

# Towards Language Support for Component-Oriented Real-Time Programming

---

Michael Franz, Peter H. Fröhlich, and Thomas Kistler  
University of California, Irvine

# Object Oriented Programming

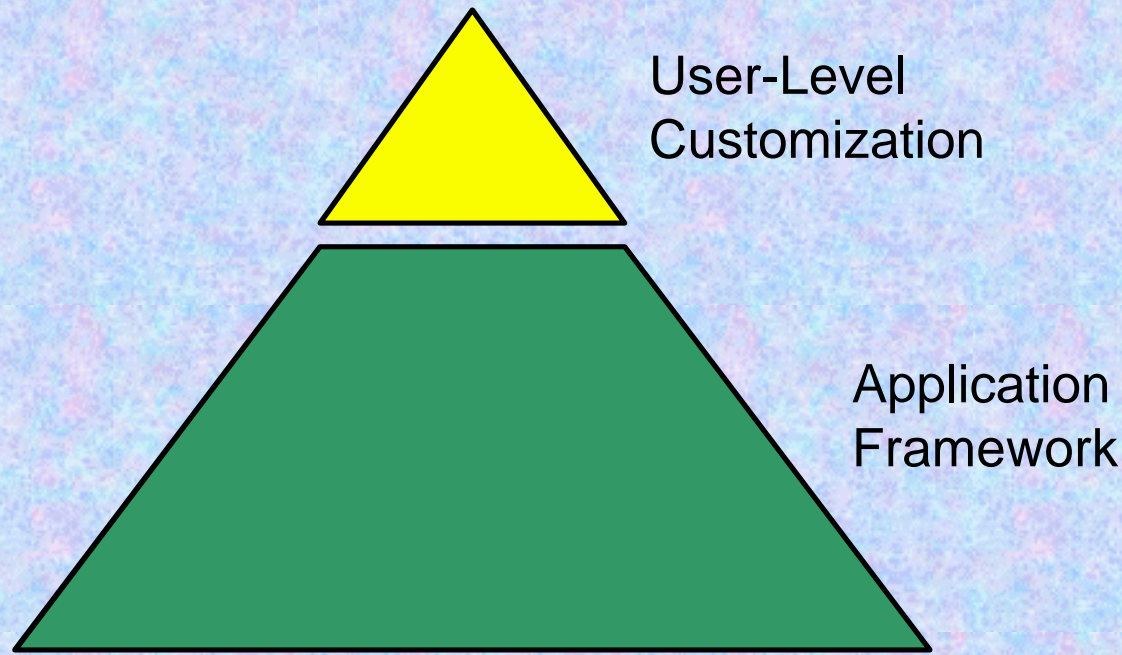
---

- ◆ current practice is framework-centered
- ◆ frameworks are “customized” with user-level functionality
- ◆ this is usually achieved by using the inheritance mechanism of object-oriented languages
  - user-level functions are implemented by overriding the default behavior of the framework

# Object Oriented Programming

---

- ◆ frameworks are usually relatively large in comparison to the actual customization code



# What are Software Components?

---

- ◆ one of the “holy grails” of software engineering
- ◆ McIlroy, 1968:
  - “mass production”
  - independently developed parts
  - black-box (re-)use
  - substitutable with equivalent ones (multi-sourcing)

