

# Design-Level Software Composition

Rudolf K. Keller and Reinhard Schauer  
Université de Montréal

(1) Introduction

(2) SPOOL Environment

(3) Pattern-Based Recovery of Design Components

(4) Pattern-Based Composition of Design Components

(5) Conclusion

**SPOOL**

Spreading Desirable Properties  
into the Design of  
Object-Oriented, Large-Scale  
Software Systems



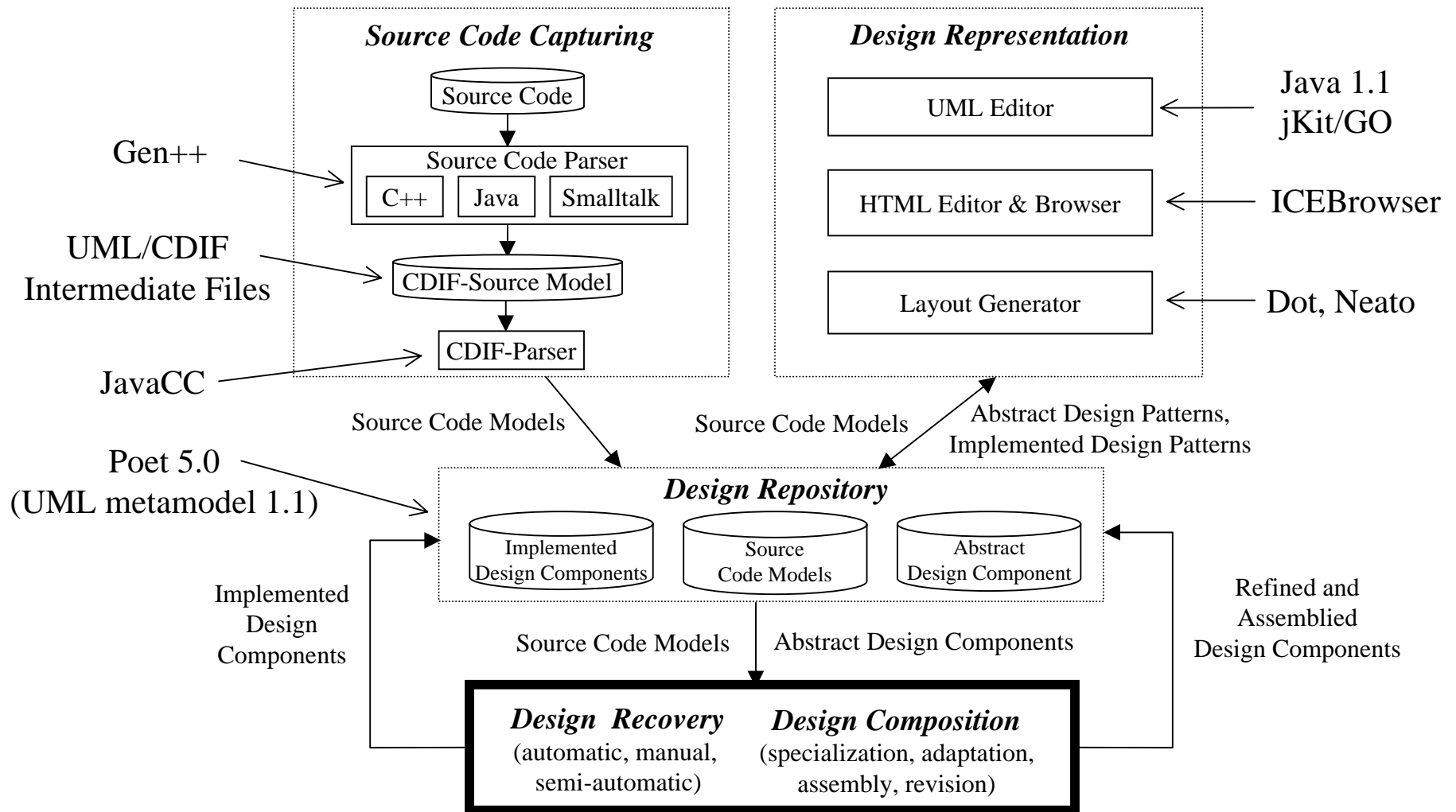
# Component Questions

However appealing and intuitive the notion of software components may be, there are some fundamental questions that are not always easy to answer... (Oscar Nierstrasz, IW-LSSC'98)

- Is a component a *class* or an *instance*?
- Are components *bigger* or *smaller* than objects?
- What information should go in a component's *interface*?
- What kinds of interfaces are good for both *usability* and *adaptability*?
- Should components be *statically* or *dynamically* configurable? (or both?)
- What is the right way to put together *distributed components*?
- How can we combine components that adhere to *different policies*?
- How can we *refactor* object-based applications into components?
- How can we gracefully *upgrade versions* of components?

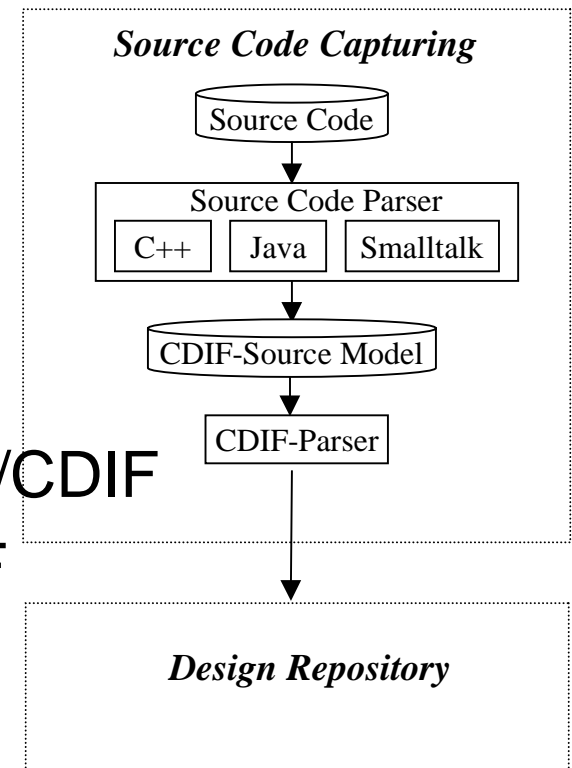


# SPOOL: Tool Architecture



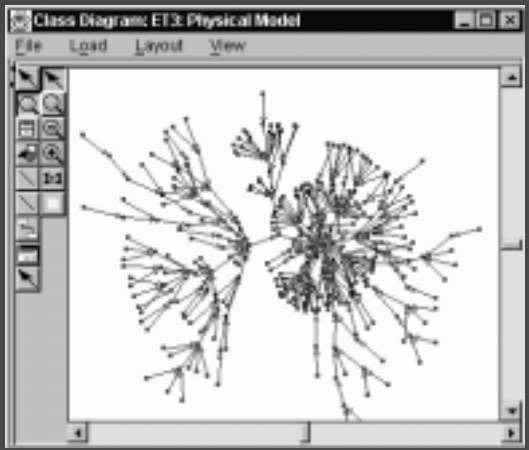
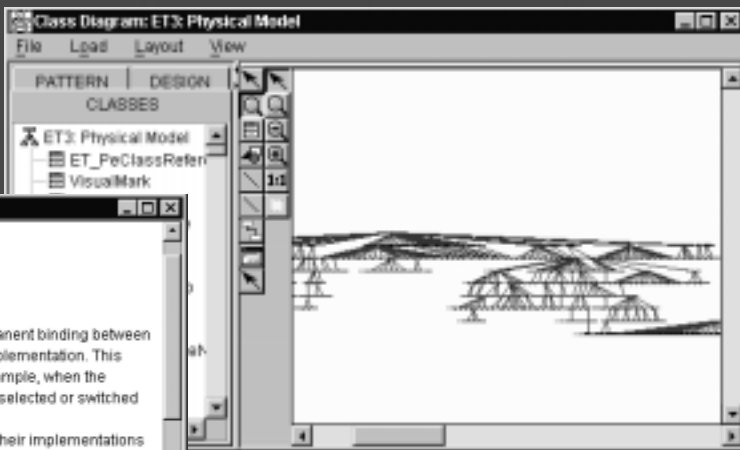
# Source Code Capturing

