Improving Software Extensibility and Modularity

Marc Eaddy
Department of Computer Science
Columbia University
Overview

Situated Awareness
- EyeGuide

Extensibility
- Scripting

Modularity
- Measuring
- Improving
SITUATED AWARENESS
EyeGuide

- Subway map (public display)
- Lost user
- Nefarious straphangers

EyeGuide “steers” user’s gaze to map locations using whispered verbal cues

EyeGuide Contributions

- Introduces private “gaze steering”
  - Maps
  - Paintings
  - Sculpture
  - “Secret Service Man”
- Lightweight eye- and head-tracking headgear

Marc Eaddy
IMPROVING SOFTWARE EXTENSIBILITY
ThomsonONE

Real-time stock market data

- Distributed, multi-threaded, event-based desktop application
- I was system architect, lead developer, supervisor
- Voted “Product of the Year” by Inside Market Data
ThomsonONE scripting

- VBScript, JavaScript, etc.
- Associate scripts with app events
ThomsonONE scripting

- VBScript, JavaScript, etc.
  - Associate scripts with app events
- Powerful object model
  - Flexible end-user customization
- Rapid prototyping
Goblin

Platform for AR, VR, and 3D apps and games
Goblin features

- Written using C# and Managed DirectX
- Scene graph
  - Animation
  - Collision detection
  - Pathfinding AI
- Devices
  - Sony optical see-through head-worn displays
  - 6DOF device abstraction
  - InterSense 6DOF tracking devices
  - EssentialReality gloves
- Innovative runtime environment
  - “Scripting”
  - Edit-and-continue
Goblin extensibility

- Source files are “scripts”
  - .NET is language agnostic
    - C#, but also JScript, Python, Perl, etc.
  - High performance
    - Compiled, not interpreted
    - No glue code
- Edit-and-continue (demo)
  - Edit source files while app is running
  - Changes compiled in the background

IMPROVING SOFTWARE MODULARITY
Modular design

Dividing program into self-contained subprograms (modules)

- Arguably most important design property
- Reduces complexity
  - Separation of concerns [Dijkstra]
  - Modular reasoning
  - Information hiding [Parnas]
- Independent development/testing
  - Programming by contract (interfaces)
- Reusability and substitutability
  - Abstract data types [Liskov]
  - Low coupling and high cohesion [Steven, Myers, Constantine]
Concerns

*Anything that affects the implementation of a program (informal definition)*

- Features, requirements, design patterns, code idioms, etc.
- *Raison d'être* for code
  - Every line of code exists to satisfy some concern
The crosscutting concern problem

Some concerns difficult to modularize

- Code related to the concern is...
  - Scattered across (crosscuts) multiple files
  - Often tangled with other concern code
  - Degrades overall modularity
Example: Pathfinding in Goblin

Pathfinding is modularized
Example: Collision Detection

Collision detection not modularized
Impact on software quality

- Modularity is degraded
- Program harder to understand
  - “Where is collision detection implemented?”
  - “What is this code for?”
  - “What features will be affected by this change?”
- Program harder to evolve
  - Be careful to update all locations
  - Changes not localized
- Bottom line

Marc Eaddy
My research

- Crosscutting concern problem poorly understood
  - Hard to evaluate “solutions”
  - How big is the problem? (i.e., $1K or $1B? – Aho)
  - Few formal studies
- Determine impact of crosscutting concerns on software quality
  - Formalize problem
    - “A problem defined is half the solution” – Kachigan
  - Quantify impact
- Find solutions
Quantifying the problem

- **Concern metrics**
  - *Degree of scattering* – a modularity measure
  - *Degree of tangling* – measures separation of concerns

- **Concern location**
  - Hard to locate the code that implements a concern
    - >50% of maintenance time

- **Our tools**: ConcernTagger and Cerberus

---

Cerberus

Three experts are better than one (and two)

- Hybrid automated concern location technique
  - Information retrieval (IR)
  - Execution tracing (tracing)
  - Prune dependency analysis (PDA)

IR-based concern location

- i.e., Google for code
- Program entities are *documents*
- Requirements are *queries*
- Term extraction
  - Parse code and reqs spec to extract terms
  - Synonym and abbreviation expansion
  - Weigh terms based on local and global frequency (tf-idf)
- Similarity = cosine distance between document and query vectors
- Requires meaningful identifier names
Tracing-based concern location