# Are *Java Applets* Components?

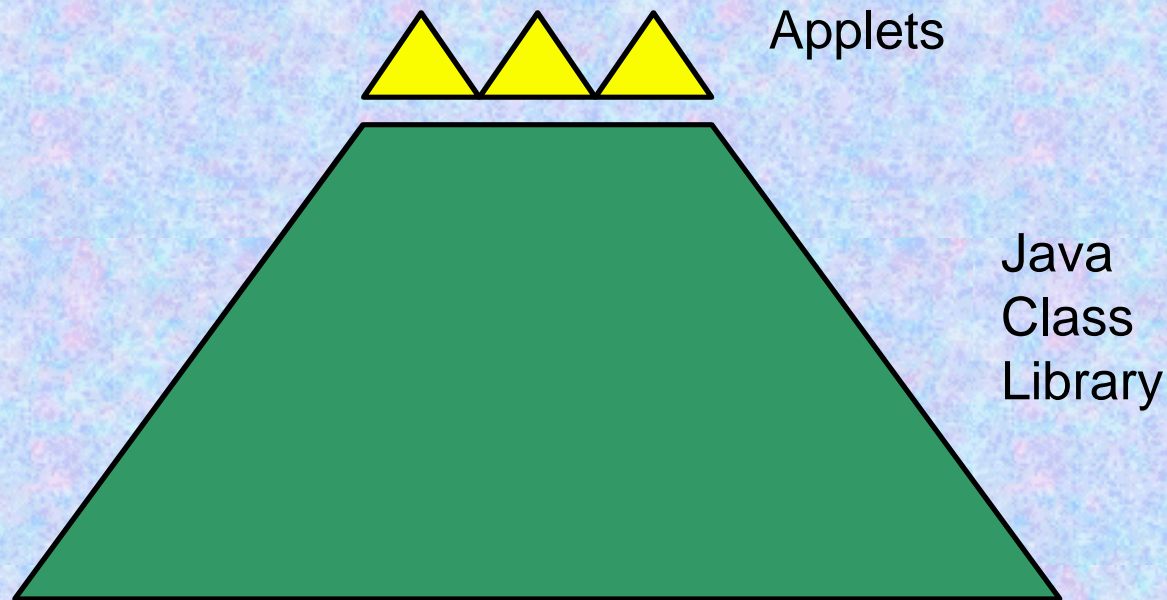
---

- ◆ definitely not in McIlroy's sense
- ◆ Applets (or even "Beans")
  - are user-level customizations of a very large application framework, the Java Class Library
- ◆ it is much easier to use code in the framework than to re-use somebody else's classes
  - that is why people are incorporating every conceivable feature into the Java Class Library

# Java Runtime Architecture

---

- ◆ mismatch between size of the framework vs. size of the user code is even more pronounced than in most other object-oriented systems



# Hierarchical Decomposition

---

- ◆ hierarchical design is “natural”:
  - divide a problem top-down into sub-problems until each is small enough to be solved directly
  - then combine the solutions bottom-up
- ◆ the various sub-systems that have been derived in this manner are self-sufficient and can therefore also be re-used in other contexts

# Structural Property

---

- ◆ when several small parts are combined “bottom-up” to form a larger part, the larger part “shadows” the smaller ones
  - one need no longer understand the smaller parts to use the larger part constructed out of them
  - one can also change the interfaces between the small parts without affecting the interface of the large part to the outside universe

# Hierarchical Decomposition vs. OOP

---

- ◆ most object-oriented systems do not provide this structural property of hierarchical decomposition
- ◆ inheritance usually prevents object-oriented design from creating tree structures

# Provocative Thesis

---

- ◆ inheritance-based object-oriented programming is a dead-end street
- ◆ the main reason why current object oriented systems are successful is because they are monolithic at heart
  - there is lots of re-use, but only a single *source* of re-use: the framework
  - unfortunately, we are approaching a complexity limit for the frameworks themselves