- GEN++ Source Code Parsing
  - AT&T Bell Labs (Prem Devanbu)
  - Source Code Analysis System
  - bound to the AT&T Cfront Compiler
  - traverses the abstract semantics graph
- GEN++ Source Code Exporter for UML/CDIF
- JavaCC Importer for parsing UML/CDIF
  - Visitor on JavaCC generated AST
  - spools the source code into the repository
- Repository based on UML 1.1 Metamodel
  - Core Backbone, Core Relationships, Common Behavior Actions, Model Management



SYSTEMS | PATTERN

- Systems
  - ET3
    - ET3: Physical Model
      - .LOOKSImages\StretchBox.C
      - .Foundation\Rectangle.C
      - .CONTAINER\SeqCell.C
      - .Images\CheckmarkOff\ima
      - .ColorPicker.C



ICE Browser - [Intent]

Detected Pattern

- Bridge
  - Intent
  - Known As
  - Motivation
  - Applicability
  - Participants
  - Collaborations
  - Consequence
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns
  - Structure
  - ConcreteImple
  - RefinedAbstra

### Applicability

Use the *Bridge* pattern when

- you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- both the abstraction and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be

Class Diagram: ET3: Physical Model

Load Layout View

PATTERN | DESIGN

Pattern Languages

- Template Method
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton
- Adapter
- Bridge
- Decorator
- Composite
- Facade
- Flyweight
- Proxy
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State

Window Bridge

Detected Pattern

- Bridge
  - Intent
  - Known As
  - Motivation
  - Applicability
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns
  - Structure
  - ConcreteImple
  - RefinedAbstrac
  - Implementor
  - Abstraction
  - ConcreteImple

Abstract Pattern: Bridge

Detected Pattern: Window Bridge

Properties

- Disconnected Classes
- Classname
- Attributes
- Methods
- Generalizations
- Aggregations
- Associations
- Instantiations

Ok Cancel



Project: System3

File

PROJECTS | PATTERN

- Projects
  - System3
    - Models
      - System3
        - System3
      - Pattern Languages

Class Diagram: System3

File Load Layout View

CLASSES | PATTERN | DESIGN

System3

Class Diagram: System3

File Load Layout View

CLASSES | PATTERN | DESIGN

System3

Class Diagram: System3

File Load Layout View

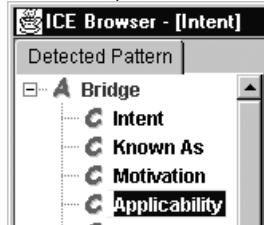
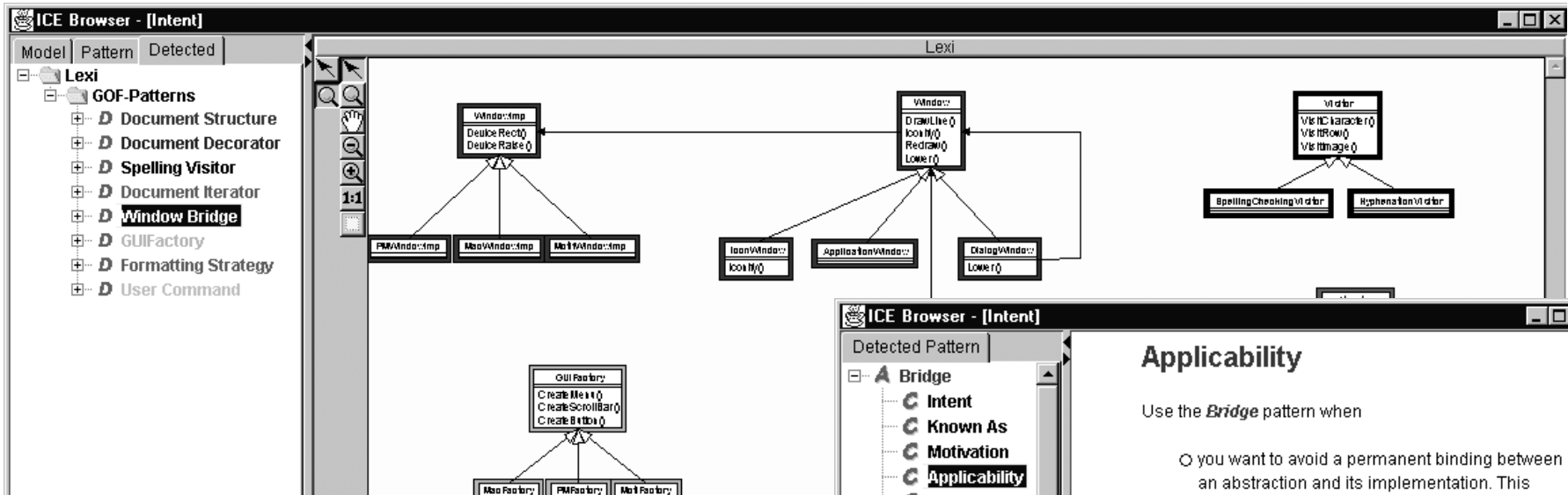
Class Diagram: System3

File Load Layout View

Properties

- Disconnected Classes
- Classname
- Attributes
- Methods
- Generalizations
- Aggregations
- Associations
- Instantiations

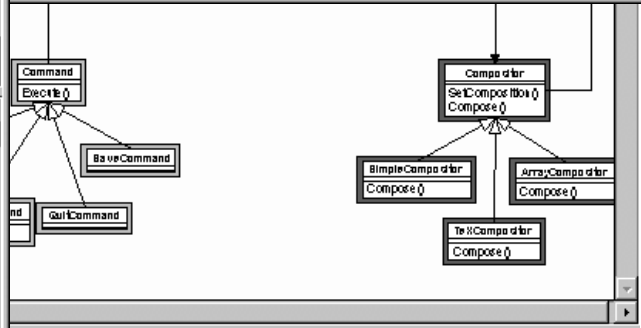
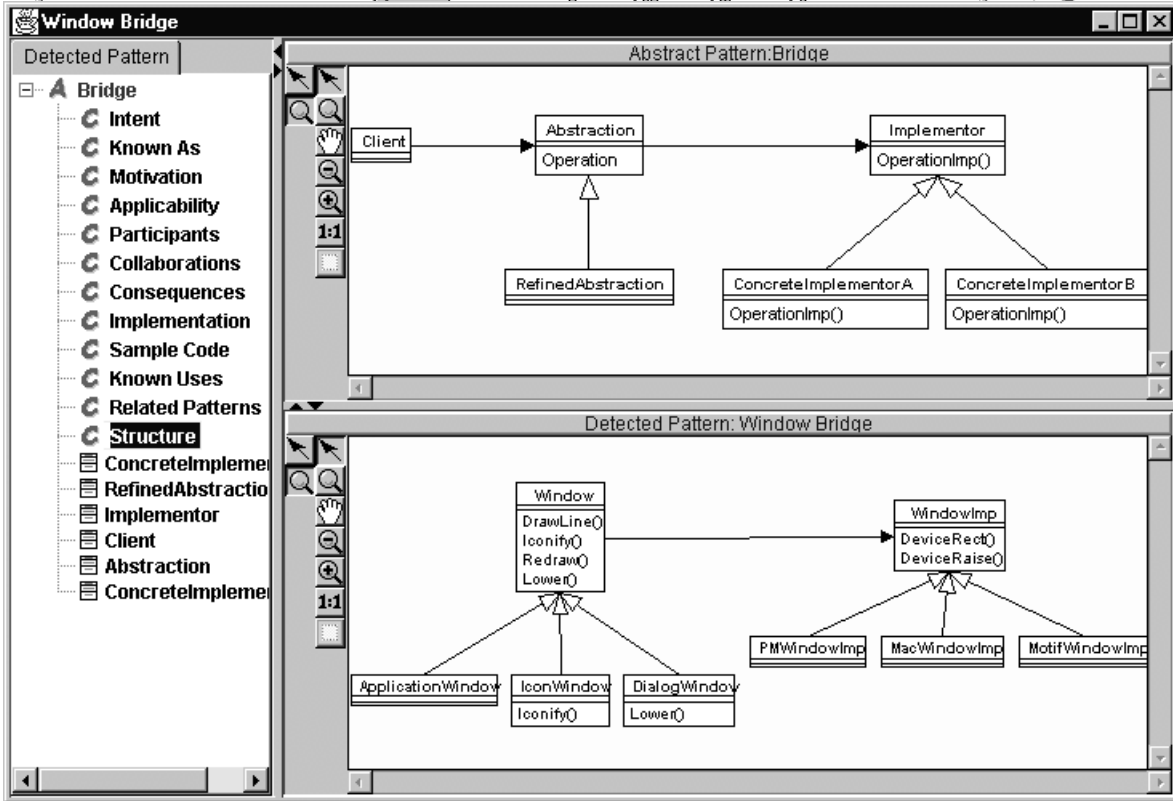
Ok Cancel



