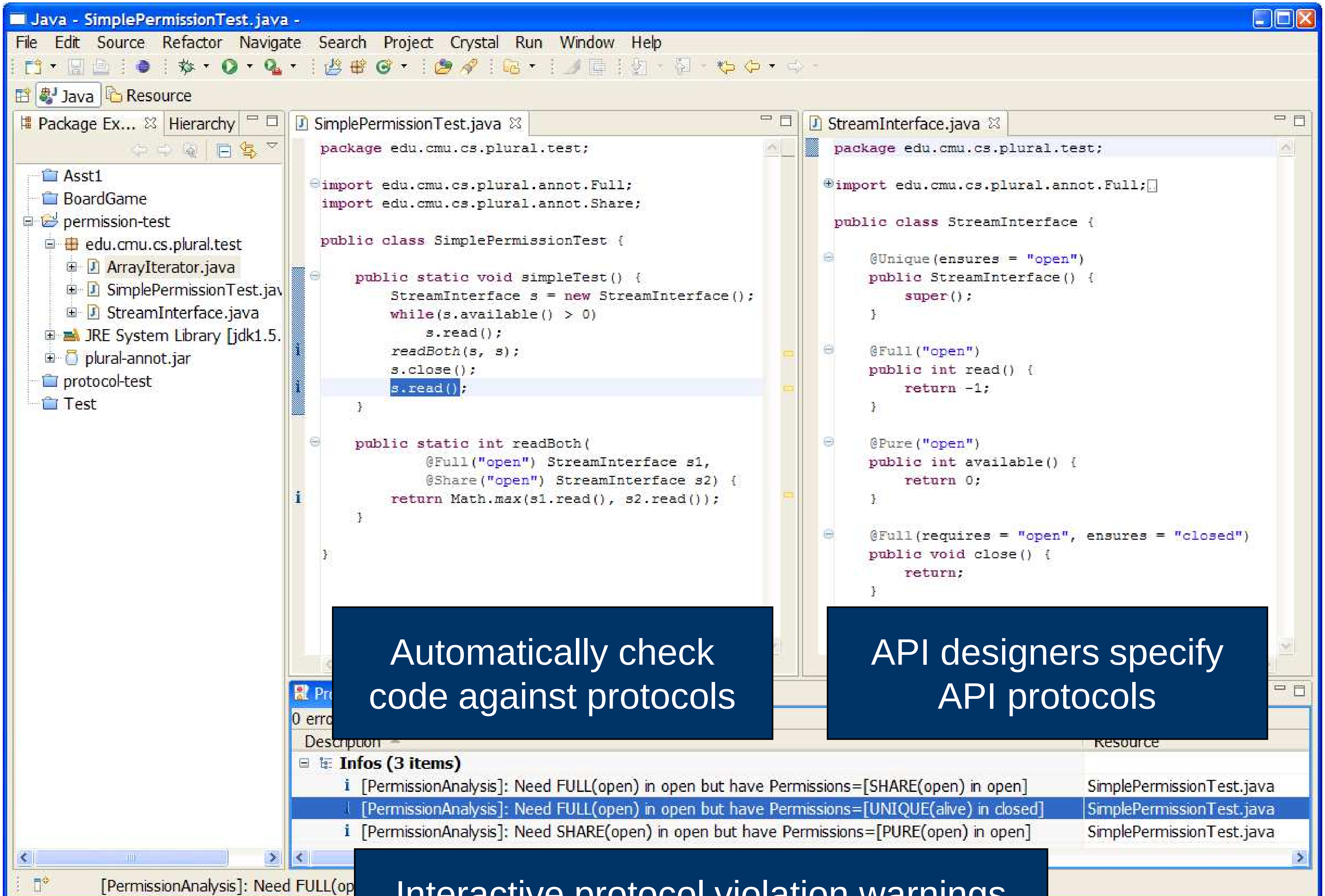
Secure Component Interfaces

With Kevin Bierhoff,
Nels Beckman

- Protocol constraints
  - Order of calls
  - Required argument state
- Challenges
  - What if we forget a step?
    - Especially on an error path?
  - What if multiple clients use the object?
    - If one client closes it, does that interfere with other clients?

