Software Architectures

Richard N. Taylor

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

taylor@ics.uci.edu
http://www.ics.uci.edu/~taylor

+1-949-824-6429
+1-949-824-1715 (fax)

Acknowledgments

Will Tracz

Neno Medvidovic

Peyman Oreizy

Dewayne Perry

Overview

- Rationale: why the focus, and what’s new
- Definitions and focus areas
- Processes and domain-specific software architectures
- Dynamic change

Why the Focus on Architecture?

- Architectures are not new -- there has long been a focus on getting the high-level design in good shape. So what's new?
  - Making the architecture explicit
  - Retaining the description
  - Using the description as the basis for system evolution, runtime change, and for analysis
  - Separating computation from communication
  - Separating architecture from implementation
  - Component-based development emphasis
  - A shift in developer focus

Explicit versus Implicit

- The architecture is there whether we make it explicit or not
- If it is implicit, then we have no way of
  - understanding it
  - conforming to it (architectural mismatch)
  - controlling or directing its change (architectural drift or erosion)

An Explicit Architecture Provides a Structural Framework for:

- System development
- Component design and implementation
- System evolution
- Composition of systems
- Systematic reuse
- Retention and exploitation of domain knowledge

Differences Between Architecture and Design

- Architecture is concerned about higher level issues
  - components v. procedures or simple objects
  - interactions among components v. interface design
  - constraints on components and interactions v. algorithms, procedures, and types
- Architecture is concerned with a different set of structural issues
  - Large-grained composition v. procedural composition
  - Component interactions (protocols) v. interaction mechanisms (implementations)
  - Information content v. data types and representations
- Architecture and Patterns

Architectures and Implementations

- A one-to-many relationship
- Bi-directional mapping must be maintained
- Some components may be generated
- The special role of connectors

Definitions

- Perry and Wolf
  - Software Architecture = {Elements, Form, Rationale}
- Garlan and Shaw
  - [Software architecture is a level of design that] goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.
- Kruchten
  - Software architecture deals with the design and implementation of the high-level structure of software.
  - Architecture deals with abstraction, decomposition, composition, style, and aesthetics.

Key Architectural Concepts

- A ***component*** is a unit of computation or a data store. Components are loci of computation and state.
- A ***connector*** is an architectural building block used to model interactions among components and rules that govern those interactions.
- An ***architectural configuration*** or ***topology*** is a connected graph of components and connectors which describes architectural structure.
    - proper connectivity
    - concurrent and distributed properties
    - adherence to design heuristics and style rules

Architecture-Based Software Engineering

An approach to software systems development which uses as its primary abstraction a model of the system's components, connectors, and interconnection topology

Focus Areas

- Architecture description techniques
- Analysis based on architectural descriptions
  - Program-like analyses
  - Predictive performance analyses
  - Static and dynamic
- Architectural styles
- Evolution
  - Specification-time
  - Run-time
- Refinement
- Traceability
- Design process support

Architecture Description Techniques

- Principal Problems
  - Aid stakeholder communication and understanding
- Desired Solutions
  - Multiple perspectives

Architecture Description Languages

- Architectural models as distinct software artifacts
  - communication
  - analysis
  - simulation / system generation
  - evolution
- Informal vs. *formal* models
- General-purpose vs. *special-purpose* languages
- Several prototype ADLs have been developed
  - ACME
  - Aesop
  - ArTek
  - C2
  - Darwin
  - LILEANNA
  - MetaH
  - Rapide
  - SADL
  - UniCon
  - Wright

ADL Definition

- An ***ADL*** is a language that provides features for modeling a software system's conceptual architecture.
- *Essential* features: *explicit* specification of
  - components
    - component interfaces
  - connectors
  - configurations
- *Desirable* features
  - specific aspects of components, connectors, and configurations
  - tool support

Classifying Existing Notations

- Approaches to modeling configurations
  - implicit configuration
  - in-line configuration
  - explicit configuration
- Associating architecture with implementation
  - implementation constraining
  - implementation independent