### Applicability

Use the *Bridge* pattern when

- you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- both the abstraction and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.



Workshop



# Design Repository

- Information content
  - Files (names, directory)
  - Classifiers (classes, structures, unions, primitive types)
  - Generalization hierarchy
  - Attributes (name, type, owner, visibility)
  - Operations and methods (name, type, owner, polymorphic, kind)
  - Parameters (name, type)
  - Return type (name, type)
  - Call actions (operation, receiver)
  - Create actions (classifier)
  - Variable use
  - Friendship relationships
  - Class and function template instantiations



# Visualization

- Architectural Review of the Design Visualization System
  - diagram manager (View Handler pattern)
    - internal frames for UML packages
    - synchronization of diagrams
  - extensibility for additional services, such as undo, tracing, logging, etc. (Command Processor pattern)
  - configurable visualization strategies (Strategy pattern)
- Integration of “Dot” and “Neato”
  - DOT - hierarchical arrangement of graphic objects
  - NEATO - symmetric arrangement of graphic objects

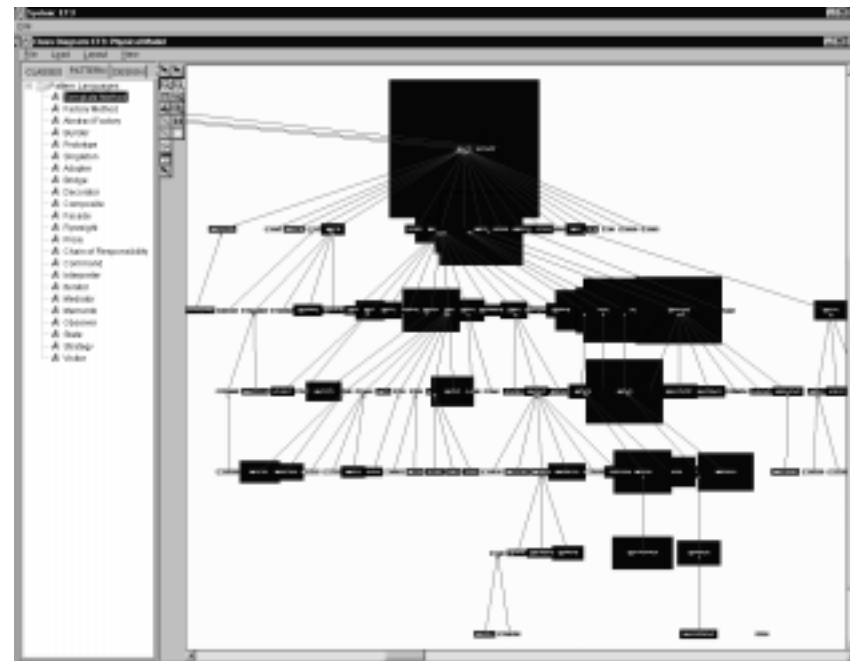
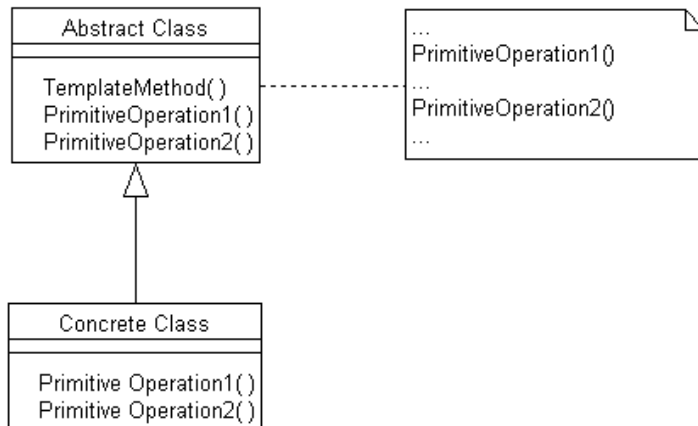


# Reverse Engineering - Background

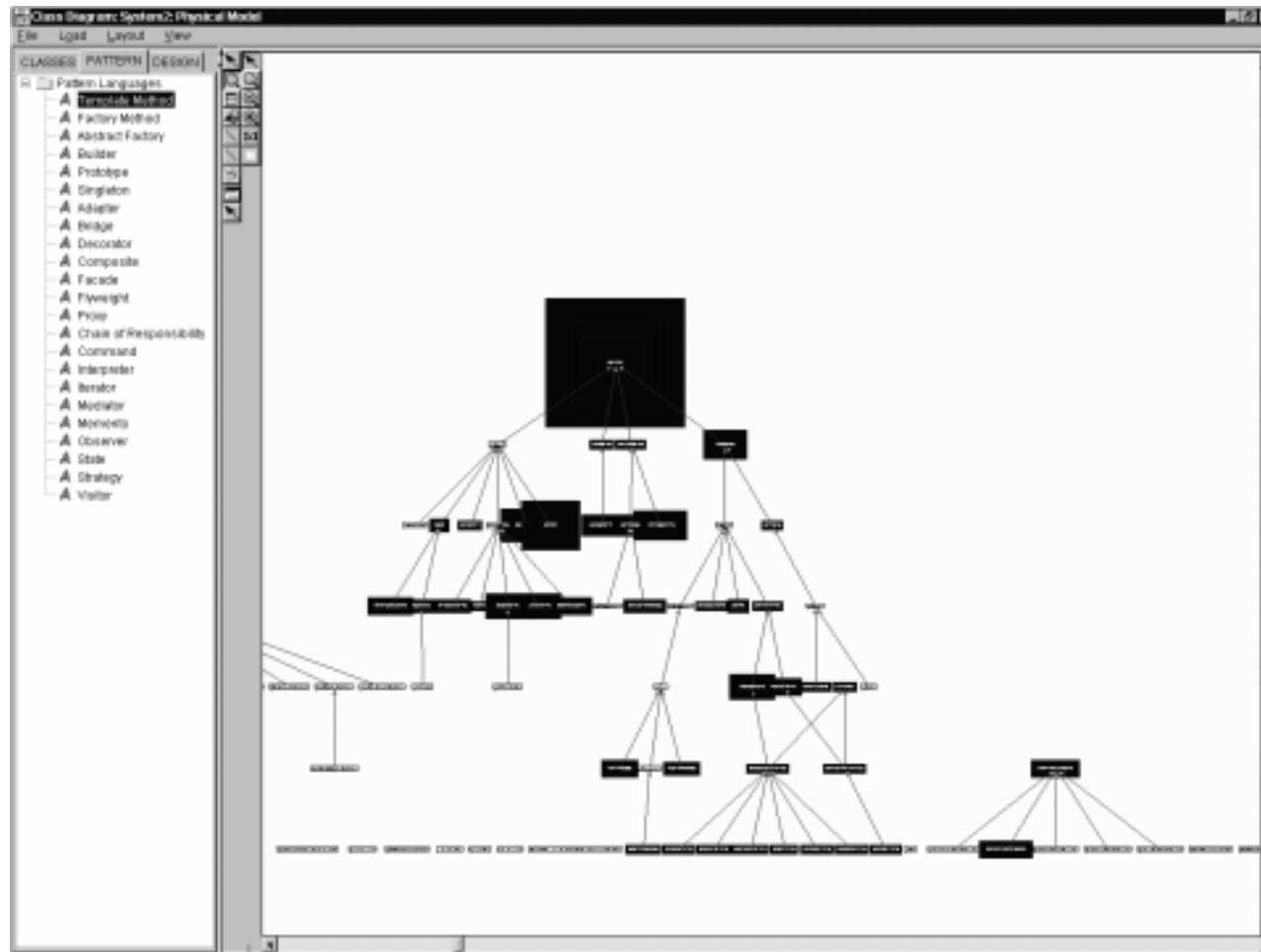
- Analysis of *Bell System 3*
  - ca. 300.000 lines of code, 1221 header files
- Use of Gen++ to run pattern queries
  - Gen++ is bound to the AT&T Cfront Compiler.
  - Gen++ reads the abstract semantic graph (ASG) produced by the AT&T Cfront Compiler.
  - For pattern detection, the whole system needs to be treated as one compilation unit.
  - However, AT&T Cfront Compiler cannot handle 300.000 lines of code in one compilation run.
- Some other problems encountered
  - files missing, unreasonable include nesting, many unresolved references, right setting of macros,...



