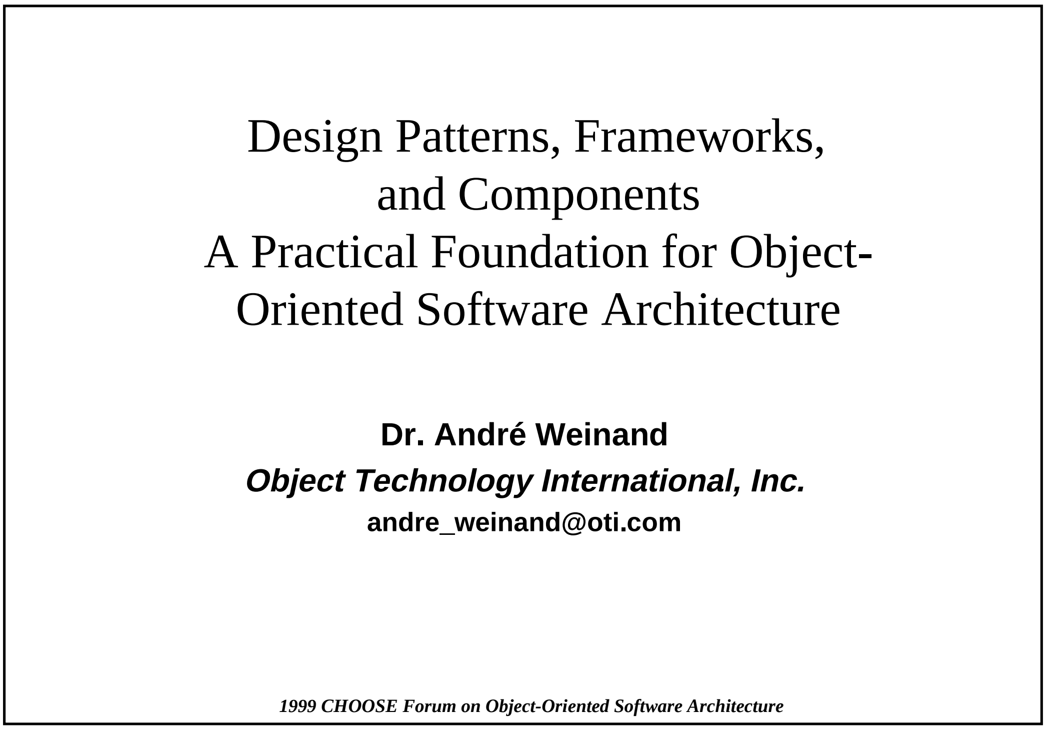
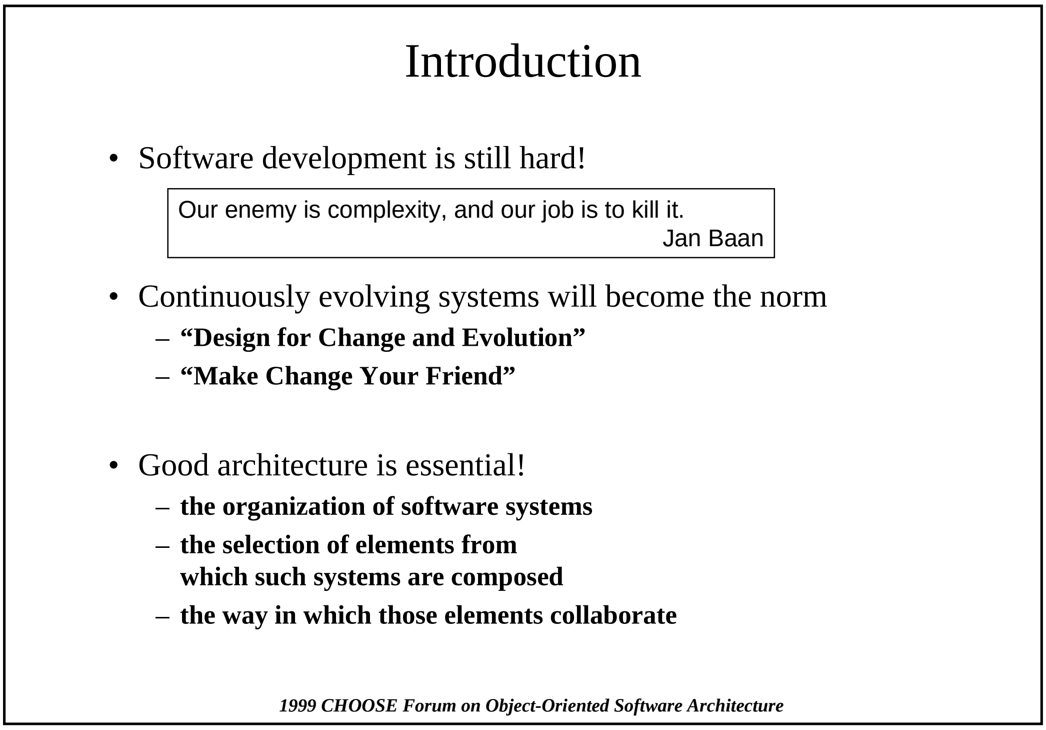