Related Notations

- High-level design notations
  - e.g., LILEANNA, ArTek
- Module interconnection languages
  - e.g., MIL
- Object-oriented notations
  - e.g., Booch diagrams, UML
- Programming languages
  - e.g., Ada, Java
- Formal specification languages
  - e.g., Z, Obj, CHAM

Analysis

- Principal Problems
  - Evaluate system properties upstream to reduce number and cost of errors
- Desired Solutions
  - Static analysis
    - internal consistency
    - concurrent and distributed properties
    - design heuristics and style rules
  - Dynamic analysis
    - testing and debugging
    - assertion checking
    - runtime properties
  - Predictive performance

Refinement

- Principal Problems
  - Bridge the gap between architecture descriptions and programming languages
- Desired Solutions
  - Specify architectures at different abstraction levels
  - Correct and consistent refinement across levels

Traceability

- Principal Problems
  - Multiple abstraction levels **+** multiple perspectives
- Desired Solutions
  - Traceability across architectural *cross-sections*

Design Process Support

- Principal Problems
    - Decompose large, distributed, heterogeneous systems
- Desired Solutions
    - Multiple perspectives
    - Design guidance and rationale

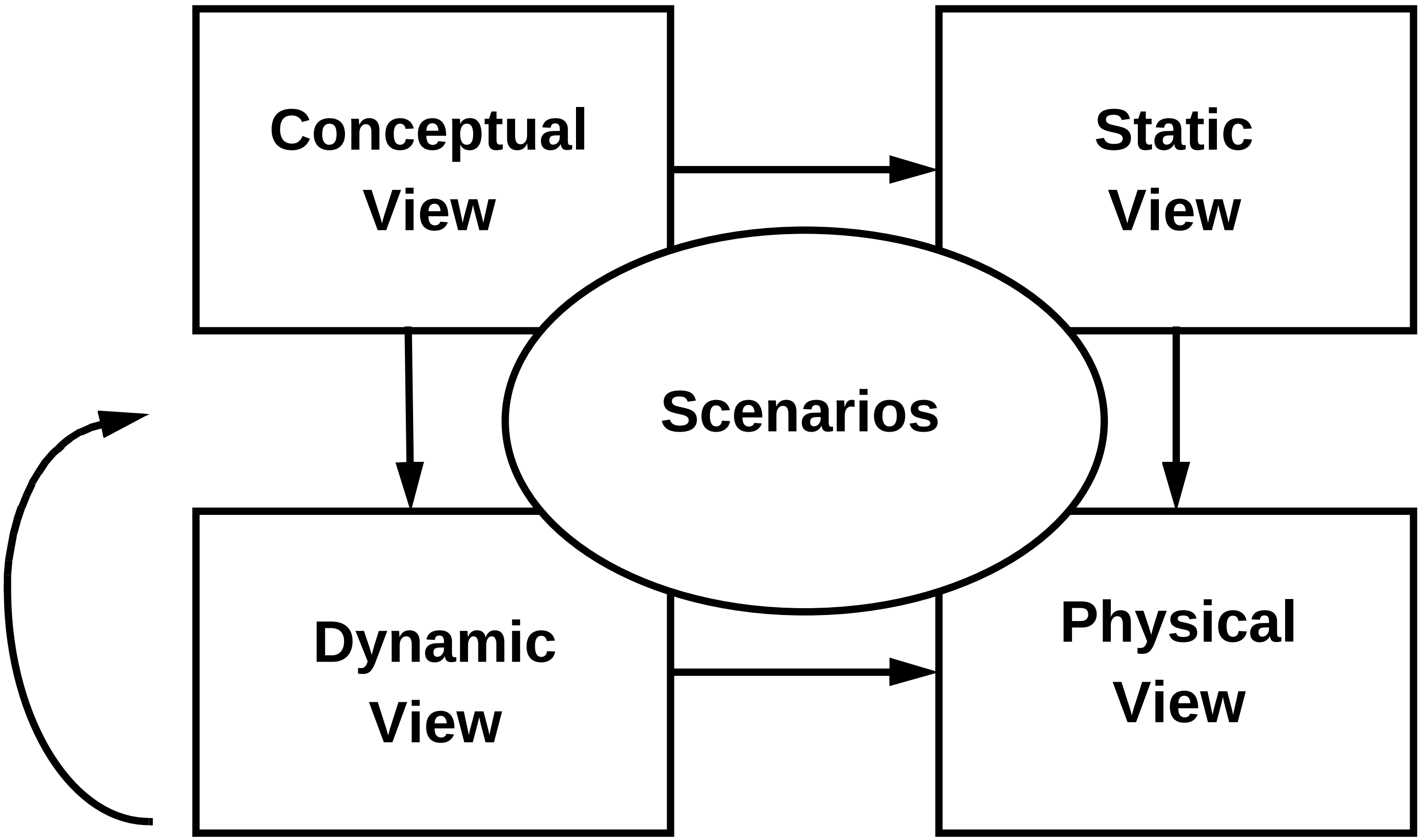
Origins: Where do Architectures Come From?