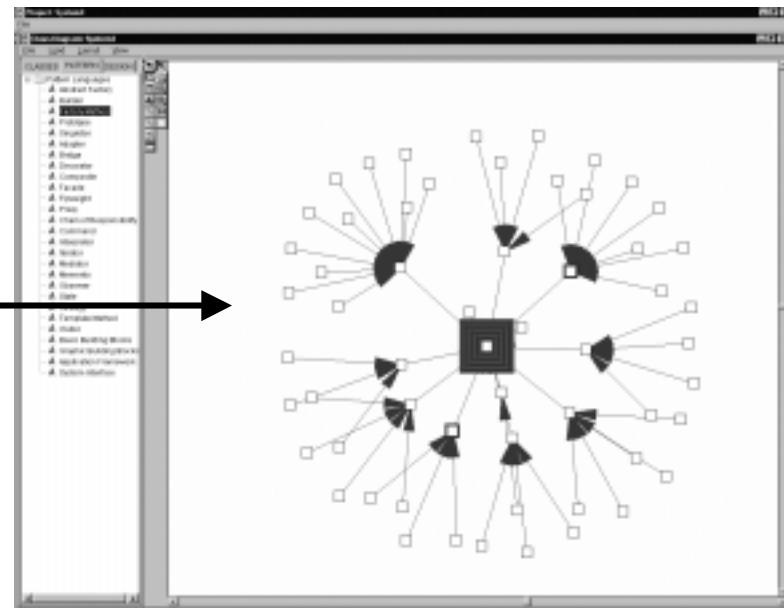
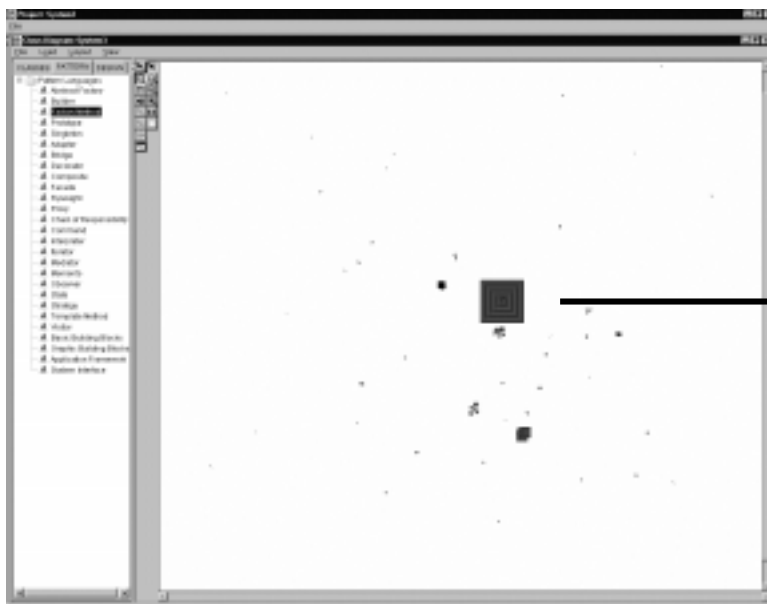
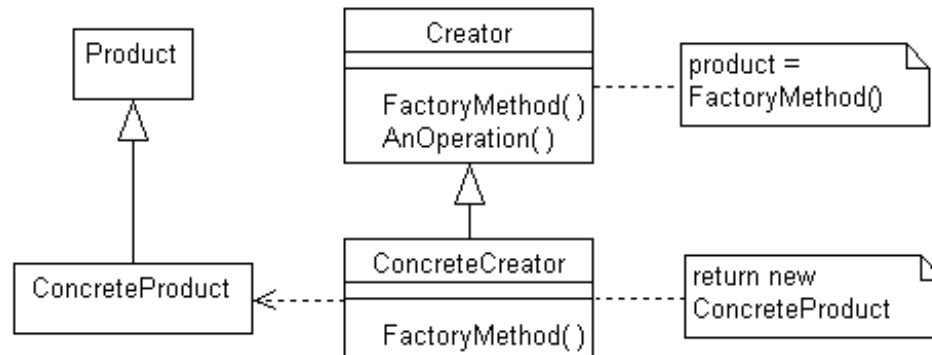
# Template Method: ET++



