

Software Architecture

Marcel Weiher

@mpweiher



Why Software Architecture?

- You have no choice!
- Default architecture: Big Ball of Mud [Foote97]

Why Software Architecture?

- You have no choice!
- Default architecture: Big Ball of Mud [Foote97]



Why Software Architecture?

- You have no choice!
- Default architecture: Big Ball of Mud [Foote97]



7.5 m tonnes

Why Software Architecture?

- You have no choice!
- Default architecture: Big Ball of Mud [Foote97]



7.5 m tonnes



Why Software Architecture?

- You have no choice!
- Default architecture: Big Ball of Mud [Foote97]



7.5 m tonnes



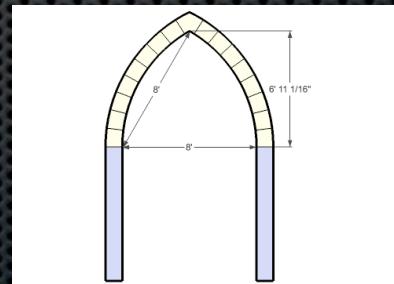
160k tonnes

Why Software Architecture?

- You have no choice!
- Default architecture: Big Ball of Mud [Foote97]



7.5 m tonnes



160k tonnes



Why Software Architecture?



- ❖ mono GUI
- ❖ Networking
- ❖ Word Processing
- ❖ 10 KLOC

- ❖ color GUI
- ❖ Internet
- ❖ Word Processing
- ❖ 100 MLOC

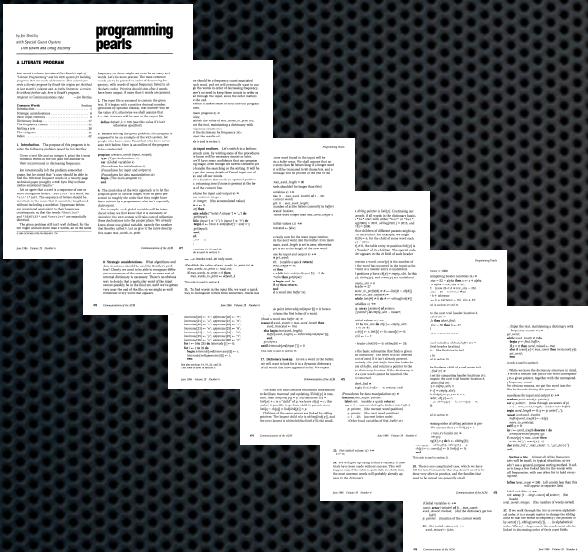
- ❖ color GUI
- ❖ Internet
- ❖ Word Processing
- ❖ 100 KLOC

Why Software Architecture?

- ## ◆ Bentley's Challenge: top n words

Don Knuth

Doug McIlroy



```
tr -cs A-Za-z '\n' |  
tr A-Z a-z |  
sort |  
uniq -c |  
sort -rn |  
sed ${1}q
```

Overview

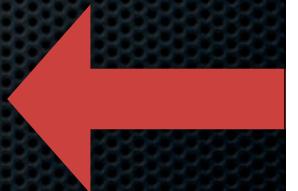
- Theory: components, connectors, configurations, styles, adapters, ADLs
- Architectural elements in Cocoa
- Example: architecture in Wunderlist
- Special architectural issues for UI software
- Linguistic Support

Architecture: Theory

- Organizing software into interconnected parts
- Components: parts (computation,state)
- Connectors: communication/organization

Architecture: Theory

- Organizing software into interconnected parts
- Components: parts (computation,state)
- Connectors: communication/organization