- Observe elements *activated* when concern is *exercised*
  - Unit tests for each concern
- Relevance = tests for concern that activate element $e$ / tests that activate element $e$
- Tests must be *complete* and *exclusive*
  - Exercise all inputs for one concern only
PDA-based concern location

- Infer relevant elements based on relationship to relevant element $e$ (seed)
- Prune dependency analysis
  - Find references to $e$
  - Find dominators and dominatees of $e$
  - Find superclasses and subclasses of $e$
- Requires good seeds
  - Everything is dominated by Main

Cerberus

Concern Tests

Concern Descriptions

Source Code

Execution Tracing

Information Retrieval

Trace Ranking

IR Ranking

Combined Ranking

Seed Extraction

Initial Relevant Elements

Prune Dependency Analysis

Final Relevant Elements

Marc Eaddy
Cerberus Effectiveness

![Graph showing Cerberus Effectiveness]

- **Cerberus Effectiveness**
  - **F-Measure**
  - **Number of Concerns Located by Technique**
  - Lines represent different techniques:
    - IR + Tracing + PDA ("CERBERUS")
    - IR + PDA
    - IR + Tracing
    - IR
    - Tracing + PDA
    - Tracing

Graph indicates the effectiveness of different techniques in locating concerns, with Cerberus showing the highest effectiveness.
Do crosscutting concerns cause defects?

<table>
<thead>
<tr>
<th>Project</th>
<th>Language</th>
<th>Size (KLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mylyn–Bugzilla</td>
<td>Java</td>
<td>13</td>
</tr>
<tr>
<td>Rhino</td>
<td>Java</td>
<td>44</td>
</tr>
<tr>
<td>iBATIS</td>
<td>Java</td>
<td>13</td>
</tr>
</tbody>
</table>

- Created mappings
  - Requirement–code map
  - Bug–code map
  - Bug–requirement map (*inferred*)

- Correlated scattering, concern size, and bug count
  - Spearman correlation

Do crosscutting concerns cause defects?

- Found moderate to strong correlation between scattering and defects.
- As scattering increases so do defects.
- Tangling appears unrelated to defects.
How widespread is the problem?

- 5 case studies of OO programs
- Scattering
  - Concerns related to 6 classes on average
    - OO unsuitable for representing problem domain
  - Most (86%) concerns are crosscutting to some extent
    - Dispels “modular base” notion
    - General-purpose solution needed
- Tangling
  - Classes related to 10 concerns on average
  - Poor separation of concerns
    - Classes doing too much
  - Crosscutting concerns severely limit modularity
IMPROVING SOFTWARE MODULARITY
Aspect-oriented programming
Inject code at specific points in the program (essentially)

- An aspect
  - Class-like construct
  - Rules for
    - Injecting code into other methods
    - Adding members to existing classes (i.e. open classes)
  - Aspect code localized in one file—effects many files
Wicca#

- Extends C#
- Aspect annotations
- Open class annotations
- Statement annotations
- Side classes (future work)

Wicca
- Wicca# “ecosystem”
- Tools and runtime
Wicca

- Extends C# tool chain
  - wsc – Wicca# preprocessor/compiler
  - Phx.Morph – Post-compile tool
  - wdbg – Debugger

Wicca

- Extends runtime environment
  - wicca – Command-line shell
    - Dynamic aspect-oriented programming
    - Dynamic software updating
    - Edit-and-continue

Wicca architecture

VM

Debugger Host

ICorDbg

ICorDbgCal

Hosted App

IL + MD Deltas

Joinpoint Breakpoints

Front-end

Compiler

Preprocessor

Weaving

Binary Weaver

Breakpoint Weaver

Source Weaver

Wicca

Projects

App'

Assemblies

App''

Assemblies

C#

Build Info
Static transformation pipeline

Source Files → Wicca# Compiler → Compiled Program → Phx.Morph

Aspect Assemblies → Deltas
Woven Program → Break Points

Frontend

Compiled Program

Compiled Program

Backend

C# Compiler

Phoenix
Example: SimpleDraw

- SimpleDraw draws shapes on a display
- Naïve implementation couples Shape and Display concerns

**SimpleDraw.cs**

```csharp
public init() {
    s = new Shape[3];
    s[0] = new Point(10, 10);
    s[1] = new Point(5, 5);
    s[2] = new Line(new Point(1, 9),
                    new Point(9, 1));
    for (int i = 0; i < s.Length; i++)
        Display.instance().addShape(s[i]);
    Display.instance().update();
}
```