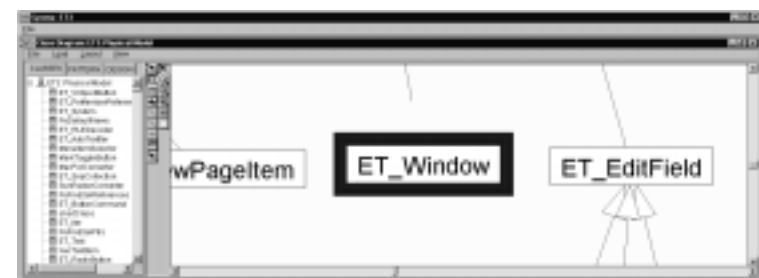
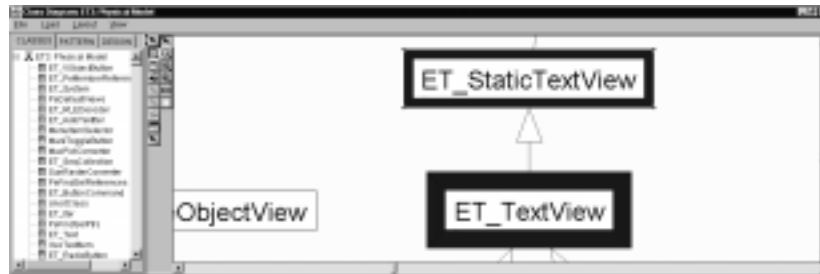
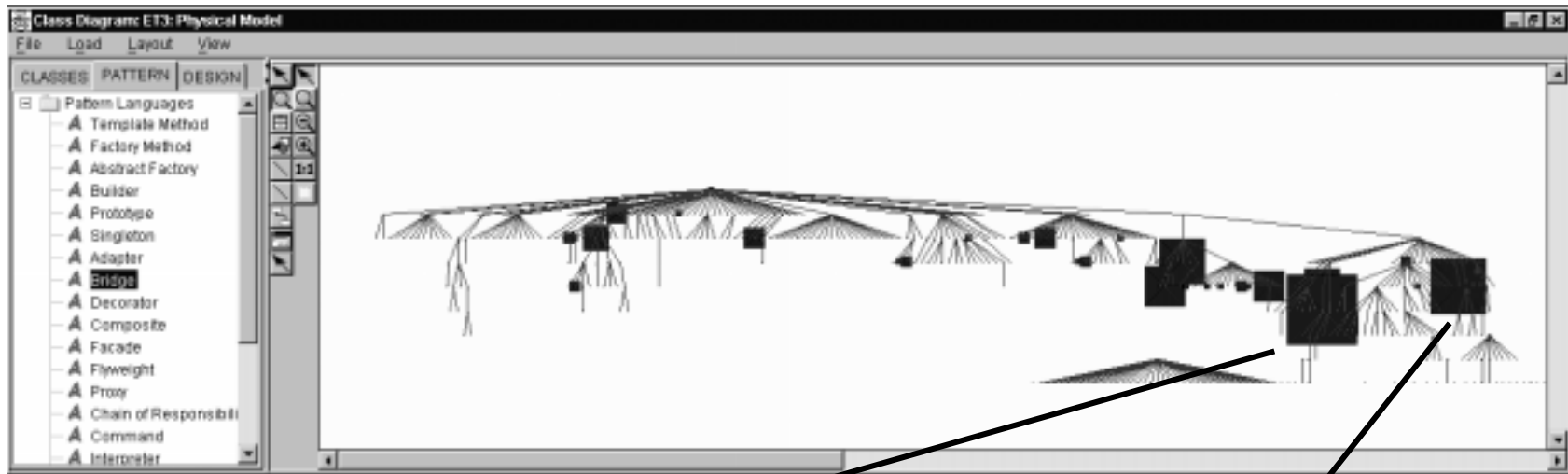
# Template Method: System2



# Factory Method: Bell System3

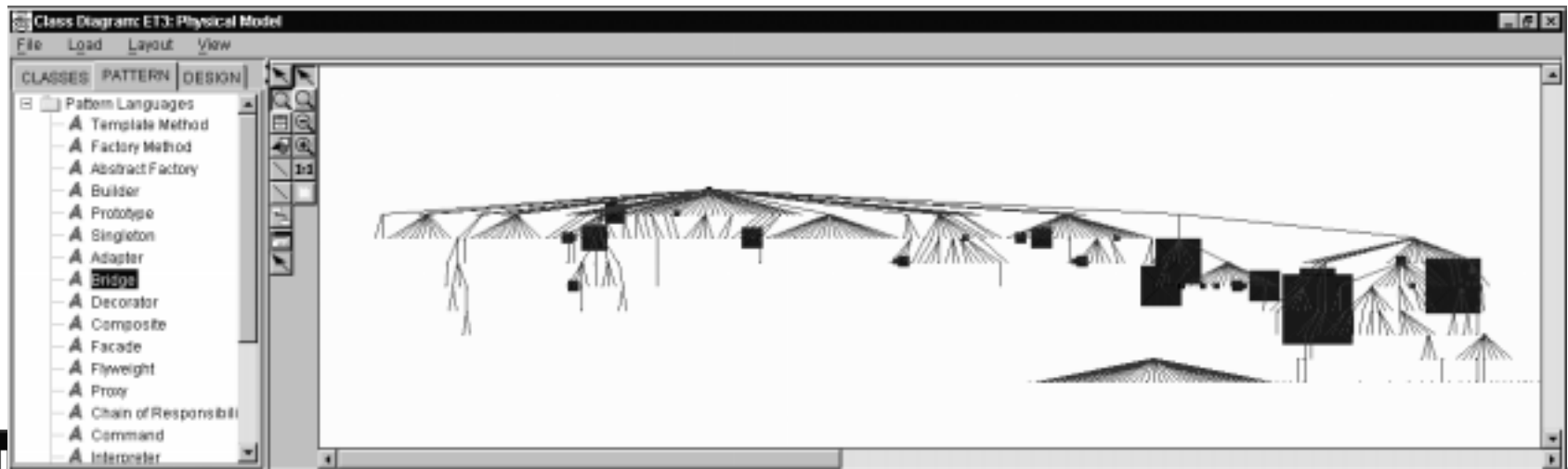
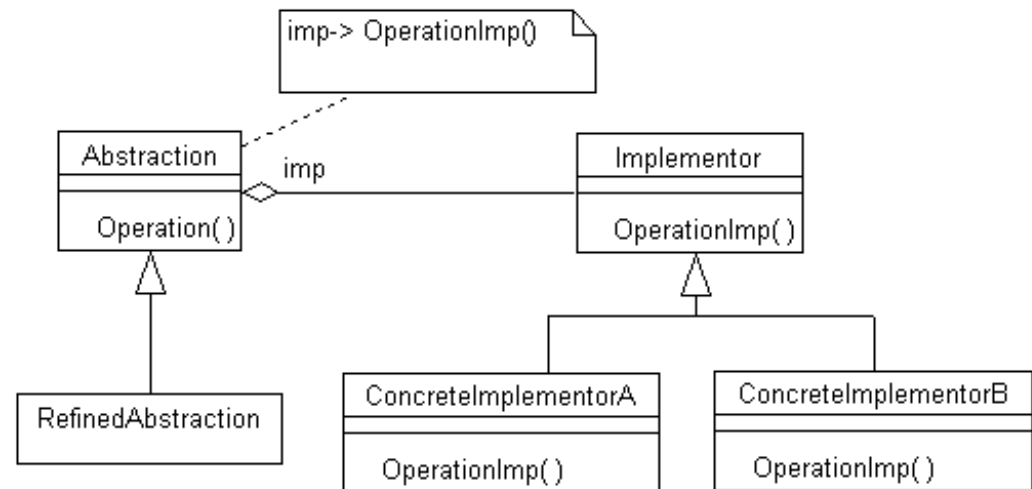


# Bridge - ET++



# Bridge - ET++

- Decouple Abstraction from Implementation



# That 's just the beginning ...

- Future Plans
  - Extension of the repository towards a C++ repository (problem: gigantic amount of data)
  - Generic design component recovery
    - specification of design to be recovered as a UML diagram
  - Architectural patterns (Layers)
  - Pattern matching based on
    - full-text retrieval
    - metrics analysis
    - combined with structural detection
  - Multi-phase detection
  - Handling of polymorphism