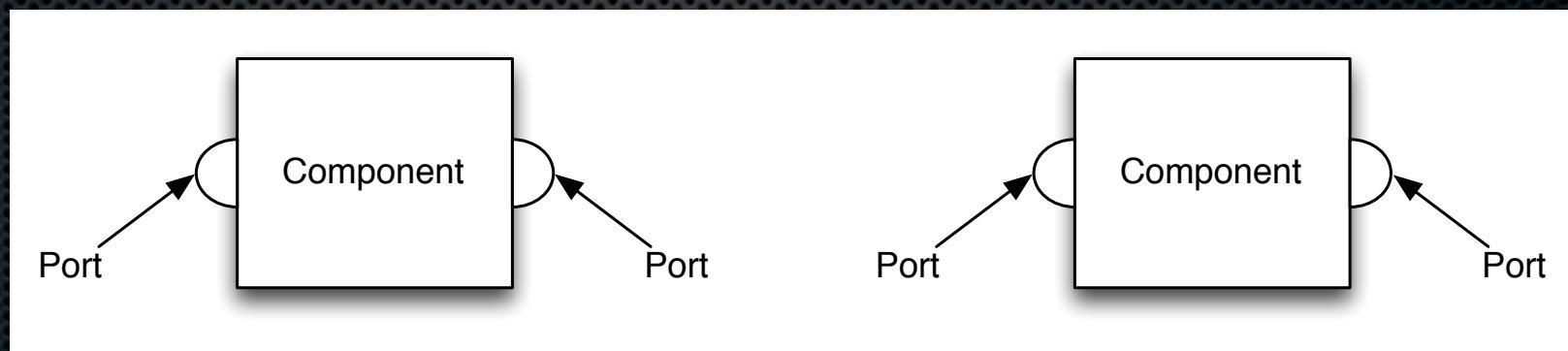
Architecture: Theory

- Organizing software into interconnected parts
- Components: parts (computation,state)
- +- Connectors: communication/organization
- =- Configurations: systems, programs, components
- Styles: classes/types of configurations