- Theft, method, or intuition?
- Krutchen’s view: scenario-driven, iterative design
- Recovery
- DSSA approach: domain model, reference requirements, and reference architecture (reflecting experience)

Kruchten’s Views

- 5 views of architectures
  - conceptual
    - “the object model of the design”
  - dynamic
    - concurrency and synchronization aspects
  - physical
    - mapping of software onto hardware
  - static
    - organization of software in the development environment
  - scenarios

The "4+1" View Model



Scenario-Driven Iterative Approach

- Prototype, test, measure, analyze, and refine the architecture in subsequent iterations
- Summary of the Approach:
  - choose scenarios and identify major abstractions
  - map the abstractions to the 4 blueprints
  - implement, test, measure, and analyze the architecture
  - select additional scenarios and reassess the risks
  - fit new scenarios into the original architecture and update blueprints
  - measure under load, in real target environment
  - review all 5 blueprints to detect potential for simplification and reuse
  - update rationale

Architecture Recovery Process

- The models are suggestive of a recovery process
  - create a configuration model
  - determine the types of the components and connectors
  - determine the patterns of interactions among the components
  - abstract the properties of and relationships among the components and connectors
  - abstract useful styles

The DSSA Insight

- Reuse in particular domains is the most realistic approach to reuse
  - reuse in general is too difficult to achieve
  - therefore focus on classes of applications with similar characteristics
- Software architectures provide a framework for reuse

Domain-Specific Software Architectures

- DSSA is an assemblage of <u>software components</u>
  - specialized for a particular type of task (domain)
  - <u>generalized</u> for effective use across that domain
  - composed in a <u>standardized structure</u> (topology) effective for building successful applications

  \- *Rick Hayes-Roth, Teknowledge, 1994*

- DSSA is comprised of
  - a <u>domain model</u>,
  - <u>reference requirements</u>,
  - a <u>reference (parameterized) architecture</u> (expressed in an ADL),
  - its supporting <u>infrastructure/environment</u>, and
  - a <u>process/methodology</u> to instantiate/refine and evaluate it.

  \- *Will Tracz, Loral, 1995*

What Is a Domain Model?

- A domain model is a representation of
  - functions being performed in a domain
  - data, information, and entities flowing among the functions
- It deals with the problem space
- Domain model is a product of *domain analysis*
  - “it is like several blind men describing an elephant”
- Fundamental objectives of domain analysis:
  - standardize domain terminology and semantics
  - provide basis for standardized descriptions of specific problems to be solved in the domain
- Domain model elements
  - customer needs statement, scenarios, domain dictionary, context and ER diagrams, data-flow, state-transition, and object models

