Components in the Key of C

Craig Wittenberg
Microsoft Research

Talk Outline
- Our project
- Themes
- Examples and a demo
- Some comparative analysis
- Core elements of our work
- Related research
- Future research possibilities

Core elements of our work
- Component-based design/dev.
- Precise interface specs
- Composition
- Dependent typing
- Collections
- Testing
- Process: Factor and Finish

Comapps Today
- In MSR, but distinct
- Core group formed in 1994
- Foundation in COM
- 19 members + 1 visiting Prof.
- research.microsoft.com/comapps

Our Goals
- Create an application architecture
- Develop infrastructure and a component framework to realize that architecture
- Develop a new generation of application software with a high degree of reliability and leverage

Make Bill’s weekend with Basic yield a product!
- 1982: Cardfile
- 1996: Address book
- 2000: Works in a weekend or Outlook, or team manager, or Pocket Office
Anecdotal experience in MS

- Word and Excel refactoring
- Existing process problems:
  - Coordination and interconnection
  - Bugs
  - Stabilization period
  - Daily builds
  - Internal Betas
  - Internal Object Wars

Definitions: The easy ones

- Interface
  - Name, methods, contract
- Type
  - Name, set of interfaces, contract
- Contract
  - Description of resp. of both parties
  - Abstract model and pre/post conditions

Definitions: Component

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Szyperski, Component Software, pg. 34.

- Key points: Contracts, Composition, Dependencies, Deployment
- Add: Binary form and packaging

Physics of Software

- Mostly unconstrained today
- Constraints today are from
  - Time
  - Languages, tools
  - Machines
  - Processes
- Component-based development is a set of constraints

Compare/Contrast

- Source code
- Language centric
- Inside
- Direct refs to classes
- Class spec
- Binary code
- Language independent
- Outside
- Indirect refs to components
- Interface and types contracts

Themes

- Relationships
- Evolution
- Robust pieces
- Inside vs. outside
  - extreme encapsulation
- Separation of Concerns
- Building on other’s work
Relationships
- People
  - Dev teams
  - Research Collaboration
  - Markets
- Software:
  - Roles, Contracts
  - Reentrancy
  - Composition

Evolving Systems
- Planning for evolution
- Hot upgrading, robust upgrading
- Versioning Contracts (don't)
- Versioning Implementations (test)
- Names and versioning
- Late binding
- Marketplace needs

Robust, Useful Pieces
- Testing:
  - For more than the 80% case
  - Verification of coverage
- Well Documented
- Binary form
- Small variations
  - Shouldn't require a new piece
  - Use dependent typing

Inside .vs. Outside
Extreme Encapsulation
- Only access via interfaces
- Only use of services via explicit ref
- Inheritance-like features done via explicit composition of parts
- Composition from small to large
- Inside .vs. Outside of groups of components

Separation of Concerns
(or Orthoginality)
- Structure from algorithms
- Think of glue differently
- Design itself is well-factored
- Separate out concurrency, transactions, etc.

Examples of Systems
- HP Instrument Software System
- CMU composition work
- SPIN OS
- Chrysler C3
- Infinium MTS app
- Internet Explorer
- Octarine
Octarine

- Internal word processor prototype
- Written by Crispin Goswell
- Exemplifies many of the themes
- Uses of Octarine:
  - Inspiration for our work
  - Proving ground for composition ideas
  - Galen Hunt's Thesis
  - Millennium project

Octarine Demo

- Prototype written in C
- Looks familiar
- Adding Bullets, Tables, Music
- Followed by architectural drawings
- Also have COMCAD versions

Application Frame Architecture

Document Window Architecture

View, Layout, Data Architecture
Example: Single Page view

Example: Draw layer

Combined

Compound Text Architecture

Adding Bullets/Numbering

Marketplace for those solutions

- Few standards
- Common techniques
- Others issues
  - e.g., concurrency, tools
- One company or many
- Darwinism: only some will survive
Precise Interface Specs
a.k.a. Software Contracts

- Interface and types include:
  - Names (compile time and runtime)
  - Methods with pre/post conditions
  - Reference to previously defined types
  - Abstract model with invariants
  - State table, correspondence table
- Common pre/post conditions
  - e.g., basic parameters validity

ILight Methods and Model

- ILight, {2461A1A0-639E-11d2-874D-00AA0060FCA9}
  - TurnOn();
  - TurnOff();
  - SetIntensity(int desired);
- Model
  - boolean on;
  - int intensity; // range is 0..100.

ILight Pre/Post Conditions

- TurnOn():
  - Post: on == true.
- TurnOff():
  - Post: on == false.
- SetIntensity(desired):
  - Pre: desired >=0 && desired <= 100.
  - Post: intensity == desired.