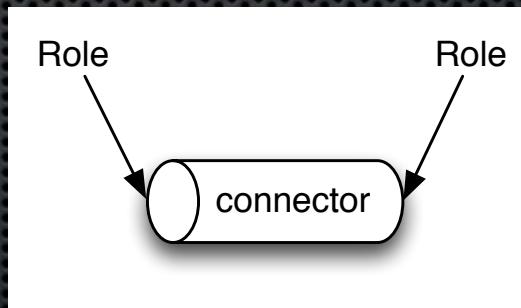
Component

- Computation
- Storage
- Attaches to Connectors via Ports



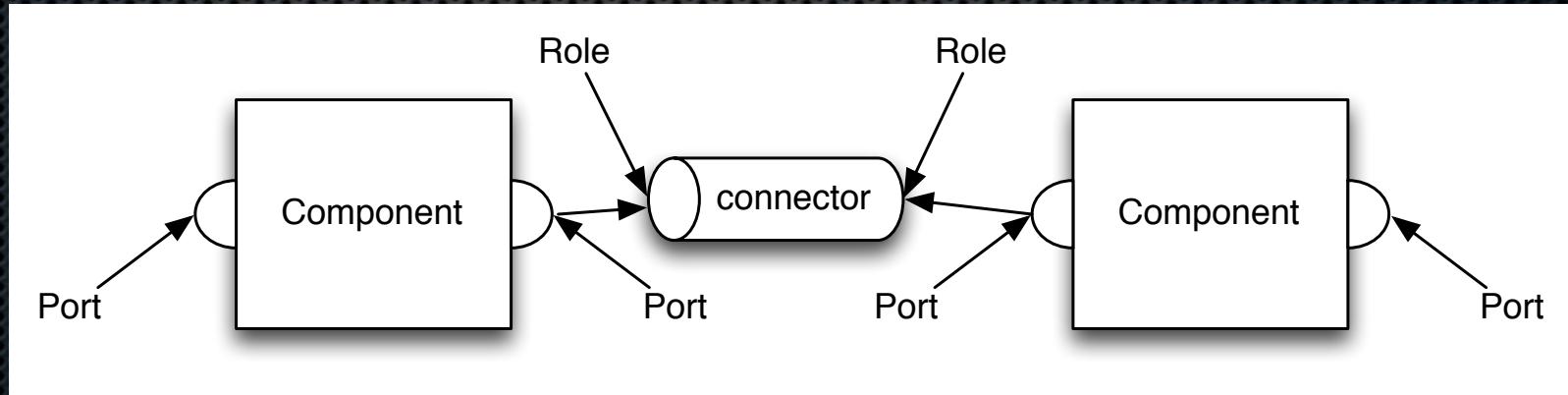
Connector

- Communication
- Interaction
- Attaches to Components via Roles



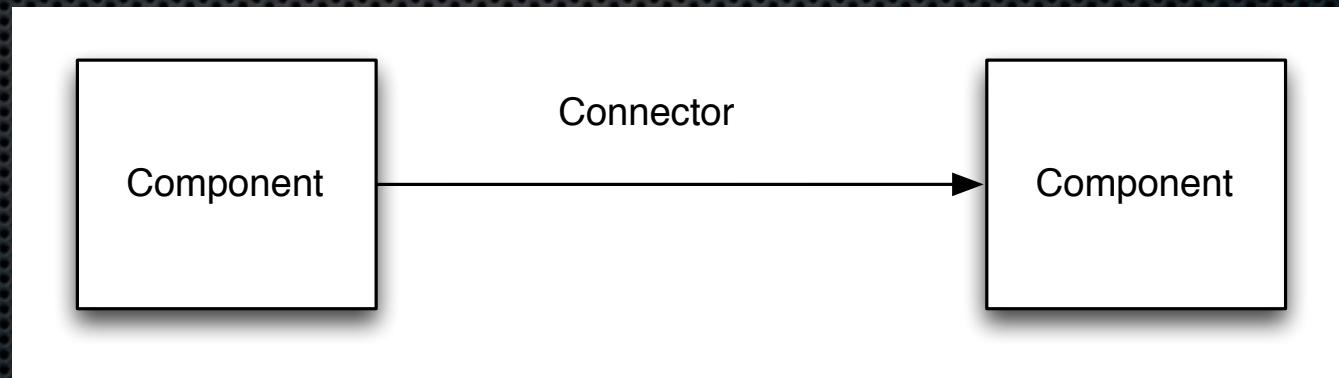
Configuration

- Also called “System”
- Components joined together via Connectors



Configuration

- Also called “System”
- Components joined together via Connectors



ADLs: ACME

```
System ls-wc {
    Component ls = {
        Port stdin,stdout;
    };
    Component wc = {
        Port stdin,stdout;
    }
    Connector pipe = {
        Role source,sink;
    }
}
```

ADLs: ACME

```
Attachments = {  
    ls.stdout to pipe.source;  
    wc.stdin  to pipe.sink;  
}  
}
```

Unix shell:

```
ls | wc
```

Connectors / Styles

- Call and return (procedural / OOP)
- Dataflow (Pipes/Filters, “reactive”, constraints)
- Implicit Invocation / Events
- Data access
- Messaging
- REST

Style: Messaging

- Kay: sorry I coined OO, it's about messaging
- synchronous / asynchronous
- local / remote
- call-based / reified
- early-bound / late-bound

Adapters

- stdio: adapts C call/return to Unix Pipe/Filter
- Objective-C: adapts C to Smalltalk messaging
- Lots of UI-programming-specific adapters in Cocoa

Architectural Adapters in Cocoa

- NSRunLoop: Events -> Messages
- Target/Action: User Input -> Messages
- Notifications: Implicit Invocation
- Bindings: Dataflow (two-way constraints)
- performSelectorOn.. : Actors
- Nibs, Storyboards: Configurations

IB as an ADL

- Connectors make NIBs possible
- Having only a graphical ADL is limiting
- No abstraction, encapsulation, reuse
- Alternative: code without ADL abstractions

Example: Wunderlist 3

- Real MVC
- Hexagonal (Ports/Adapters)
- In-Process REST
- Network: callback hell (replace with dataflow)

Example: Wunderlist 3

- Real MVC
- Hexagonal (Ports/Adapters)
- In-Process REST
- Network: callback hell (replace with dataflow)