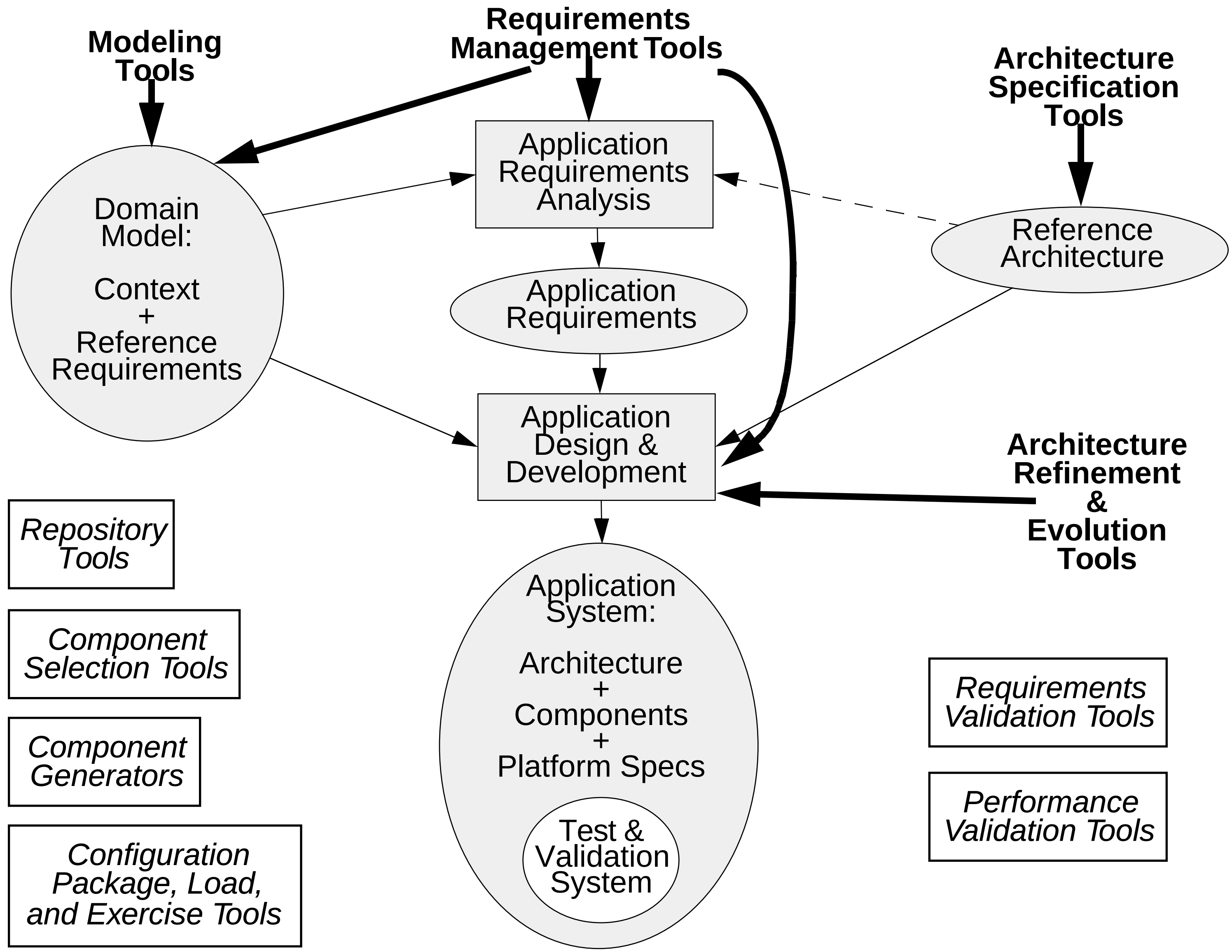
What Are Reference Requirements?