Issues in Contracts

- Language for pre/post conditions
  - RESOLVE, OCL
  - Executable or not
  - Translate into testing support
- Mutation
  - multi-threading
  - reentrancy
  - events (e.g., change events)

Composition

- Uniform connection model
  - Custom connects are possible
  - Custom action on connect possible
  - Connection done at runtime
- Composites looks like components
  - Make and test small pieces
  - Build and use large ones

Composition, cont.

- Separate structure from algorithms
  - Definition/analysis of relationships
  - Eliminates explicit dependencies
- No glue as usually meant
  - wiring glue is factored out
    - Common runtime
    - Wiring descriptions
    - User-defined actions on connect
  - algorithmic glue is encapsulated
A Connectable Thing

- A connectable object implements:
  
  ```
  interface IConnector : IUnknown
  {
    GetElement (... which element..., REFIID rid, void **ppv);
    Connect (... which role..., IUnknown *punkOther);
    ...
  }
  ```

- And has associated type info

Example of Composition

- Button to turn on light
- Each piece has one+ contracts

```
<table>
<thead>
<tr>
<th>Button</th>
<th>IButton</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Button-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to-Light</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Desc.: Button, Adapter+stuff, Light, B->A, A->L

Encapsulation of the Switch

- Edge Connector used during wiring

```
<table>
<thead>
<tr>
<th>Edge Conn.</th>
<th>Button</th>
<th>IButton</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Button-</td>
<td>Light</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to-Light</td>
<td>Adapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Light</td>
</tr>
</tbody>
</table>
```

Desc.: Button, Adapter+stuff, Light, B->A, A->L, EC(out B, in A)

Status of our System

- Static wiring complete
- Visio extension to edit (COMCAD)
- Dynamic wiring in final prototype
- Using the system in several ways
  - C+COM translator
  - UI architecture prototypes
  - Factory simulation demo

Issues in Composition

- Pi-calculus and derivatives
- Testing
- Reentrancy

Dependent Typing

- Problem: how to customize an implementation to such that its behavior depends upon the customization.
- QueryInterface(REFIID, void **)
- Specialization technique
- Like partial evaluation
- Multiple specializations allowed
Specializable Adapter

- Data: IID1, IID2, method map
- Button-to-Light Adapter
- "Create"
- "Done"

Issues in Specialization

- Tool support
  - Defining format of specialization data
  - Creating specific specializations

Collections

- Encapsulate standard algorithms
- Use in a type-safe way
- Factor out
  - Data representation
  - Indexing mechanism
  - Storage

Collections, cont.

- IKeyedCollection, ISpecializeCollection are generic
- IKCGUIDIUnknown is type-specific
  - Lookup(GUID, IUnknown **)
- Data rep describes, but doesn’t store the data.
- Tools support is the only issue.

Testing

- Contract-based testing
  - Interface, type, class
  - Backdoor for model access
  - Wrappers/hooks for outgoing calls
  - Test the release binary form
  - Finding evidence of correctness
  - Measurement: 100% Coverage
Our Testing Framework

Testing an impl. of ILight

Issues in Testing

Process/organization

Factor Recommendations

Finishing Recommendations

- Auto-convert specs into test frameworks
- Race conditions, reentrancy
- Ensuring independence of mind
- Test ordering
- Beyond 100% coverage (Beizer)
- Testing composites

- Scalable overall organization
- Small teams that do it all
- A bit like Brooks suggested
- Write all tests and documentation
- Can contract out for help
- Feedback loops at various levels
- Factor/Finish
- Use of C, C++COM

- Favor smaller, simpler interfaces
- Plan for extensibility
- Favor composition style over framework style
- Understand 3+ potential uses
- Address large scale performance issues
- Verify factoring before implementing
- Study other designs

- Finish for more than one client
- Address small scale performance issues
- Decide on packaging plan
- Decide on versioning plan
- Even before you ship the first version
Finishing, cont.

- Write and run full regression tests per interface
- Conformance tests for clients to use
- Avoid tightly-coupled development if possible
- Few OS dependencies
- Accurate Documentation

Related Work

- Brooks, Wirth, Parnas: various
- Gall: Systemantics
- Petroski: Designs and Errors
- Williams: Inheritance + QI
- Meyer: Design by Contract
- Kiczales: Open Implementation
- Harel, Wegner: Turing revisited

Related Work, cont.

- CMU, Nierstrasz: Composition
- Gamma, et. al., Coplien: Patterns
- D’Souza & Wills: Catalysis
- Beizer, RST Corp: Testing
- Beck and others: Extreme Prog.
- Holmes, et. al: Synchronization
- Szyperski: Beyond OOP

Summary

- Background to our group’s work
- Complex, robust, evolving software systems
- Contracts, dependent typing, composition, collections, testing
- research.microsoft.com/comapps
- Let’s talk