Design Patterns, Frameworks, and Components A Practical Foundation for Object-Oriented Software Architecture

Dr. André Weinand
Object Technology International, Inc.
andre_weinand@oti.com



Introduction

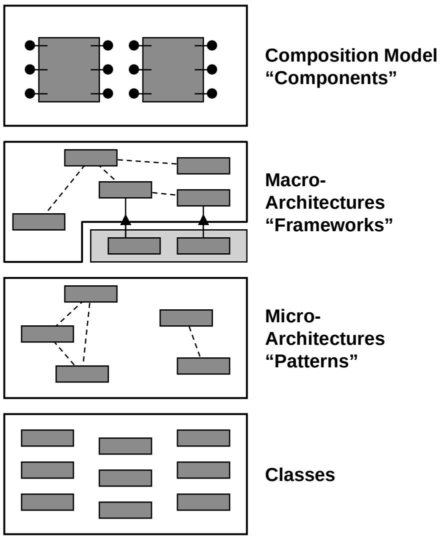
- Software development is still hard!

> Our enemy is complexity, and our job is to kill it.
> Jan Baan

- Continuously evolving systems will become the norm
  - **"Design for Change and Evolution"**
  - **"Make Change Your Friend"**
- Good architecture is essential!
  - **the organization of software systems**
  - **the selection of elements from which such systems are composed**
  - **the way in which those elements collaborate**

Elements of Object Technology

- OT provides a spectrum of elements to craft a system



1999 CHOOSE Forum on Object-Oriented Software Architecture

Overview

- Practical introduction to key concepts of OT
  - **design insights into a *Browser Framework***
- Showing a typical architectural evolution:
  - **prototype/sample application**
    ⇒ white-box framework
    ⇒ black-box framework
    ⇒ componentized system
- Combining different OO techniques
  - **there is no “one size fits all”**

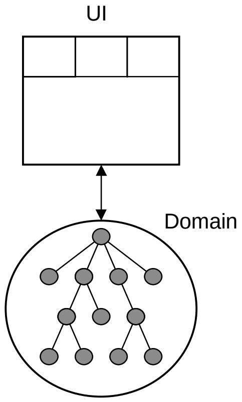
1999 CHOOSE Forum on Object-Oriented Software Architecture

Background

- Comprehensive OO Systems
  - **ET++**
  - **Taligent's CommonPoint**
  - **framework for dynamic web pages (IFA WebDisplay)**
  - **ultralight client infrastructure (OTI ULC)**
- Played multiple roles
  - **framework architect, implementor, client**
  - **technical support**
  - **mentor**
  - **teacher**

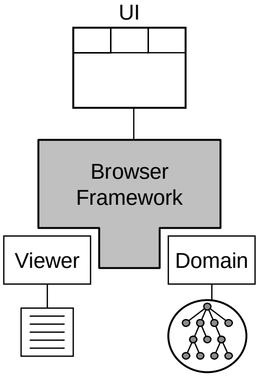
The Problem

- Exploring and manipulating hierarchically structured *Domains*
  - **navigating relationships**
  - **viewing/editing of a node's contents**
- Examples:
  - **file systems**
  - **mail**
  - **Web**
  - **program representation**



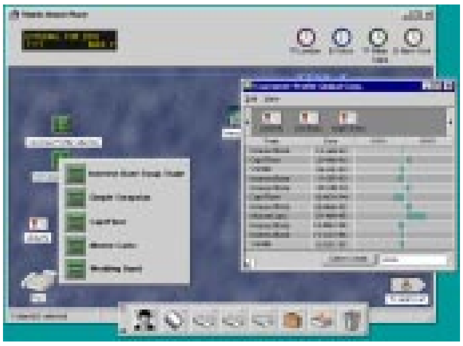
The Goal

- A framework that…
  - **defines the browsing metaphor**
    - generically implements all the “complex stuff”
  - **allows clients to focus on**
    - domain definition
    - node content editors/viewers
  - **is simple!**
    - small number of concepts



Origins of Browser Framework

- Taligent Hoops/cpProfessional
  - **C++ IDE**
  - **Components & Properties**
- Taligent Workspace
  - **“People, Places & Things”**
  - **InfoNodes & Viewers**



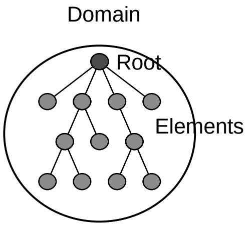
Domain Access

- ***Elements***
  - **browseable entities**
  - **data nodes in the domain**
  - **examples: a file, a mailbox**
- Elements have ***Properties***
  - ***aspects* of the browsable entities**
  - **examples: mails in a mailbox, the file's contents**
- Elements provide a *dynamic data access* API
- Property kinds
  - **simple: Object, Boolean, String, Element**
  - **indexed: ordered set of Elements**

```
IProperty getProperty(String)
void setProperty(String, IProperty)
```

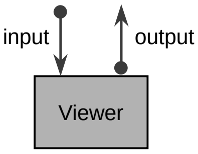
Domain Model

- knows a root element
  - **the "portal" into a domain**
- is the model in the Model/View architecture
  - **notifier for domain changes**
  - **elements fire domain changes via model**
    - ⇒ elements know their domain model
  - **notification specifies changed property**
  - **observers register with domain model**