- Requirements that apply to the entire domain
- Reference requirements contain
  - *defining* characteristics of the problem space
    - functional requirements
  - *limiting* characteristics (constraints) in the solution space
    - non-functional requirements (e.g., security, performance)
    - design requirements (e.g., architectural style, UI style)
    - implementation requirements (e.g., platform, language)

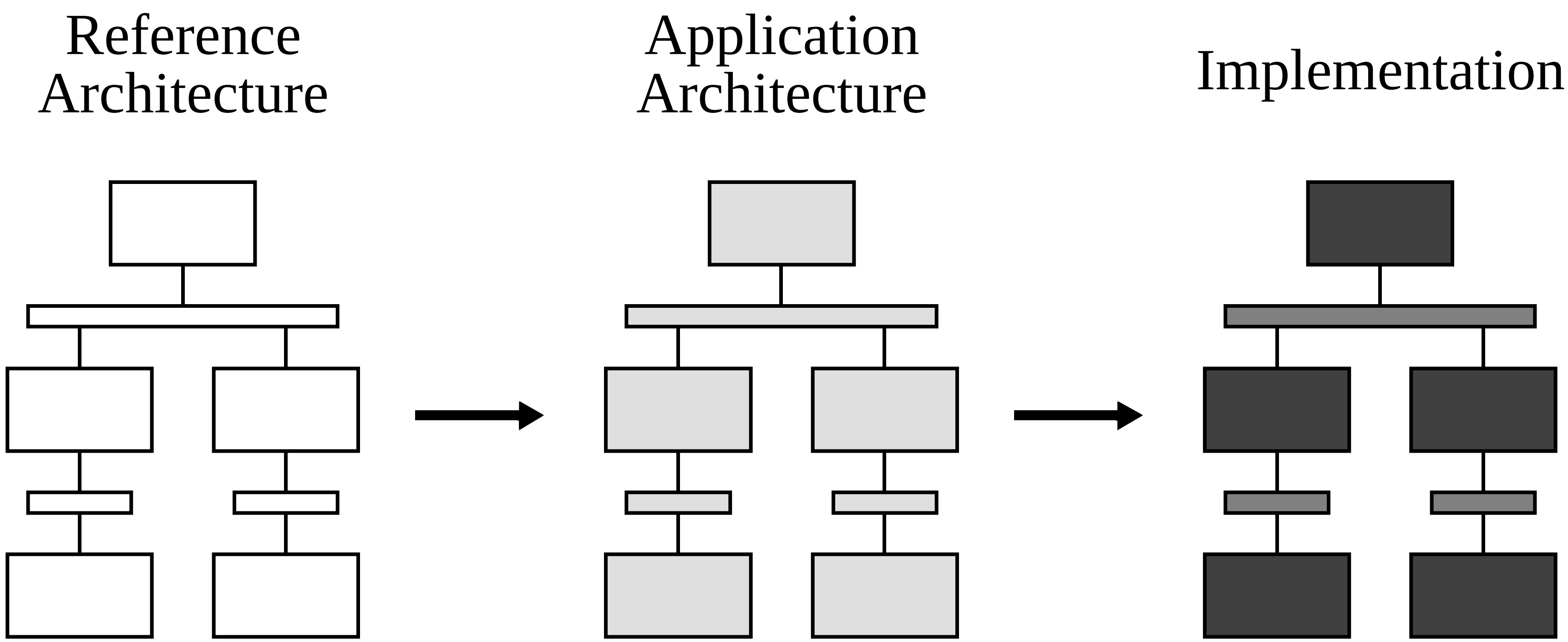
What Is a Reference Architecture?

- Standardized, generic architecture(s) describing all systems in a domain
- Based on the constraints in reference requirements
- Specifies syntax and semantics of high-level components
- It is reusable, extendable, and configurable
- Instantiated to create a specific application’s architecture
- Reference architecture elements
    - model (topology), configuration decision tree, architecture schema (design record), dependency diagram, component interface descriptions, constraints, rationale

A Simplified DSSA Process with Principal Supporting Tool Types



Another Simplified View of the DSSA Process



Organizational Considerations

- Architecture/Asset base
  - across product lines
  - product line specific
  - product specific
- Supporting technology
  - global to the company
  - Processes - support multiple product lines

Architectural Style

- Garlan:
  - Architectural styles are recurring organizational patterns and idioms.
- Medvidovic, Oreizy, Robbins, Taylor:
  - Architectural style is an abstraction of recurring composition and communication characteristics of a set of architectures.
  - Styles are key design idioms that enable exploitation of suitable structural and evolution patterns and facilitate component and process reuse.