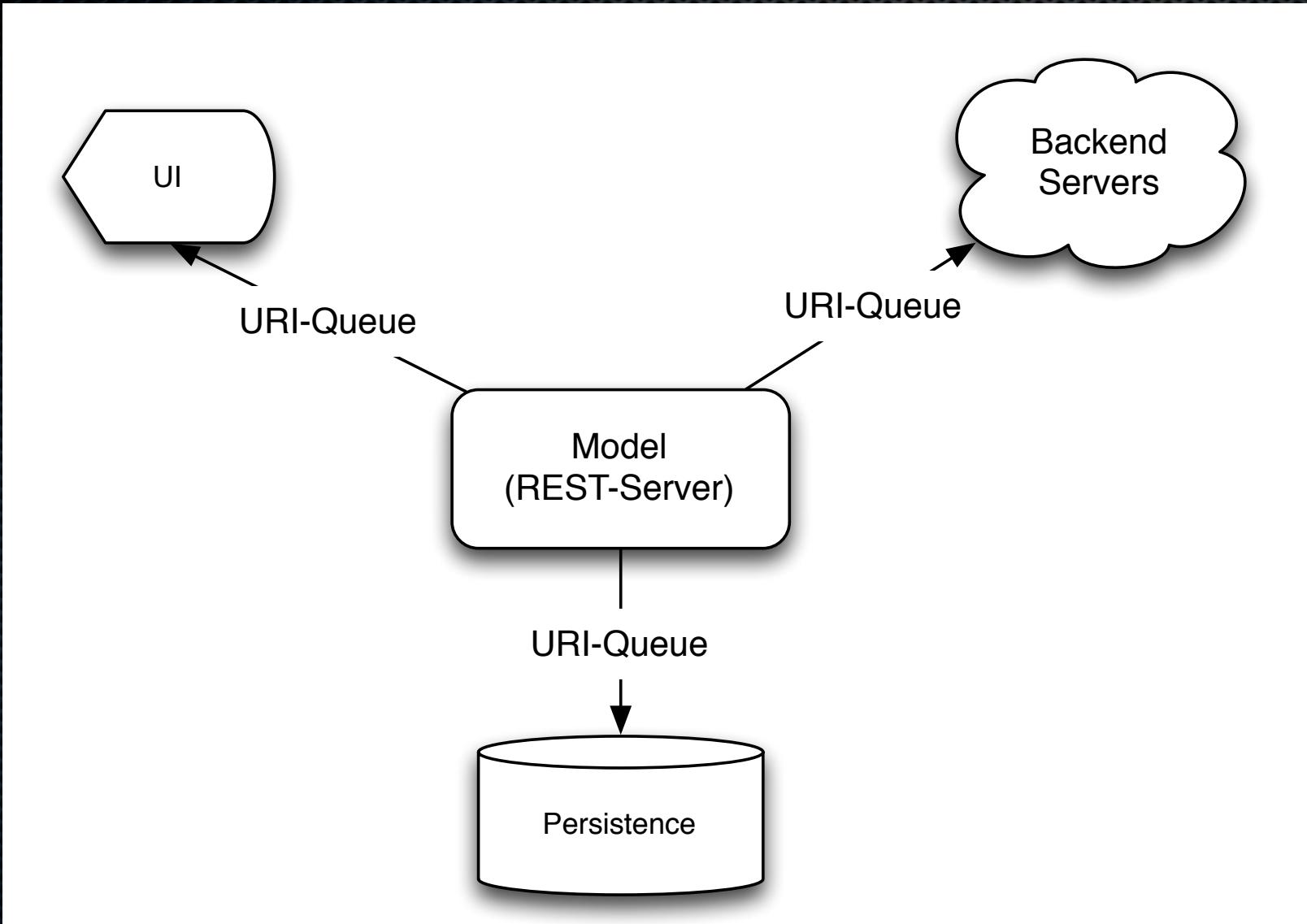
In-Process REST

- Use URLs (StoreReference) to refer to model
- URLs can be persisted
- URLs are structured:

wlstore:<entity>/container/<id>/object/<id>

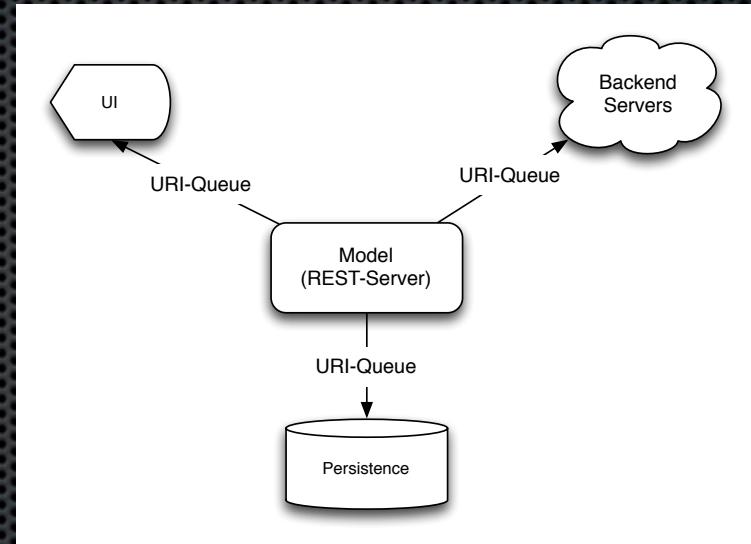
- URLs can refer to single objects or groups

Overall Structure



URI Queues

- A queue of references
- Asynchronous
- Persistable (network)
- Uniqued (network)
- “Bucketized” (storage)
- Dynamic coalescing/throttling (UI)



Architectural Mismatch



The image shows the front cover of a technical paper. At the top left is a small icon of a classical column. The title 'Architectural Mismatch' is at the top center, followed by a subtitle 'Why Reuse Is So Hard'. Below that is the author's name 'DAVID GARLAN, ROBERT ALLEN, and JOHN OCKERBLOOM' and their affiliation 'Carnegie Mellon University'. The main text begins with a large, bold letter 'F'.

◆ Why isn't there more progress toward building systems from existing parts? One answer is that the assumptions of the parts about their intended environment are implicit and either don't match the actual environment or conflict with those of other parts. The authors explore these problems in the context of their own experience with a compositional approach.

Future breakthroughs in software productivity may well depend on the ability of reuse to combine existing pieces of software to produce new applications. The current build-from-scratch techniques that dominate most software production must eventually give way to techniques that emphasize construction from reusable building blocks. If not, software designers may hit a production ceiling in generating large, high-quality software applications.

The last decade has seen increased support for compositional approaches to software. There is considerable research and development in reuse; industry standards like CORBA have been created for component interaction; and many domain-specific archi-

lectures, toolkits, application generators, and other related products that support reuse and open systems have been developed.

Yet the systematic construction of large-scale software applications from existing parts remains an elusive goal. Why? Some of the blame can rightfully be placed on the lack of pieces to build on or the inability to locate the desired pieces when they do exist.

But even when the components are in hand, significant problems often remain because the chosen parts do not fit well together. In many cases these mismatches may be caused by low-level problems of interoperability, such as incompatibilities in programming languages, operating platforms, or database schemas. These are hard

IEEE SOFTWARE 0740-1295/94/0204-00 © 1994 IEEE

- Packaging doesn't match
- Subclassing required
- Thread of control
- Dependencies

Architectural Mismatch



Architectural Mismatch: Why Reuse Is So Hard

DAVID GARLAN, ROBERT ALLEN, and JOHN OCKERBLOOM
Carnegie Mellon University

◆ *Why isn't there more progress toward building systems from existing parts? One answer is that the assumptions of the parts about their intended environment are implicit and either don't match the actual environment or conflict with those of other parts. The authors explore these problems in the context of their own experience with a compositional approach.*

F

uture breakthroughs in software productivity may well depend on the ability to reuse existing parts to combine existing pieces of software to produce new applications. The current build-from-scratch techniques that dominate most software production must eventually give way to techniques that emphasize construction from reusable building blocks. If not, software designers may hit a production ceiling in generating large, high-quality software applications.

The last decade has seen increased support for compositional approaches to software. There is considerable research and development in reuse; industry standards like CORBA have been created for component interaction; and many domain-specific archi-

tecures, toolkits, application generators, and related products that support reuse and open systems have been developed.