Viewer

- A Viewer …
  - **is fed with input element**
  - **presents properties of its input element**
  - **creates widget hierarchy**
  - **observes domain model for changes**
  - **handles user interactions**
  - **sends out selection change events**

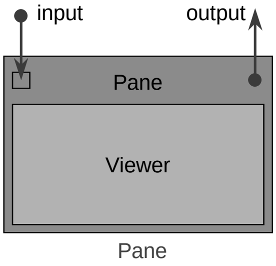


- Standard Viewers exists
  - **Structure oriented Viewers**
    - Tree, List, Table
  - **Content oriented Viewer**
    - Text

1999 CHOOSE Forum on Object-Oriented Software Architecture

Pane - a Viewer’s Container

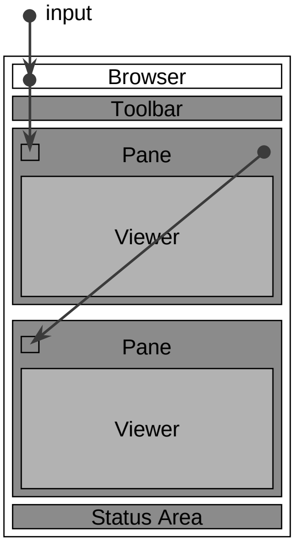
- installs Viewer dynamically based on its input
- adds more controls
- optionally provides UI to pick other viewers for the viewed property
- tracks viewer selection changes



1999 CHOOSE Forum on Object-Oriented Software Architecture

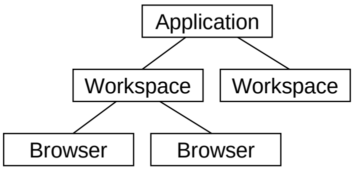
Browser - Pane's Container

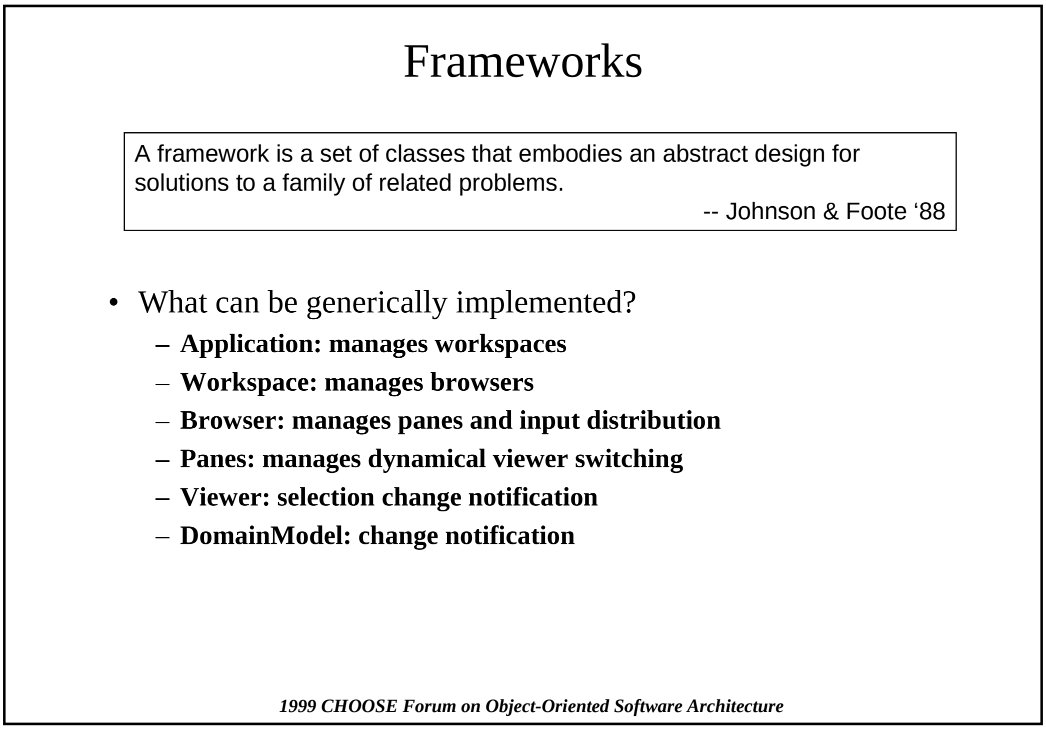
- implements browsing metaphor
- is fed with an Element
- manages panes
- defines wiring between panes
- defines layout between panes
- adds more controls