Benefits of Styles (1)

- Design Reuse
  - solutions with well-understood properties can be reapplied to new problems
- Code Reuse
  - invariant aspects of a style lend themselves to shared implementations
- Understandability of system organization
  - just knowing that something is a “client-server” architecture conveys a lot of information

Benefits of Styles (2)

- Interoperability
  - supported by style standardization (e.g., CORBA, SoftBench)
- Style-specific analyses
  - constrained design space
  - some analyses not possible on ad-hoc architectures or architectures in certain styles
- Visualizations
  - style-specific depictions that match engineers' mental models

Basic Properties of Styles

- They provide a *vocabulary* of design elements
  - component and connector types (e.g., pipes, filters, servers...)
- They define a set of *configuration rules*
  - topological constraints that determine permitted composition of elements
- They define a *semantic interpretation*
  - compositions of design elements have well-defined meanings
- They define *analyses* that can be performed on systems built in the style
  - code generation is a special kind of analysis

Three Views of Architectural Style

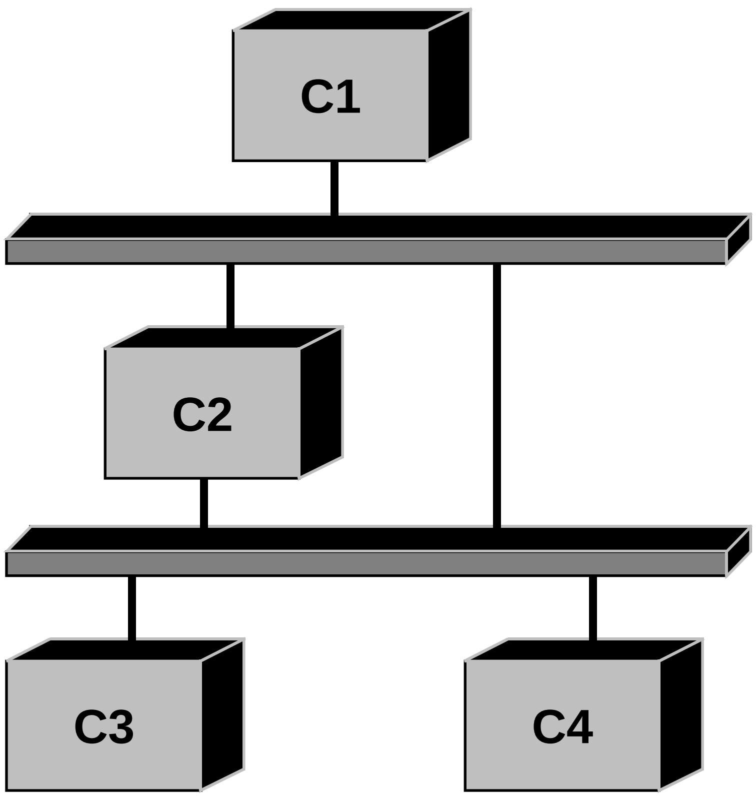
	Language	System of Types	Theory
Vocabulary	a set of grammatical productions	a set of types; in OO, sub- and super-types possible;	represented indirectly, in terms of elements' logical properties
Configuration Rules	context-free and -sensitive grammar rules	maintained as type invariants	defined as further axioms
Semantic Interpretations	standard techniques for assigning meaning to languages	operationally realized in the code that modifies type instances	defined as further axioms
Analyses	performed on architectural "programs" (compilation, flow)	dependent on types involved (type checking, code generation)	by proving theorems, thereby extending the theory of the style

Comparisons of Style Views

	Language	System of Types	Theory
Representation of Structure	explicit (language expression or abstract syntax tree)	explicit (interconnected collection of objects)	implicit (set of assertions)
Substyles	no way to define them	new style types are subtypes of superstyle (e.g., Aesop)	defined in terms of theory inclusion
Refinement	not handled	not handled	natural for defining inter-layer abstraction mappings
Automated Support	programming language tools, e.g., type checkers, code generators, etc.	OO databases and tools for storing, visualizing, and manipulating designs	formal manipulation systems, e.g., theorem provers and model checkers