**Point.cs**

```csharp
public void moveBy(int dx, int dy) {
    x += dx;
    y += dy;
    Display.instance().update();
}
```

**Line.cs**

```csharp
public void moveBy(int dx, int dy) {
    a.moveBy(dx, dy);
    b.moveBy(dx, dy);
    Display.instance().update();
}
```
Example: SimpleDraw

- SimpleDraw draws shapes on a display
- Naïve implementation couples Shape and Display concerns

```csharp
SimpleDraw.cs
public init() {
    s = new Shape[3];
s[0] = new Point(10, 10);
s[1] = new Point(5, 5);
s[2] = new Line(new Point(1, 9),
                new Point(9, 9));
    for (int i = 0; i < s.Length; i++)
        Display.instance().addShape(s[i]);
    Display.instance().update();
}
```

```csharp
Point.cs
public void moveBy(int dx, int dy) {
    x += dx;
    y += dy;
    Display.instance().update();
}
```

```csharp
Line.cs
public void moveBy(int dx, int dy) {
    a.moveBy(dx, dy);
b.moveBy(dx, dy);
    Display.instance().update();
}
```
class DisplayUpdating {
    [Advice(AdviceType.after, "execution(void Shape.moveBy)")]
    [Advice(AdviceType.after, "execution(void Point.moveBy)")]
    [Advice(AdviceType.after, "execution(void Line.moveBy)")]
    static public void updateAfterMove() {
        Display.instance().update();
    }
    ...
}
Statement annotations

Difficult to modularize irregular concern code

- Concerns with “irregular” implementations
  - Error handling, assertions, optimizations, logging
  - Hard to modularize using regular aspects
- Apply aspects to specific locations, statements, or object instances

```java
[NonNull] AuthorizationRequest ar = new AuthorizationRequest(this, dest);
```

Future Work

- Reduce crosscutting
  - Side classes
- Improve concern location
  - Improve program analysis precision
  - Use machine learning to combine judgments
  - Incorporate smart “grep” and PDA into IDE
- Gather empirical evidence
  - Impact of reducing crosscutting
  - Impact of crosscutting on maintenance effort
  - Impact of code tangling on quality
Conclusion

- Powerful software extensibility mechanisms
- Dangers of poor concern modularity
- Improving concern modularity

Marc Eaddy
Acknowledgements

- Alfred Aho
- Gobin
  - Erik Petterson
  - John Waugh
  - Hrvoje Benko
- Wicca
  - Boriana Ditcheva
  - Rajesh Ramakrishnan
  - Adam Vartanian
- ConcernTagger/ConcernMapper
  - Vibhav Garg
  - Martin Robillard
  - Frédéric Weigand-Warr
- Cerberus
  - Giuliano Antoniol
  - Yann-Gaël Guéhéneuc
- Microsoft Phoenix Team
Questions?

Marc Eaddy
Columbia University
eaddy@cs.columbia.edu
ThomsonONE scripting
Edit-and-continue

Common Language Runtime

replaces
reads & appends

App.exe

foo()
New code of foo()

IL_0000:72
IL_0005:72

Profiler & Metadata APIs

Patcher
(Patcher.dll)

Metadata (Current)

Post-Edit Module

Post-Edit (Post-Edit)

Common Language Runtime

Watcher (Watcher.exe)

App-01.exe

foo.cs

foo()

Old code of foo()

IL_0000:72
IL_0005:72

C# Application

Current Assembly

Legend

File I/O
API Call
socket communication
Incremental Build File

Edit-and-continue
Phx.Morph architecture

**Phx.Morph**

Editors
- Open Classes, binary and breakpoint weaving

Phoenix-specific AOP

Attribute Handlers

**PEREW**

Assembly Re-Writer

**Morph Plugin**

**Phoenix**

**Phx.Aop**

AOP
- Joinpoints, pointcuts, ...