Ganymed SSH-2 Protocol





Automatically check code against protocols

API designers specify API protocols

Interactive protocol violation warnings

Protocol Specification

With Kevin Bierhoff,
Nels Beckman

states open, closed
class StreamProtocol {
    true ⇒ **unique**(*this*) **in** open
    public StreamProtocol() { … }

    unique(*this*) **in** open
    ⇒ **unique**(*this*) **in** open
    public read(**int**) { … }

    unique(*this*) **in** open
    ⇒ **unique**(*this*) **in** closed
    public void close() { … }
}

- Declare states open, closed
- Constructor returns **unique** permission to open stream
- Read requires **unique** (exclusive) access to open stream
- Close transitions from open to closed

Protocol Verification

With Kevin Bierhoff,
Nels Beckman

```
states open, closed
class StreamProtocol {
    true ⇒ unique(this) in open
    public StreamProtocol() { … }

    unique(this) in open
    ⇒ unique(this) in open
    public read(int) { … }

    unique(this) in open
    ⇒ unique(this) in closed
    public void close() { … }
}
```

```
StreamProtocol s = new StreamProtocol();
            unique(s) in open
while(s.available() > 0)
      s.read(); // precondition satisfied
            unique(s) in open
s.close();
            unique(s) in closed
s.read(); // error: require open state
```


Carnegie Mellon CyLab CONFIDENCE FOR A NETWORKED WORLD

Modular Protocol Verification

With Kevin Bierhoff,
Nels Beckman

states open, closed
class StreamProtocol {
  true ⇒ **unique**(*this*) **in** open
  public StreamProtocol() { … }

  unique(*this*) **in** open
  ⇒ **unique**(*this*) **in** open
  public read(**int**) { … }

  unique(*this*) **in** open
  ⇒ **unique**(*this*) **in** closed
  public void close() { … }
}

  unique(*s*) **in** open
  ⇒ **unique**(*s*) **in** open
void process(StreamProtocol s) {
    unique(*s*) **in** open
  s.read(); *// precondition satisfied*
    unique (*s*) **in** open
}

StreamProtocol s = **new** StreamProtocol();
    unique(*s*) **in** open
while(s.available() > 0)
  process(s); *// precondition satisfied*
    unique(*s*) **in** open
s.close();
    unique(*s*) **in** closed

Implementation Verification

With Kevin Bierhoff,
Nels Beckman

states open, closed
class StreamWrapper {
  invariant open: **unique**(*str*) **in** open
  invariant closed: **unique**(*str*) **in** closed

  private StreamProtocol str;

  unique(*s*) **in** open
  ⇒ **unique**(*this*) **in** open
  StreamWrapper(StreamProtocol s)
  {
    unpacked(*this*, **unique**) ⊗
    unique(*s*) **in** open
    str = s;
    pack *this* **to** open;
  }

unique(*this*) **in** open
  ⇒ **unique**(*this*) **in** open
public int read() {
    unique(*this*) **in** open
  unpack *this*;
    unpacked(*this*, **unique**) ⊗
      unique(*str*) **in** open
  str.read(); *// precondition satisfied*
    unpacked(*this*, **unique**) ⊗
      unique(*str*) **in** open
  pack *this* **to** open;
    unique(*this*) **in** open
}

Implementation Verification (2)

With Kevin Bierhoff,
Nels Beckman

```
states open, closed
class StreamWrapper {
    invariant open: unique(str) in open
    invariant closed: unique(str) in closed

    private StreamProtocol str;

    unique(s) in open
    ⇒ unique(this) in open
    StreamWrapper(StreamProtocol s)
    {
        unpacked(this, unique) ⊗
        unique(s) in open
        str = s;
        pack this to open;
    }

    unique(this) in open
        ⇒ unique(this) in closed
    public void close() {
            unique(this) in open
        unpack this;
            unpacked(this, unique) ⊗
                unique(str) in open
        str.close(); // precondition satisfied
            unpacked(this, unique) ⊗
                unique(str) in closed
        pack this to closed;
            unique(this) in closed
    }
```

Field Experience [FSE '05, ECOOP '09]

With Kevin Bierhoff, Nels Beckman

Java Specifications

- Ganymed SSH-2 Protocol
- Collections and iterators
- I/O streams
- JDBC (database connectivity)
- Regular expressions
- Exceptions

Verification Studies

- **Depth:** Apache Beehive
  - Open Source resource access library
  - Has its own protocol
    - Common scenario: one API builds on another
  - Verified implementation uses JDBC correctly
- **Breadth:** PMD analysis tool
  - 39 kLOC of realistic code
  - Verified correct use of iterators

First field study of semantically deep protocol analysis for objects at this scale

Design Intent:
a Principled Approach to Application Security

- Design intent is critical
  - Can <u>assure</u> that secure design is realized in secure code

- Design intent is available today
  - Languages, libraries, and emerging tools
  - Provides immediate value

- Research is exploring new kinds of design intent