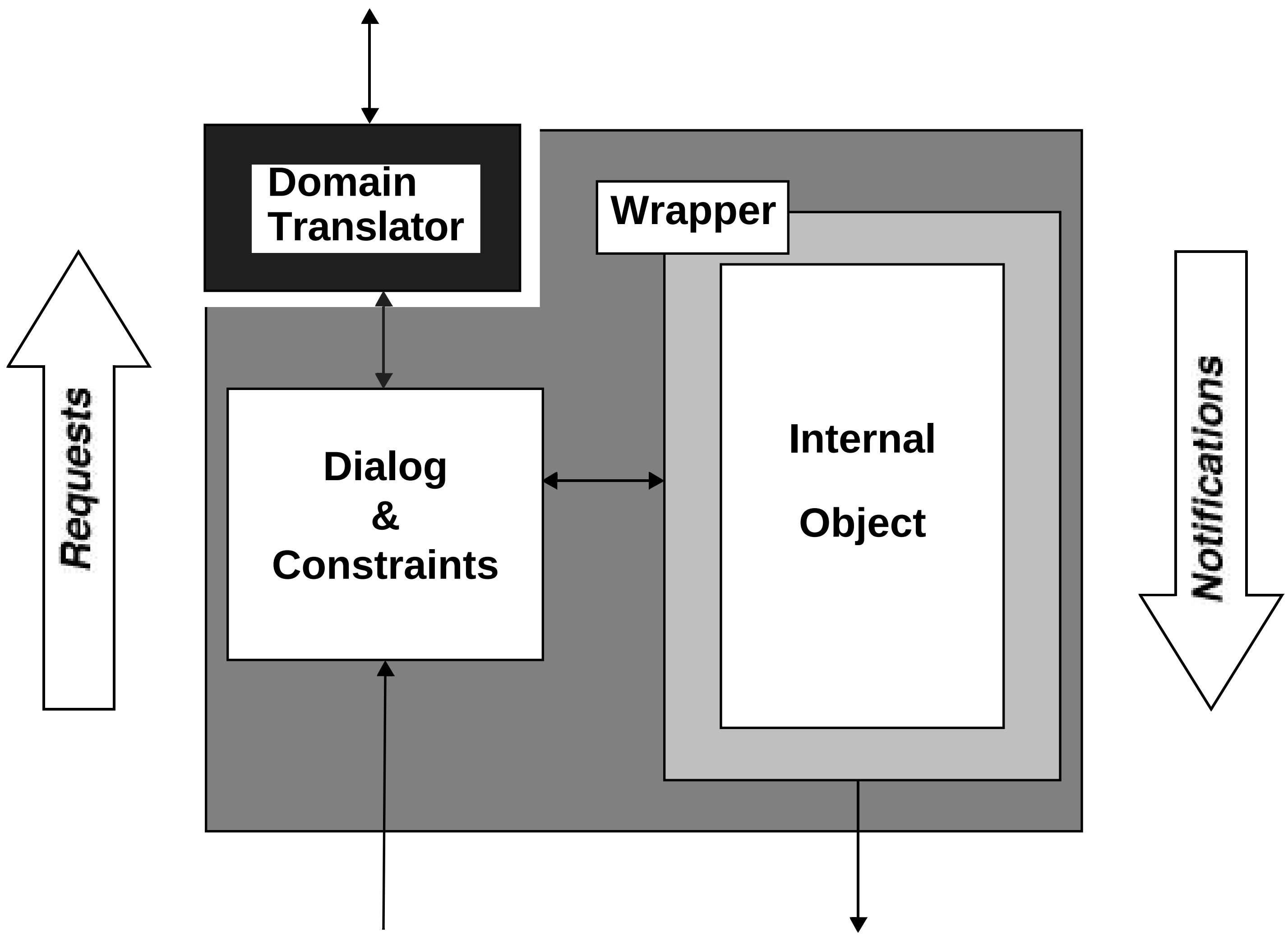
Overview of the C2 Style

- A component- and message-based style
- C2 architectures are networks of concurrent components hooked together by connectors