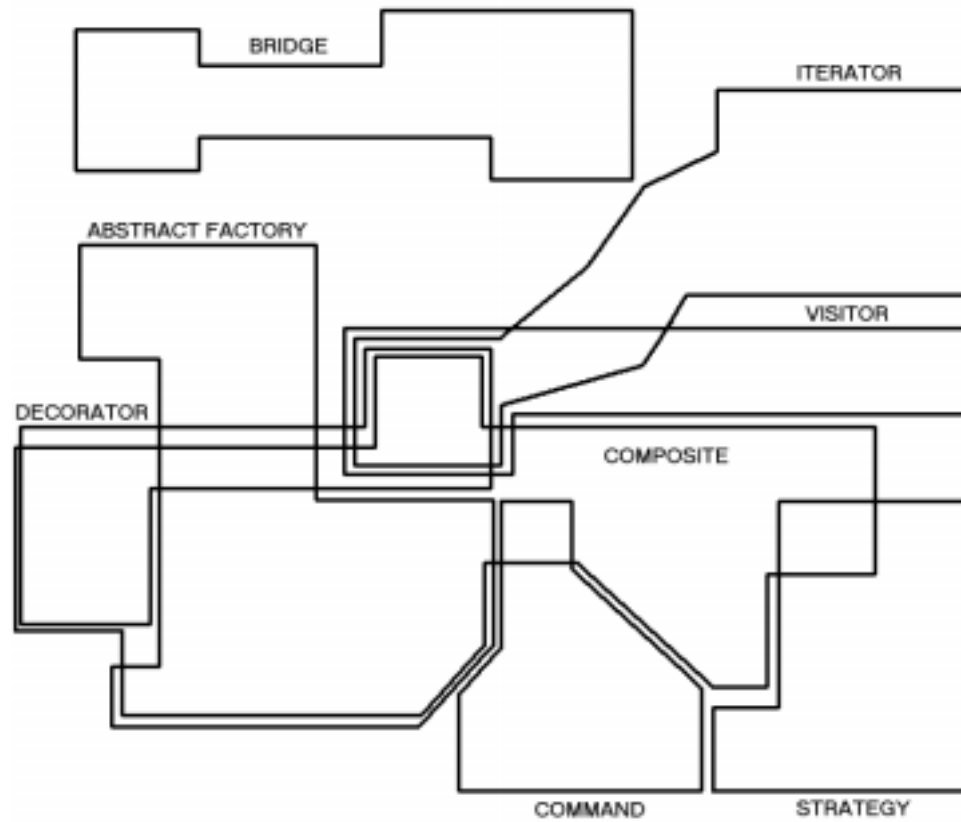
# Design Components: Towards Software Composition at the Design Level

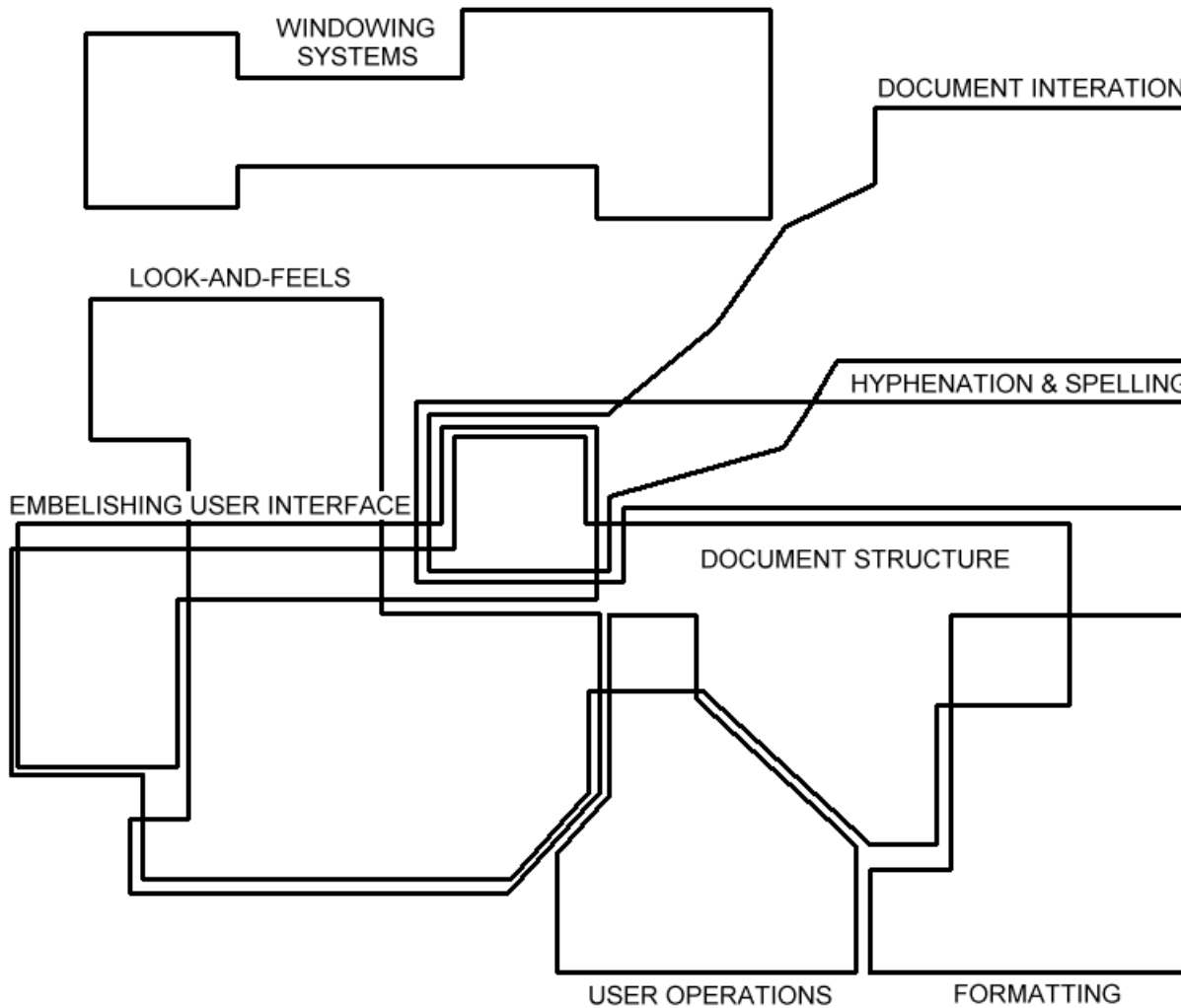
Reified Design Solutions (Patterns) Fit  
for Software Components