Overall Presentation Architecture

- Hierarchical system of supervisors
- Pattern:
  - **Chain of responsibility**



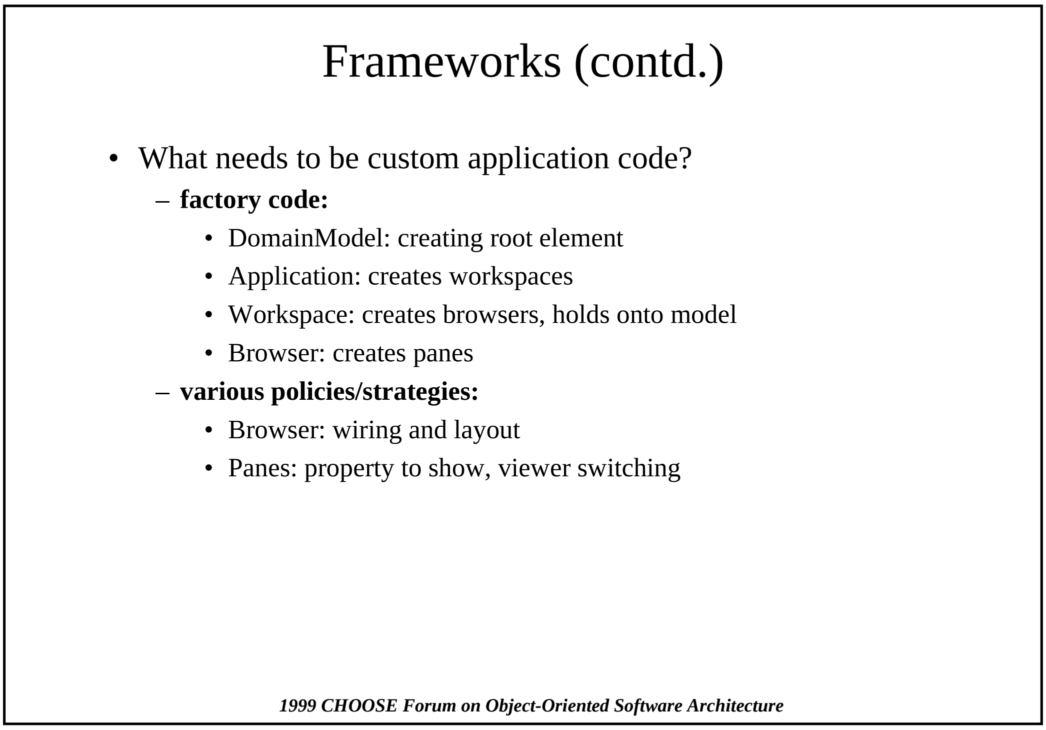


Frameworks

> A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.
>
> -- Johnson & Foote ‘88

- What can be generically implemented?
  - **Application: manages workspaces**
  - **Workspace: manages browsers**
  - **Browser: manages panes and input distribution**
  - **Panes: manages dynamical viewer switching**
  - **Viewer: selection change notification**
  - **DomainModel: change notification**

1999 CHOOSE Forum on Object-Oriented Software Architecture



Frameworks (contd.)

- What needs to be custom application code?
  - **factory code:**
    - DomainModel: creating root element
    - Application: creates workspaces
    - Workspace: creates browsers, holds onto model
    - Browser: creates panes
  - **various policies/strategies:**
    - Browser: wiring and layout
    - Panes: property to show, viewer switching

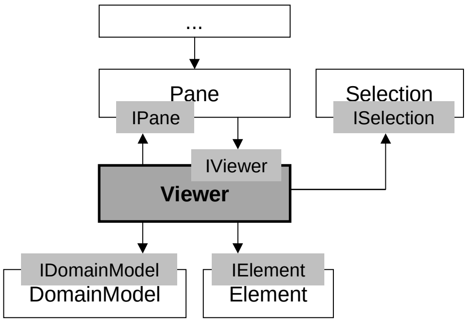
1999 CHOOSE Forum on Object-Oriented Software Architecture

Defining the Framework

- Separation of design from code
  - **define the “design” as Java interfaces in one package**
  - **move “implementation details” into a separate package**
- Motivation
  - **encapsulate volatile implementation details behind stable interfaces**
    - make the difference explicit for clients
    - convince clients to use interfaces but avoiding the implementations
      - clients are very creative in taking advantage of every implementation detail
  - **clients shouldn’t be forced into implementation inheritance!**
    - less flexible

Discovering the Viewer Interface

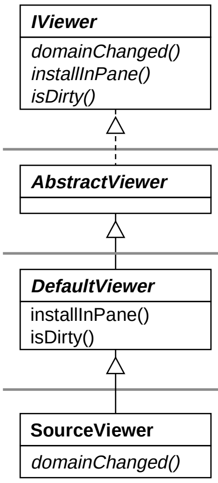
- Interfaces describe interactions between the Viewer and the rest of the system
- Another example:
  - **Elements**
  - **Properties**
  - **DomainModels**



Problems with Interfaces

- Interfaces cannot have default implementation
  - **cumbersome for clients to implement**
  - ⇒ **Provide default implementations in a separate layer**
    - difference between design (interfaces) and implementation remains explicit!
- Solves another Problem:
  - **if clients derive directly from an interface**
    - every interface change is a breaking change!
  - ⇒ **introduce an abstract class as an insulation layer on top of interfaces**
    - if interface has to be changed, provide compatibility implementations there