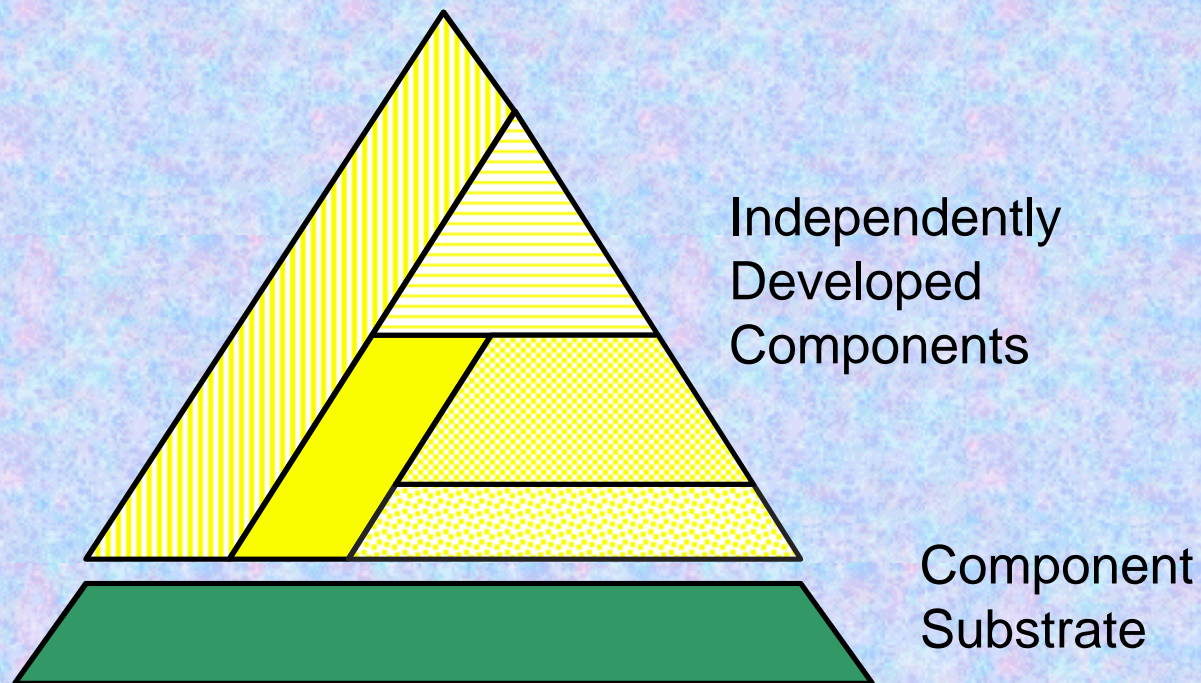
# Solution: “True” Components

---

- ◆ genuine division of concerns (and labor)
  - ◆ peer-to peer communication
  - ◆ substitutability, multi-sourcing
  - ◆ small shared common substrate
- 
- ◆ build systems by composing from independent sources rather than by customizing a monolithic framework

# Software Component Architecture

---



# Component Based Programming

---

- ◆ on the macroscopic level, this is nothing new
- ◆ much work on interoperability standards
- ◆ curiously, component interaction is often modeled as object oriented programming *without inheritance across component boundaries*
  - instead of invoking “super”, objects have to forward requests by explicit message passing

# Component-Based Programming

---

- ◆ we have been designing a system that duplicates this macroscopic model on a much finer scale
- ◆ key is an experimental programming language “Lagoona” with curious features:
  - “messages” are stand-alone entities and not subordinate to classes
  - there is no inheritance, but objects can forward received messages to other objects; even if they don’t “understand” them

# More on Lagoona

---

- ◆ *messages* define protocols of interaction
- ◆ *methods* define particular implementations of such a protocol
- ◆ *message set types* are used to define more complex interaction protocols
  - e.g.,  $S = \{A\} + \{B, C\}$
- ◆ structural compatibility based on subset relation
- ◆ subtyping and supertyping easily expressible

# More on Lagoon

---

- ◆ instead of defining abstract superclasses as done in OOP, “interface contracts” are specified by listing the minimal set of call-back messages that a passed object must understand

```
MODULE WindowManager;
```

```
  MESSAGE Redraw(); -- understood by displayed objects
```

```
  MESSAGE AddObject(...); -- understood by window
```

```
  TYPE Window = {..., AddObject, ...};
```

```
  ...
```

```
  METHOD (self: Window) AddObject(obj: {Redraw});
```

# More on Lagoona

---

- ◆ instead of inheritance, Lagoona provides generic message forwarding
- ◆ can be used to mimic class-based and prototype-object-based inheritance, but generally leads to simpler alternative architectures
- ◆ implementation challenge lies in making message-forwarding no more costly than a virtual function call

# Benefits

---

- ◆ a looser coupling between components
- ◆ structural uniformity: the fine-grained interaction between individual objects follows the same model as the interactions between whole components
- ◆ architectural simplifications, explicit control-flow

# Towards a Real-Time Lagoon

---

- ◆ component-oriented paradigm = functional requirements
- ◆ real-time = non-functional constraints
- ◆ timing constraints are a global property, not local to a component
- ◆ attaching deadlines to individual *methods* (=implementations) limits their reusability
- ◆ attaching deadlines to *messages* (=semantics) is somewhat better, but still not ideal

# Towards a Real-Time Lagoon

---

- ◆ distinguish between mandatory and optional message-sends
- ◆ an optional message send can be suppressed as a deadline approaches
- ◆ use this as a system structuring device, for example, in iterative computations, or when certain operations are non-essential, such as debugging and logging

# Summary and Conclusion

---

- ◆ software component architectures differ from conventional object-oriented systems
- ◆ they are currently modeled using object-oriented languages, but with limitations such as “no inheritance across component boundaries”
- ◆ our approach is the reverse: model component-messaging as a language primitive and use compiler technology to make this efficient even on a microscopic scale

# Summary and Conclusion

---

- ◆ some of Lagoona's mechanisms may be useful to support separation between essential and non-essential computations