Attributes
- Custom AOP annotations

**.NET**
Confounding effect of size

- Larger concerns are buggier
- Scattering, size, and bug count strongly interrelated
  - Is scattering just measuring the “size effect”?
- Control for size using covariant analysis
  - Step-wise regression
  - Principal component analysis
- Scattering still explains some bug count variance
Side classes

*Powerful and disciplined class extension*

**Powerful**
- **Combines** subclasses, aspects, open classes, and refinements
- Unified **subclass-like** notation and semantics
- Multiple base classes can be extended simultaneously

**Disciplined**
- Extensions must be explicitly allowed
- Limits quantification
The expression problem

- Need to parse simple expression language
  - e.g., 3 + 4
  - Expressions: Literal, Add
  - Semantics: eval

- Extensibility goals
  - Easily add new expressions
  - Easily add new semantics

- Cannot do both easily

abstract class Expr {
    abstract int eval();
}

class Add : Expr {
    public Expr l, r;
    public Add(Expr l, Expr r) { this.l = l; this.r = r; }
    public int eval() { return l.eval() + r.eval(); }
}

• New concern: Add printing semantics
  – Concern is crosscutting!
Add printing using subclassing

abstract class PrintingExpr : Expr {
    abstract void print();
}

class PrintingAdd : PrintingExpr, Add {
    public void print() {
        Console.WriteLine(l + " + " + r);
    }
}

• Invasive
• Inflexible
Printing the side class way

abstract class PrintingExpr + Expr {
    abstract void print();
}

class PrintingAdd + PrintingExpr, Add {
    public void print() { Console.WriteLine(l + "+" + r); }
}

- Modularizes the printing concern
- Side class *implicitly* extends base class
- Clients (and other sub/side classes) oblivious
Caching result of eval()

- Concern: Cache expression evaluation
- Invalidate cache when expression changes
interface IObserver {
    void onChange();
}

class CachableExpr + Expr : IObserver {
    protected IObserver obs = null;
    protected int cache;
    protected bool isCacheValid = false;

    void onChanged() {
        isCacheValid = false;
        if (obs != null) obs.onChanged();
    }
}
**Intertype declarations**

```java
interface IObserver {
    void onChange();
}

class CachableExpr + Expr : IObserver {
    protected IObserver obs = null;
    protected int cache;
    protected bool isCacheValid = false;

    void onChanged() {
        isCacheValid = false;
        if (obs != null) obs.onChanged();
    }
}
```

- **Interface intertype declaration**: `interface IObserver {
    void onChange();
}
`
- **Member intertype declarations**: `class CachableExpr + Expr : IObserver {
    protected IObserver obs = null;
    protected int cache;
    protected bool isCacheValid = false;

    void onChanged() {
        isCacheValid = false;
        if (obs != null) obs.onChanged();
    }
} }`
Method chaining

class CachableAdd + Add {
    public Add(Expr l, Expr r) {
        base(l, r);
        this.l.obs = this;
        this.r.obs = this;
    }

    override public int eval() {
        if (!isCacheValid) {
            cache = base.eval();
            isCacheValid = true;
        }
        return cache;
    }
}
class CachableAdd + Add {
    public Add(Expr l, Expr r) {
        base(l, r);
        this.l.obs = this;
        this.r.obs = this;
    }

    override public int eval() {
        if (!isCacheValid) {
            cache = base.eval();
            isCacheValid = true;
        }
        return cache;
    }
}

Call ‘next’ delegate
Virtual fields

```java
override Expr l {
    set {
        base.l = value;
        this.l.obs = this;
        this.onChange();
    }
}

override Expr r {
    set {
        base.r = value;
        this.r.obs = this;
        this.onChange();
    }
}
```
Virtual fields

```java
override Expr l {
  set {
    base.l = value;
    this.l.obs = this;
    this.onChange();
  }
}

override Expr r {
  set {
    base.r = value;
    this.r.obs = this;
    this.onChange();
  }
}
```
Metaprogramming

Metavaraible

Limited quantification

```
override Expr [\$which = l | r] {
    set {
        base.$which = value;
        this.$which.obs = this;
        this.onChange();
    }
}
```
However, base class must permit extension

class Add : Expr {
    public virtual Expr l, r;
    public Add(Expr l, Expr r) { this.l = l; this.r = r; }
    public virtual int eval() { return l.eval() + r.eval(); }
}
Addresses issues of existing mechanisms

- **Subclasses**
  - Inflexible composition
  - Only extend one base class
  - Only extend instance methods
  - Requires invasive changes in clients

- **Open classes/refinements/mixins**
  - Need for composition ordering
  - Need for constructor and static member overriding

- **Aspects**
  - Lack of modular reasoning
  - Poor integration with existing OO syntax and semantics
  - Need for symmetric model
  - Matching wrong join points