Yet the systematic construction of large-scale software applications from existing parts remains an elusive goal. Why? Some of the blame can rightfully be placed on the lack of pieces to build on or the inability to locate the desired pieces when they do exist.

But even when the components are in hand, significant problems often remain because the chosen parts do not fit well together. In many cases these mismatches may be caused by low-level problems of interoperability, such as incompatibilities in programming languages, operating platforms, or database schemas. These are hard

update
25th-anniversary top picks.....

Architectural Mismatch: Why Reuse Is Still So Hard

David Garlan, Carnegie Mellon University

Robert Allen, IBM

John Ockerbloom, University of Pennsylvania

In 1995, when we published "Architectural Mismatch: Why Reuse Is So Hard"¹ (an earlier version of which had appeared elsewhere²), we had just lived through the sobering experience of trying to build a system from reusable parts but failing miserably. Although the system had the required functionality, developing it took far longer than we had anticipated. More important, the resulting system was sluggish, huge, brittle, and difficult to maintain.

Why had things gone so awry? The usual explanation for reuse failure did not seem to apply. The parts had been engineered for reuse. We were reuse experts, skilled at identifying what had reuse code and were familiar with all the parts' implementation languages. We knew what we wanted, and we used the parts in accordance with their advertised purposes.

In searching for answers, we realized that virtually all our problems had resulted from incompatible assumptions that each part had made about its operating environment. We termed this phenomenon "architectural mismatch," and our article tried to explore in more depth how and why it occurs.

The Problem
Specifically, we examined four general categories for assumptions that can lead to architectural mismatch:

- the nature of the components (including the control model);
- the nature of the connectors (protocols and data);
- the global architectural structure; and
- the construction process (development environment and build).

We also noted three facets of component interaction in which assumptions can lead to mismatch:

- the infrastructure on which the component relies;
- application software that uses the component (including user interfaces); and
- interactions between peer components.

Figure 1 illustrates these facets.
Finally, we argued that to make progress,

In January 2009, I asked for follow-up pieces from several sets of authors whose insightful and influential Software classics made the magazine's 25th-anniversary top picks list (Jan/Feb. 2009, pp. 9–11). Here, David Garlan, Robert Allen, and John Ockerbloom provide fresh perspectives on their winning article, addressing how their thinking has evolved over the years, what has changed, and what has remained constant.

—Hakan Erdogan, Editor in Chief

66 IEEE SOFTWARE Published by the IEEE Computer Society

0740-7459/09/0206-0066\$25.00 © 2009 IEEE

Architectural Mismatch II

Programs = Data + Algorithms + Architecture:
consequences for interactive software

Stéphane Chatty

ENAC, Laboratoire Informatique et Interaction,
7 avenue Edouard Belin, 31055 Toulouse Cedex, France
and
IntuLab, Prologue 1, La Pyrénée, 31672 Labège Cedex, France
<http://recherche.enac.fr/~chatty>
chatty@intulab.com, chatty@enac.fr

Abstract. This article analyses the relationships between software architecture, programming languages and interactive systems. It proposes to consider that languages, like user interface tools, implement architecture styles or patterns aimed at particular stakeholders and scenarios. It lists architecture issues in interactive software that would be best resolved at the language level, in that conflicting patterns are currently proposed by languages and user interface tools, because of differences in target scenarios. Among these issues are the contravariance of reuse and control, new scenarios of software reuse, the architecture-induced concurrency, and the multiplicity of hierarchies. The article then proposes a research agenda to address that problem, including a requirements- and scenario-oriented deconstruction of programming languages to understand which of the original requirements still hold and which are not fully adapted to interactive systems.

1 Introduction

Niklaus Wirth, renowned computer science teacher and programming language designer, wrote in 1975 a reference book entitled “Algorithms + Data structures = Programs” [1] that has influenced thousands of programmers. It may be that his equation was incomplete though. Software architecture, that is the way of organising software into interconnected parts, has progressively become recognized as a central issue in programming and software engineering, to the point where students now spend more time learning about patterns and frameworks than data and algorithms. Yet, software architecture is still mostly considered a separate issue from programming languages. We contend that this is a serious issue for the software engineering of interactive systems. Short of being able to write ‘Programs = data + algorithms + architecture’ and addressing architecture issues at the language level, the architecture of interactive software may be doomed to inconsistency and complexity.