Example: Viewers Layering

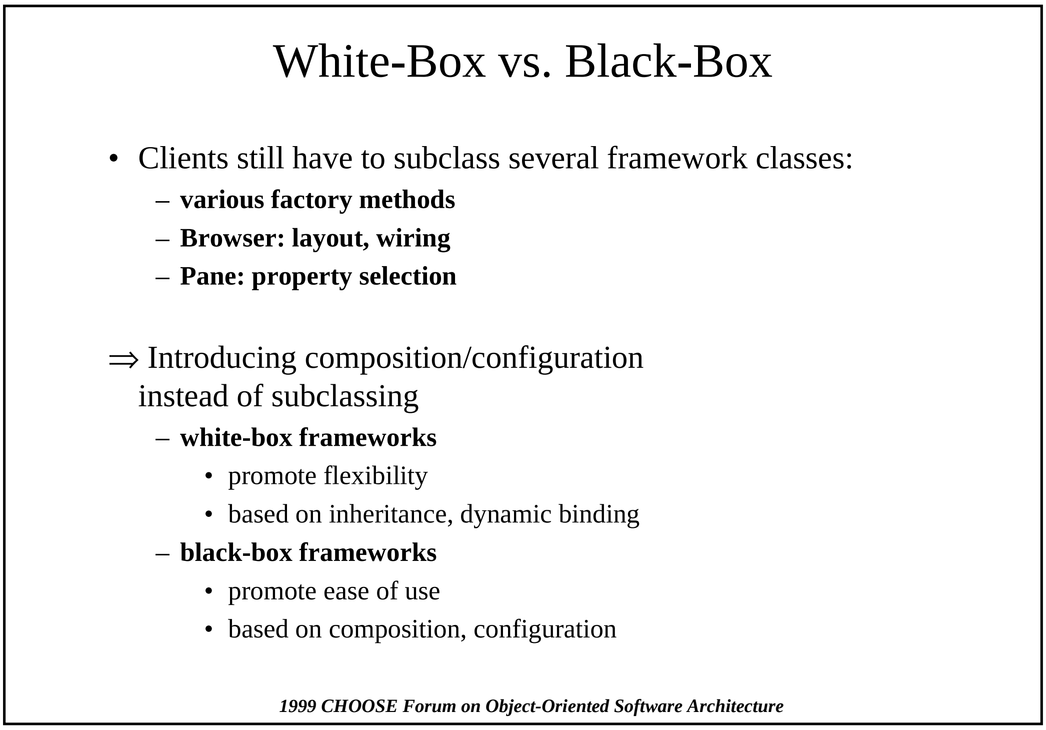


Interfaces
- reuse of design
- stable
- specifies public interfaces

Abstract Implementation
- insulation between interface and client
- initially no implementation

Default Implementation
- reuse of design and code
- less stable
- specifies protected interfaces

Client Implementation
- reuse of implementation
- least stable

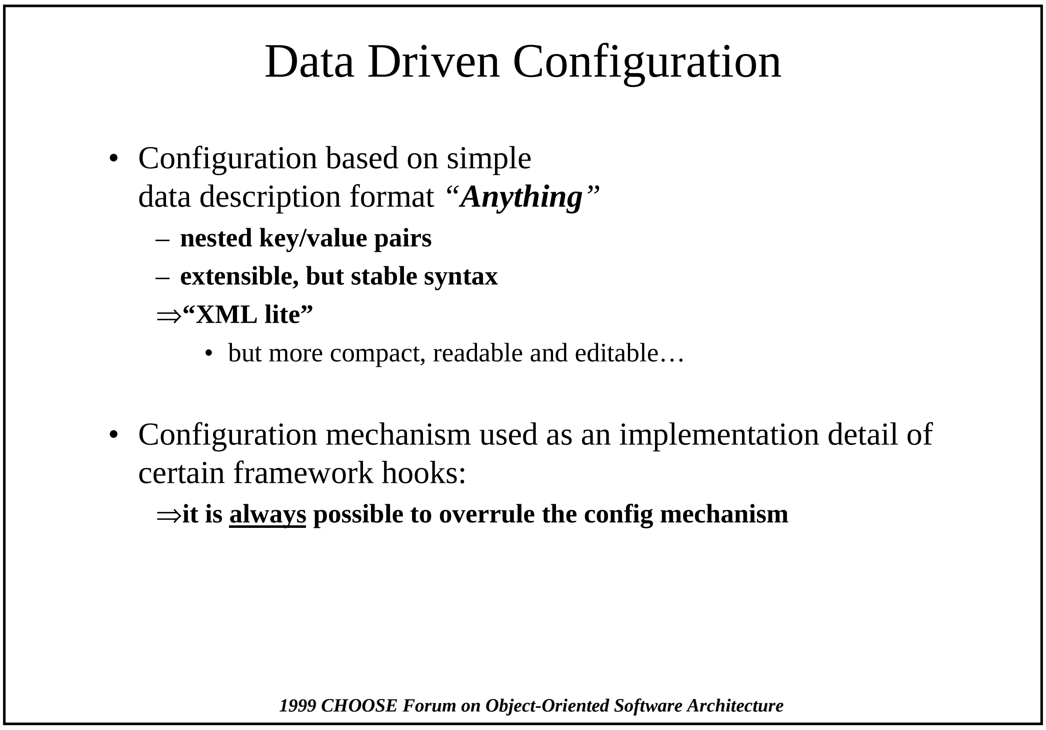


White-Box vs. Black-Box

- Clients still have to subclass several framework classes:
  - **various factory methods**
  - **Browser: layout, wiring**
  - **Pane: property selection**