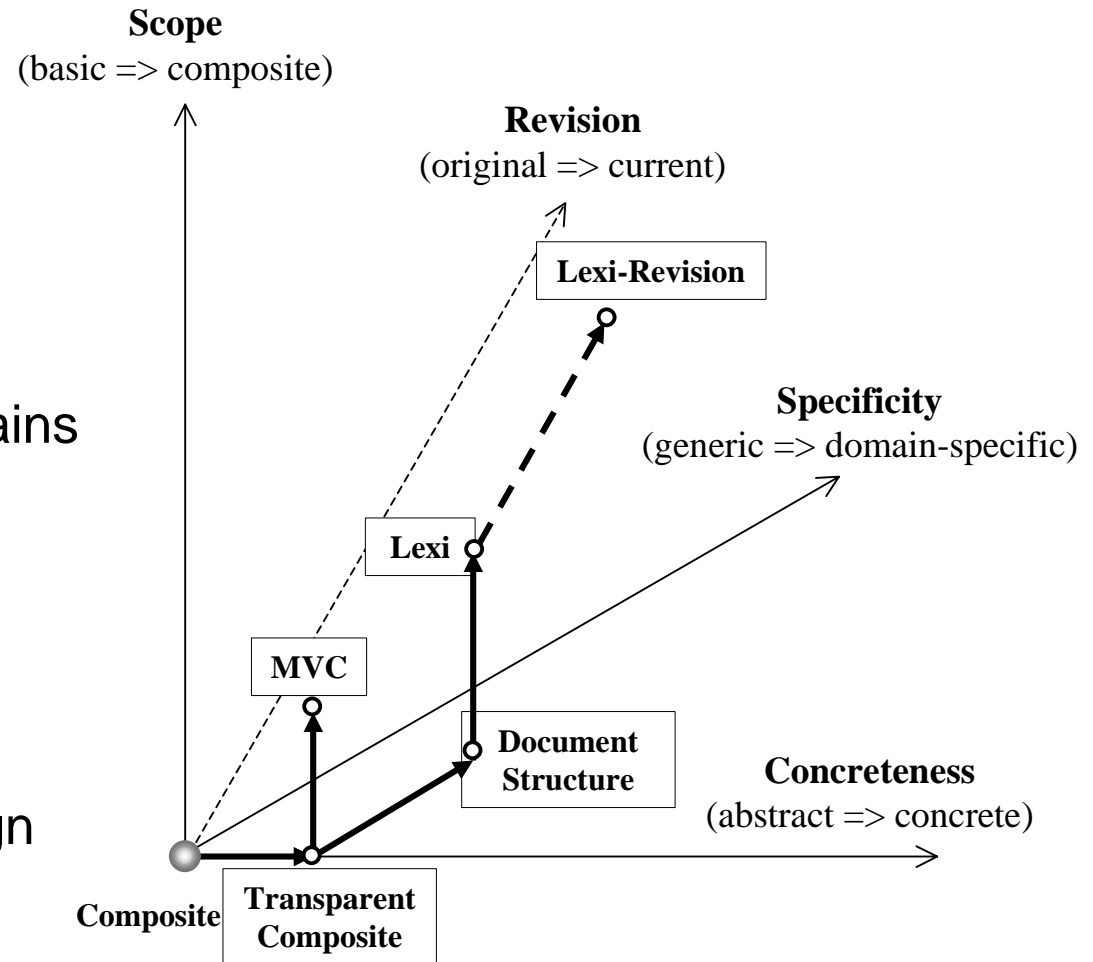






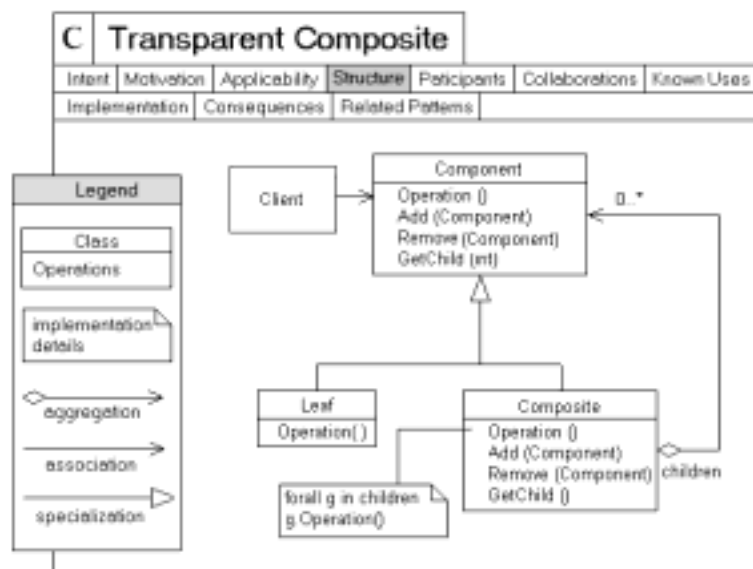
# Four-Dimensional Design Space

- **Concreteness**
  - level of detail of a design solution
- **Specificity**
  - applicability of a design solution to different domains
- **Scope**
  - level of composition of a design solution
- **Revision**
  - change history of a design solution



# Concreteness

- Relates to the level of detail of a design solution.
  - *Abstract components* comprise a set of incomplete elements.
  - *Concrete components* address design trade-offs and tactical decisions.
- Example:
  - Adapter => Object Adapter, Class Adapter
  - Composite => Transparent Composite, Safe Composite

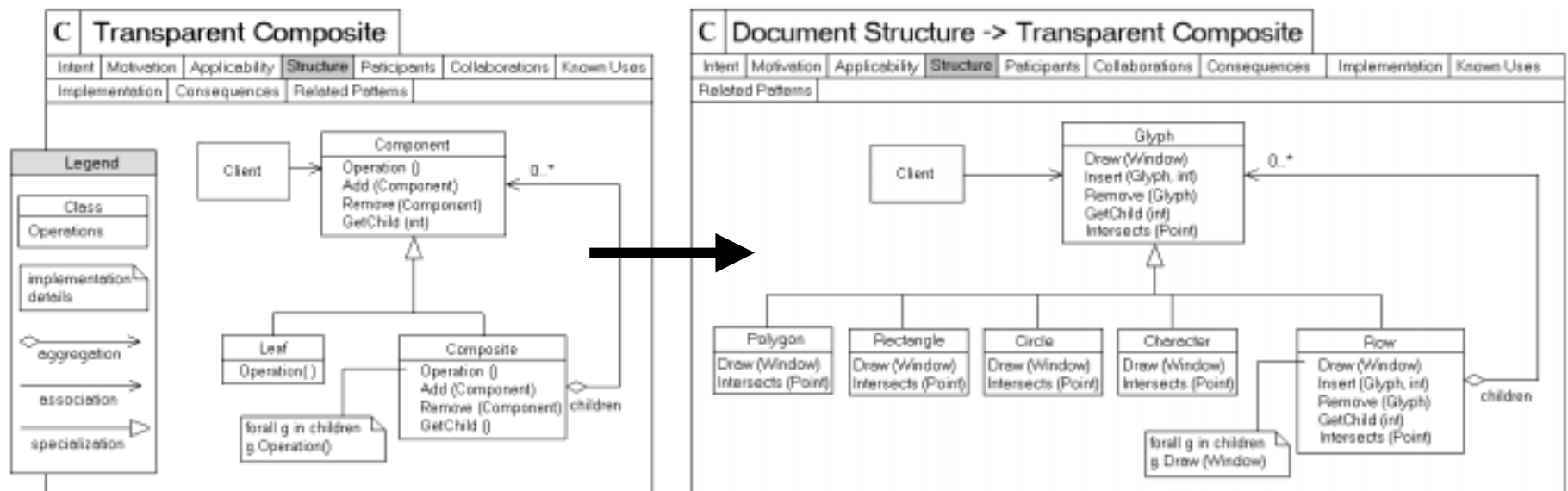


- *Transparent Composite* => Child manipulation methods are in *Component*.
- *Safe Composite* => Child manipulation methods are in *Composite*.



