
Stand-Alone Messages

A Step Towards Component-Oriented Programming Languages

Peter H. Fröhlich
phf@acm.org

Michael Franz
franz@uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Overview

- Introduction
- Interfaces and Implementations
- The Standard Approach
- Interface Conflicts
- Avoiding Interface Conflicts
- Stand-Alone Messages
- Further Applications
- Comparison
- Summary

Introduction

- Goal of **component-oriented programming**:
 - Replace monolithic software systems by **compositions** of
 - * reusable **software components** (extensions)
 - * hierarchical **component frameworks** (environments)
- **Extensibility** of component-oriented software systems:
 - **Dynamic** (the **time** of composition is open)
 - **Independent** (the **vendor** of a component is open)
- **Challenge**: Design **programming languages** supporting dynamic **and** independent extensibility

Interfaces and Implementations

- **Interfaces:** Sets of **abstract** operations called **messages**
 - **What** effect is to be achieved.
- **Implementations:** Sets of **concrete** operations called **methods**
 - **How** the effect is to be achieved.
- Characterize programming languages as follows:
 - **Modular:** message/method = procedure header/body bound to some **non-extensible module**. (Modula-2, Oberon)
 - **Object-oriented:** message/method = “procedure” header/body bound to some **extensible type**. (Sather, Eiffel)

The Standard Approach

- Component-oriented language = **combination** of object-oriented and modular features
 - dynamic extensibility \Rightarrow **inclusion polymorphism**
 - independent extensibility \Rightarrow **sealed modules**
- Languages that offer **some** form of this combination:
 - Oberon-2, Modula-3, Component Pascal, Java, *LOOM*, Moby.
- Imagine an **idealized** language \mathcal{IJ} similar to Java except that we
 - replace (closed) packages with (sealed) modules
 - separate subtyping from subclassing completely

Interface Conflicts

```
module edu.uci.original {  
  public interface Stack {  
    void push (Object o);  
    void pop ();  
    Object top ();  
    boolean empty ();}}}
```

```
module gov.nsa.compatible {  
  public interface Stack {  
    void push (Object o);  
    void pop ();  
    Object top ();  
    int size ();}}
```

⇒ Independent extensibility **not** guaranteed!

```
module com.sun.syntactic {  
  public interface Stack {  
    void push (Object o);  
    Object pop ();  
    boolean empty ();}}}
```

```
module org.cthulhu.semantic {  
  public interface Stack {  
    void push (Object o);  
    void pop ();  
    Object top ();  
    boolean empty ();  
    int size ();}}
```

Avoiding Interface Conflicts

- **Programming conventions** (message names) and **design patterns** (Adapter, Command): Hard to **enforce**
- **Overloading** (Java): No **general** solution
- **Renaming** (Eiffel): Works but **clutters** namespace
- **Explicit qualification** (C++): Can work but **verbose**
- All these approaches
 - increase the **complexity** of either language or software system
 - fix the problem **after the fact** instead of avoiding it by design

Stand-Alone Messages (1)

- \mathcal{IJ} binds **messages and methods** to **extensible types**:
 - Methods: Enables **inclusion polymorphism**
 - Messages: Leads to **interface conflicts**
- **Stand-alone messages**: Bind **messages** to **non-extensible modules** instead!
 - Messages have to be **fully qualified**
 - Interface types become **sets of messages**
- **Any** combination of interface types
 - results in an interface type \Rightarrow **no syntactic conflicts**
 - preserves all constituent messages \Rightarrow **no semantic conflicts**

Stand-Alone Messages (2)

```
module edu.uci.original {  
  public message void push (Object o);  
  public message void pop ();  
  public message Object top ();  
  public message boolean empty ();  
  public interface Stack { push, pop, top, empty }}}
```

```
module com.factorial.cool.extension {  
  import f1 = edu.uci.original;  
  public class CoolStack implements f1.Stack {...  
    public void f1.push (Object o) { ... }  
    public void f1.pop () { ... }  
    public Object f1.top () { ... }  
    public boolean f1.empty () { ... }}}
```

Further Applications

- **Safe structural conformance:**
 - Messages fully qualified \Rightarrow No **accidental** conformance
 - In some form **required** for component-oriented programming

- **Minimal signatures:**
 - Structural conformance \Rightarrow **Anonymous** interface types
 - Type references **minimally** to contain only relevant messages
 - Solves certain component **re-entrance** problems

Comparison

- **Renaming:**
 - **Original** meaning of message hard to see in the interface
 - **Anonymous** interfaces require rather **clumsy** syntax
- **Explicit qualification:**
 - Support **dynamic binding** and force qualification **everywhere**
 - Type names **and module names** have to be included.

Design Space	Message \in Type	Message \in Module
• Method \in Type	Object-Oriented	<i>Component-Oriented</i>
Method \in Module	Useless?	Modular

Summary

- Component-oriented programming languages must support **dynamic** and **independent** extensibility
- In current languages supporting dynamic extensibility, independent extensibility is **not guaranteed**:
 - Messages bound to **extensible types** (like methods)
 - Interface conflicts require **additional** constructs
- Stand-alone messages **solve** this problem:
 - Messages bound to **non-extensible modules**
 - Interface conflicts **ruled out by design**
 - Further applications to component-oriented programming