⇒ Introducing composition/configuration instead of subclassing

- **white-box frameworks**
  - promote flexibility
  - based on inheritance, dynamic binding
- **black-box frameworks**
  - promote ease of use
  - based on composition, configuration



Data Driven Configuration

- Configuration based on simple data description format *“**Anything**”*
  - **nested key/value pairs**
  - **extensible, but stable syntax**

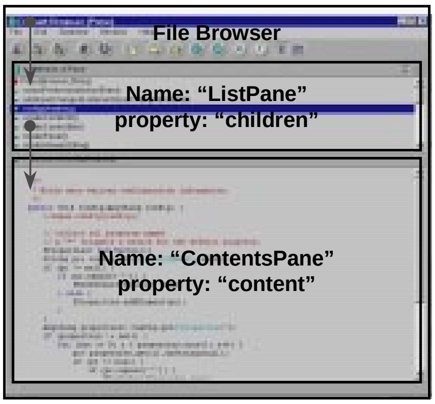
  ⇒“XML lite”
  - but more compact, readable and editable…

- Configuration mechanism used as an implementation detail of certain framework hooks:

  ⇒it is always possible to overrule the config mechanism

Example: Browser Definition

```
/title “File Browser”          # name of browser
/outputs { “ListPane” }        # forward input to
/panes {                       # Pane definitions
   /ListPane {
      /properties { “children”}
      /outputs { “SourcePane” }
   }
   /ContentsPane {
      /properties { “contents” }
   }
}
/layout {                      # layout for Panes
   /type “vsplit”              # vertical layout
   /members {
      { /type “pane” /name “List” /weight 100 }
      { /type “pane” /name “Source” /weight 200 }
   }
}
```



Communication Issues

- Closing the framework makes communication harder
- Example: Viewers
  - **Viewers are unaware of each other**
  - **one custom viewer wants to talk to another custom viewer**
    - e.g. ListViewer with search results wants to select text in TextViewer
- Framework has to support unanticipated interactions

⇒WireCommands

- **custom viewer sends custom WireCommand**
- **framework distributes them along the wiring against viewer targets**
- **dispatch method checks whether target is acceptable**

Componentizing the Framework

- Component Definition:

> A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.
>
> -- Grady Booch

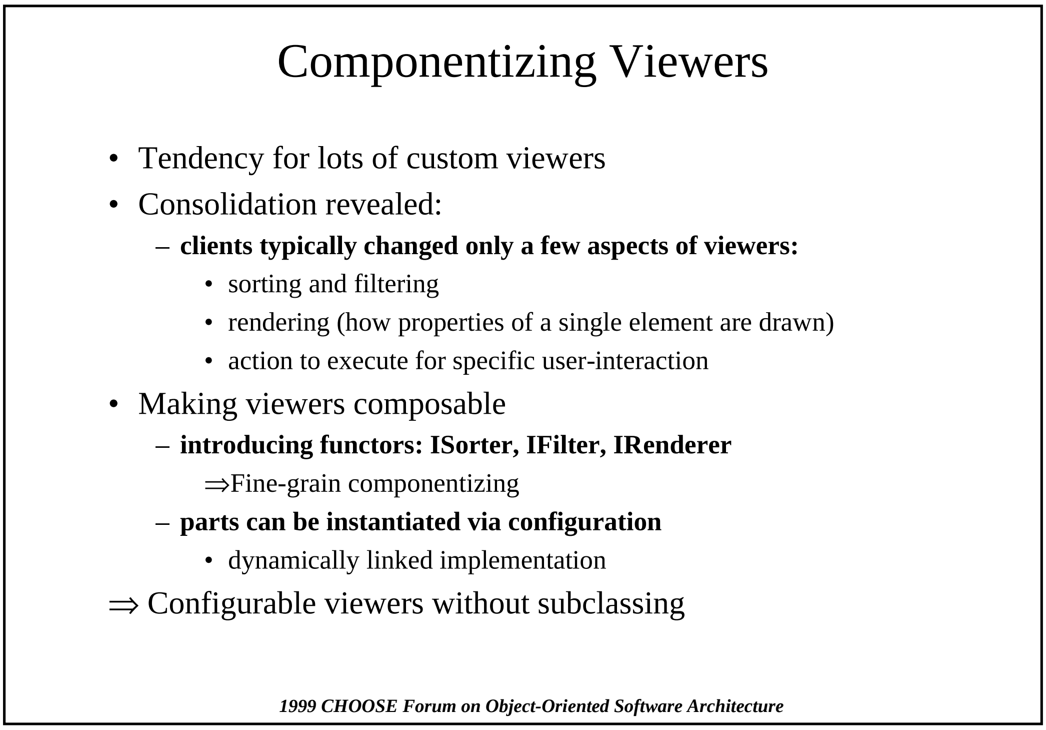
- Components can be simple:
  - **no need for standardization or “marketplace”**
  - **just application-specific core business assets**
- Examples:
  - **Viewers**
  - **DomainModels**