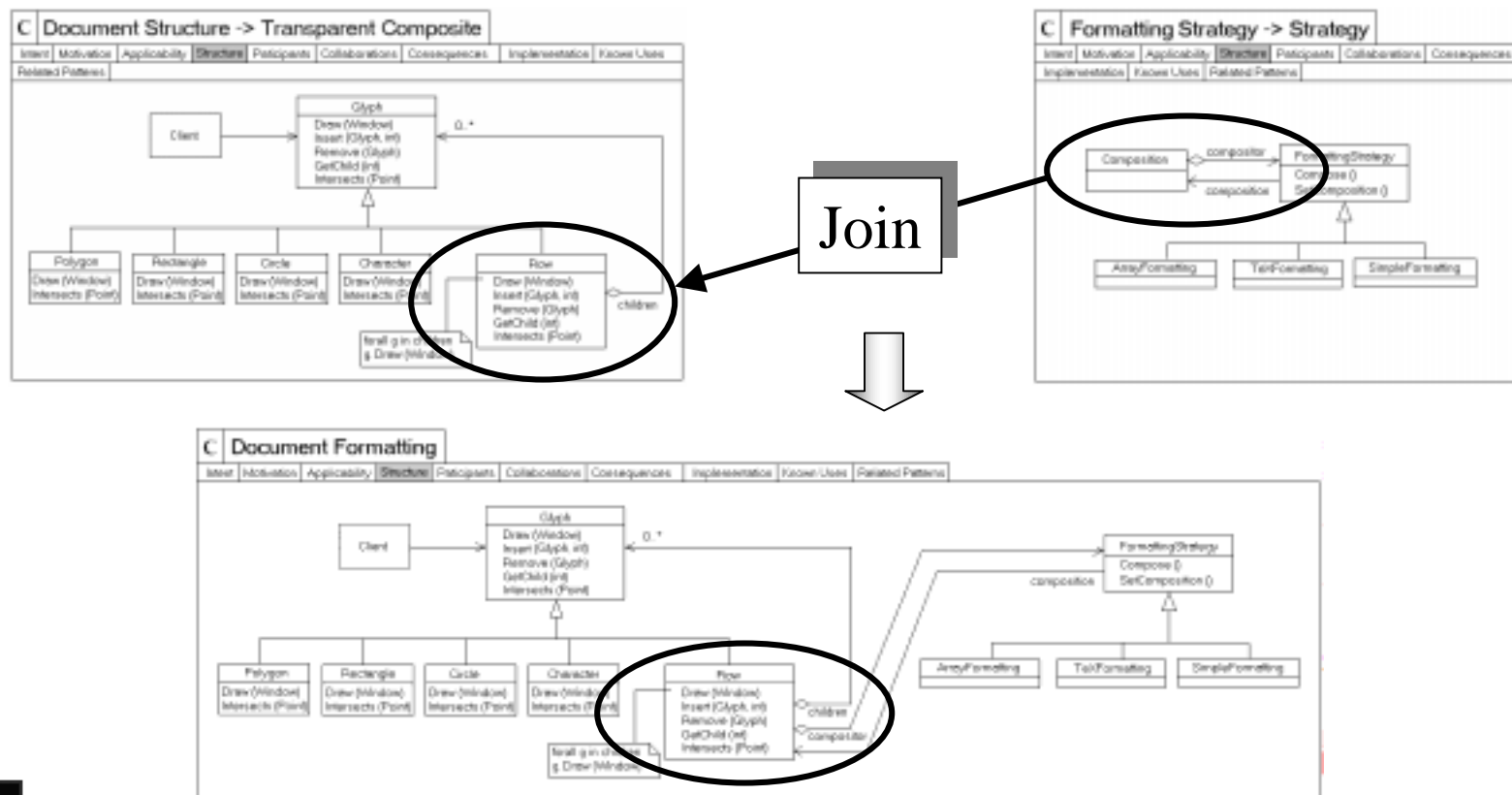
# Specificity

- Relates to the applicability of a design solution to different domains.
  - *Generic components* use domain-independent vocabulary, and the mechanisms are flexible enough to be adapted to many different application areas.
  - *Domain-specific components* are adapted to particular terminology, practices, and technologies used in the application domain at hand.



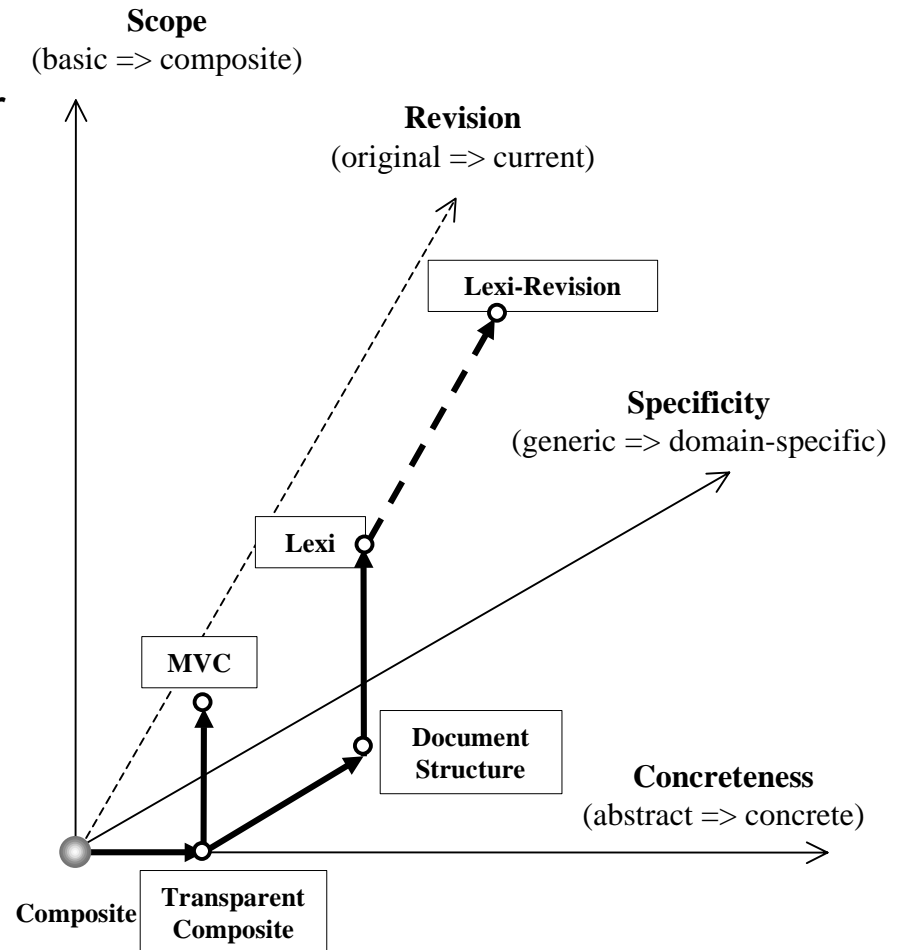
# Scope

- Relates to the level of decomposition of a design solution
  - *Basic components* constitute the primitives for design composition
  - *Composite component* assemble basic components



# Revision

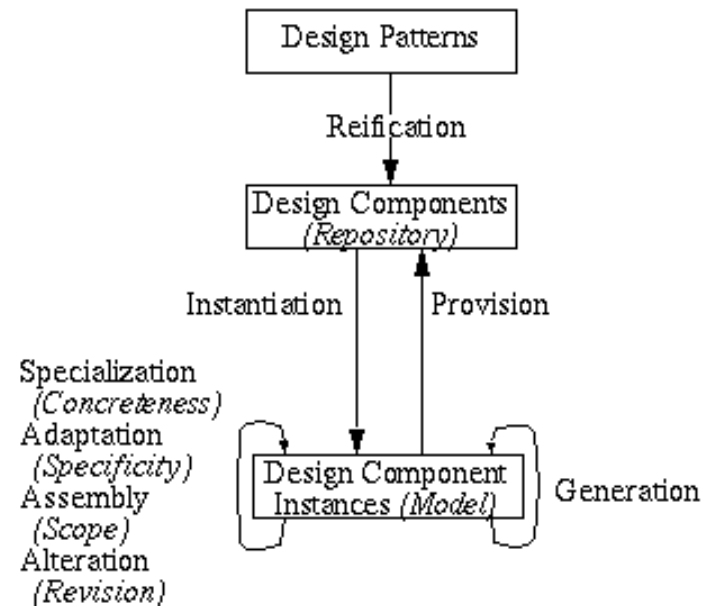
- Relates to the change history of a design solution.
  - *Original components* constitute older revisions.
  - *Current components* constitute the most recent.
- Examples
  - Fault correction.
    - addition, removal, and change of classes, attributes, or methods.
  - Modification of textual constituents of design components.
- Component changes that lead to new revisions should be propagated through some use paths.



# Design Composition

*(a journey through design space)*

- **Reification** - turning patterns into tangible artifacts (=> design components)
- **Instantiation** - creation of particular examples of design components
- **Specialization** - adjustment to design trade-offs and implementation techniques
- **Adaptation** - accommodation of design components to the application domain at hand
- **Assembly** - combination of basic components
- **Alteration** - adjustment to changing requirements
- **Generation** - transformation into executable code
- **Provision** - shelving of an evolving model as a design component





# Design Pattern Engineering

## Concluding Remarks



# Concluding Remarks

- Contribution
  - A software design technique and tool that allows for
    - reification,
    - evolution (specialization, adaptation, assembly, alteration), and
    - deployment
  - of design solutions (patterns) as tangible components
- Impact
  - design rationale becomes a tangible constituent of the software
  - traceability of the software design's evolution
  - preservation and reuse of proven software engineering practices.



# Concluding Remarks

- Vision
  - A design component repository that allows for storage, retrieval, and evolution of design solutions within the four dimensions of design composition.