The architecture of interactive software has been heavily studied and many influential results in software architecture were obtained by researchers with a background in interactive software, or derived from their work. Compare for example the authors and topics in the following list of publications: [2–10]. Still,

- Situation is a lot worse for interactive software
- Architectural style of languages doesn't match
- Our day-to-day slog
- Cocoa adapters help
- But: linguistic support needed

Swift

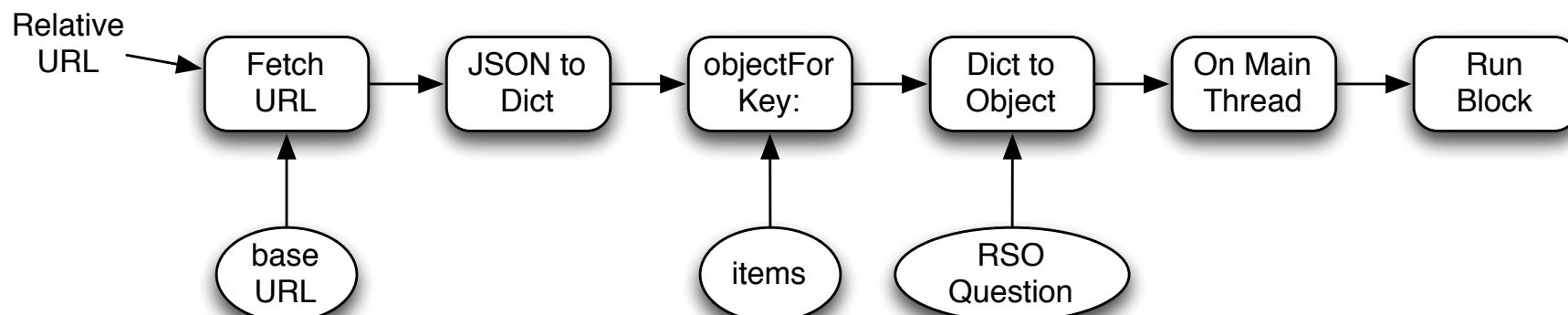
- Improvements for algorithmic code (compilers)
- Algorithmic code is rare in UI programs
- Restricts dynamic features...
- ... crucial for architectural adaptation...
- ... crucial for UI programming.

Outlook: Linguistic support

- Objective-Smalltalk: compact ADL
- Flexibility of messaging for other connectors
- Dataflow
- Flexible data access (Polymorphic Identifiers)
- Dataflow constraints
- Tooling

Dataflow

```
URLFetchStream streamWithBaseURL: baseUrl  
    parseJSON  
        objectForKey: 'items'  
    dict2objwithClass: (RSOQuestion class)  
    onMainThreadStream  
        onBlock:target.
```



Polymorphic Identifiers

- Full URIs as programming language identifiers

html := <http://www.amazon.com/>

file:/tmp/{name} := ‘Hello World’.

- Unified composite references

textField/intValue := defaults:tempCelsius

- First class references without strings

ref:textField/intValue

Dataflow Constraints I

```
loginButton/enabled |=  
    self validLogin:userField/stringValue  
        password:passwordField/stringValue  
        repeat:repeatField/stringValue.
```

Dataflow Constraints II

```
celsius := defaults:temperature.  
celsiusTextField/intValue =|= celsius.  
fahrenheitTextField/intValue =|= fahrenheit.  
celsius      |= fahrenheit * 5/9 - 32.  
fahrenheit   |= (celsius + 32) * 9/5.  
defaults:temperature |= celsius.
```

Conclusions + Q&A

- Software Architecture key to success of
 - Cocoa(Touch)/Objective-C/Interface Builder
 - Projects
- Many problems still not solved (20+ years)
- Swift is a step back, maybe: <http://objective.st>