1999 CHOOSE Forum on Object-Oriented Software Architecture

Componentizing UI Handlers

- Actions
  - **based on Swing Action**
  - **specifies the action to be executed on a dynamic target**
  - **arguments are:**
    - current selection
    - current Viewer, Browser under focus
- Actions can be installed in different contexts:
  - **Pane control bar**
  - **Browser menubar**
  - **Browser toolbar**
- Actions define properties for different UIs
  - **enable/disable state**
  - **icon, label, tooltips**

1999 CHOOSE Forum on Object-Oriented Software Architecture



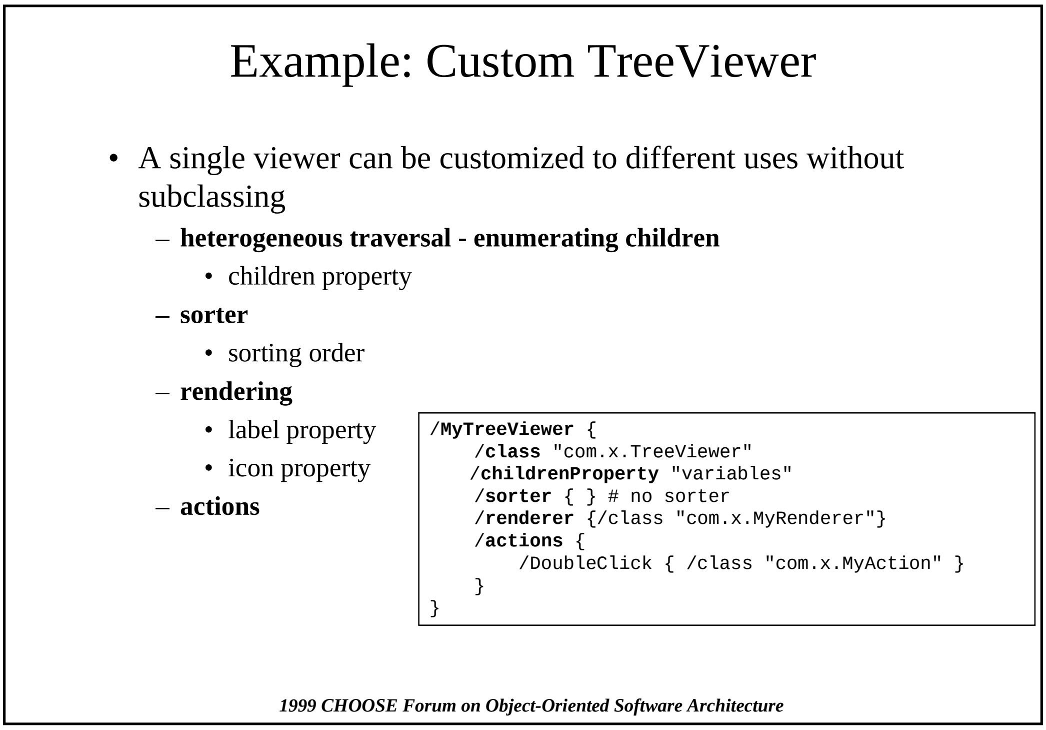
Componentizing Viewers

- Tendency for lots of custom viewers
- Consolidation revealed:
  - **clients typically changed only a few aspects of viewers:**
    - sorting and filtering
    - rendering (how properties of a single element are drawn)
    - action to execute for specific user-interaction
- Making viewers composable
  - **introducing functors: ISorter, IFilter, IRenderer**

    ⇒Fine-grain componentizing
  - **parts can be instantiated via configuration**
    - dynamically linked implementation

⇒ Configurable viewers without subclassing

1999 CHOOSE Forum on Object-Oriented Software Architecture



Example: Custom TreeViewer

- A single viewer can be customized to different uses without subclassing
  - **heterogeneous traversal - enumerating children**
    - children property
  - **sorter**
    - sorting order
  - **rendering**
    - label property
    - icon property
  - **actions**

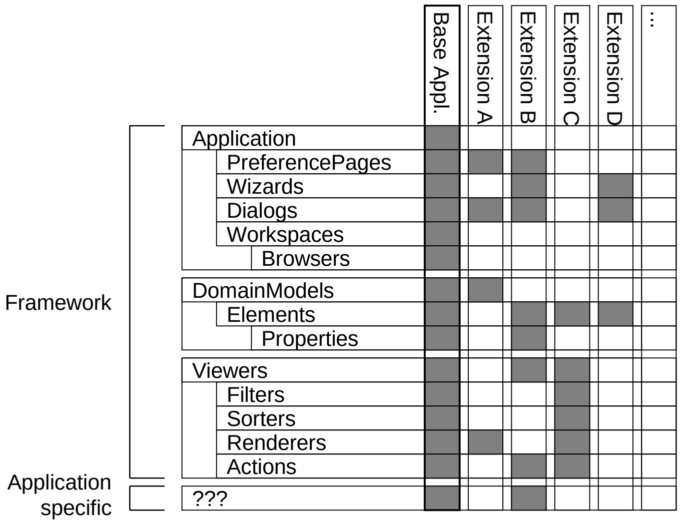
```
/MyTreeViewer {
    /class "com.x.TreeViewer"
   /childrenProperty "variables"
    /sorter { } # no sorter
    /renderer {/class "com.x.MyRenderer"}
    /actions {
        /DoubleClick { /class "com.x.MyAction" }
    }
}
```

1999 CHOOSE Forum on Object-Oriented Software Architecture

Beyond Fine Grained Components

- Browser framework provides fine grain extensibility
  - **Viewers, Sorters, Filters, Renderers, Actions**
  - **DomainModels, Elements, Properties**
- Typical applications have additional requirements
  - **grouping components**
    - an application extension is more than a single component
  - **more flexibility for extensions**
    - new elements for existing models
    - new properties for existing elements
    - application specific extensibility

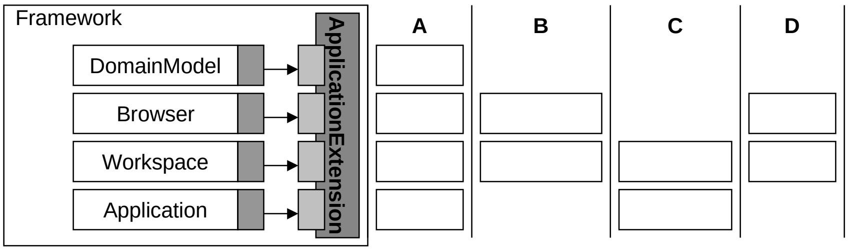
Example: Application Extensibility



		Base Appl.	Extension A	Extension B	Extension C	Extension D	...
Framework	Application	■					
	PreferencePages	■	■	■			
	Wizards	■		■		■	
	Dialogs	■	■	■		■	
	Workspaces	■					
	Browsers	■					
	DomainModels	■	■				
	Elements	■		■	■	■	
	Properties	■		■			
	Viewers	■		■	■		
	Filters	■			■		
	Sorters	■			■		
	Renderers	■	■		■		
	Actions	■		■	■		
Application specific	???	■		■			

Discovering Initial Extension Support

- Setting up a “vertical” project structure
- Moving components to it
- Decoupling via interfaces
  - **IUIExtension, IModelExtension, etc.**
- Adding extension framework code
- Introducing the *ApplicationExtension*



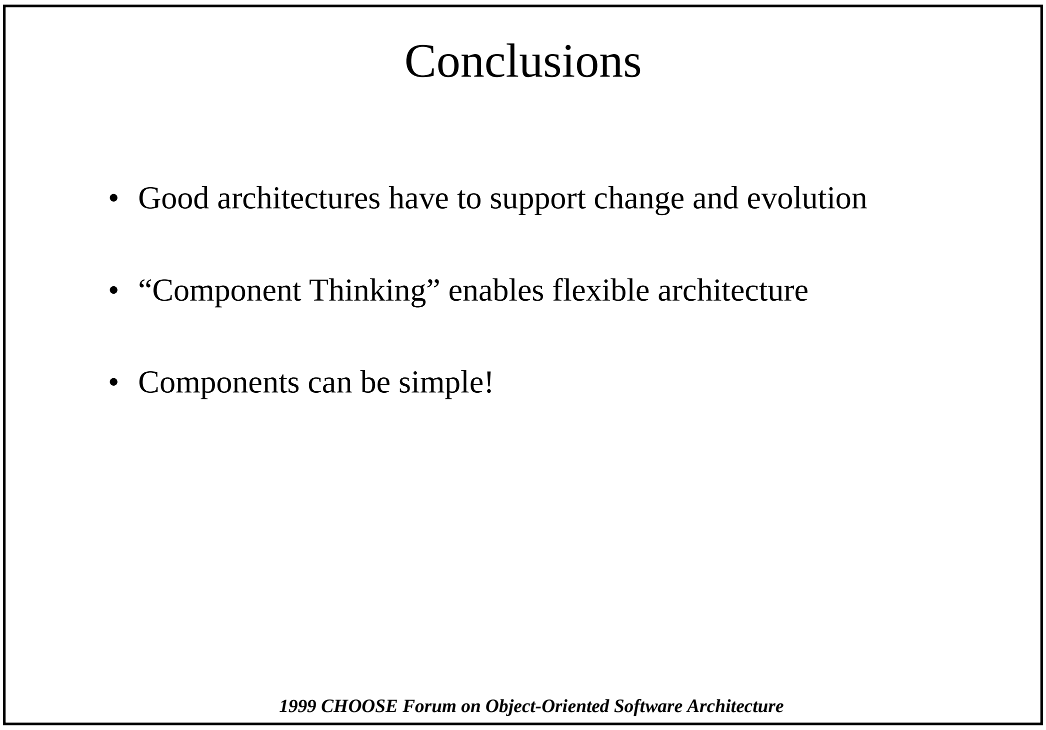
1999 CHOOSE Forum on Object-Oriented Software Architecture

Defining Extension Support

- *ApplicationExtension* defines
  - **getters for different aspects of a typical extension**
    - return new object
    - or return self (*this*)
  - **a “root” or “base” for all resource requirements**

- Adding more extensibility to framework, e.g.:
  - **extending DomainModels with new Elements**
  - **adding factory objects for Browsers and Viewers**
    - e.g. factory object includes resource base

1999 CHOOSE Forum on Object-Oriented Software Architecture



Conclusions

- Good architectures have to support change and evolution
- “Component Thinking” enables flexible architecture
- Components can be simple!