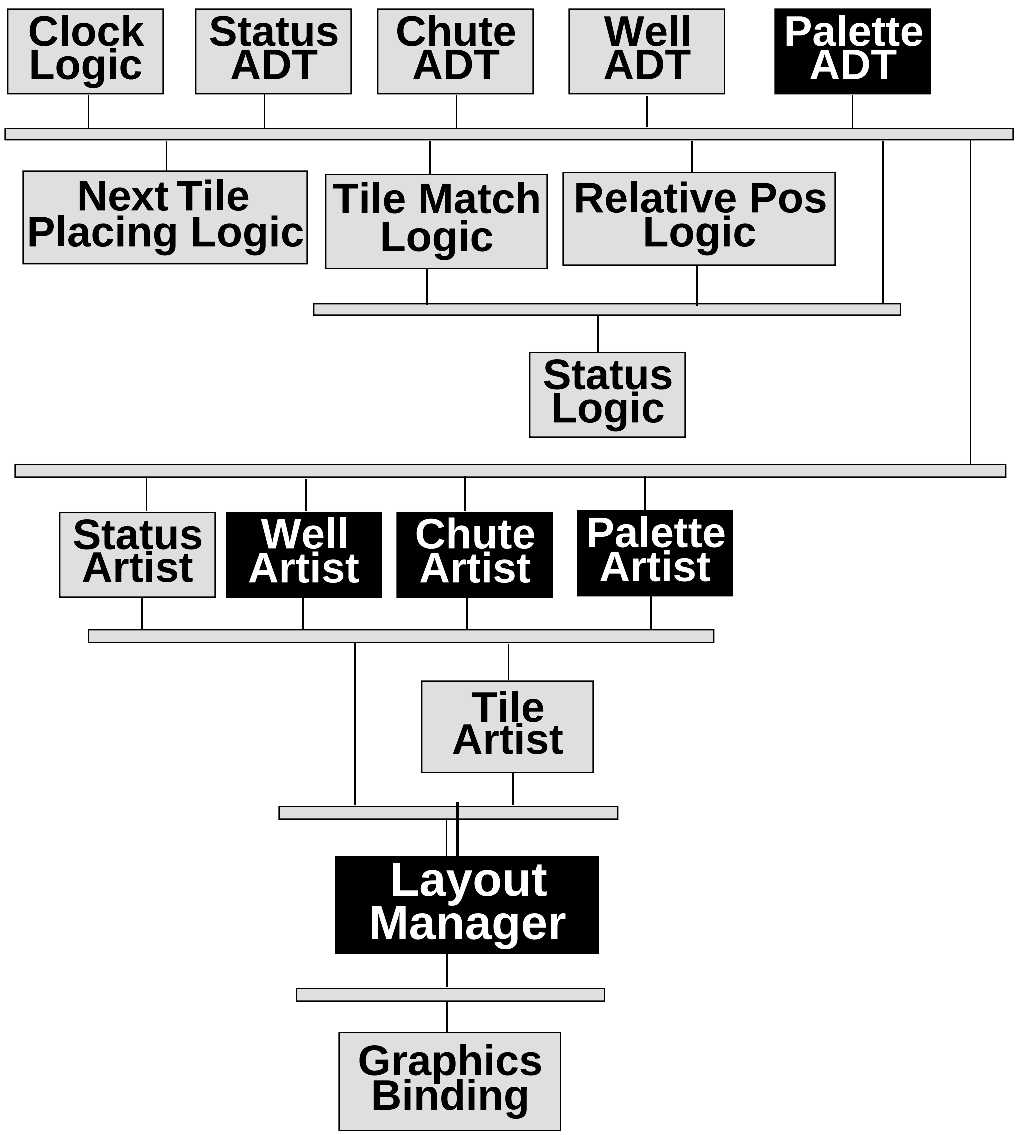


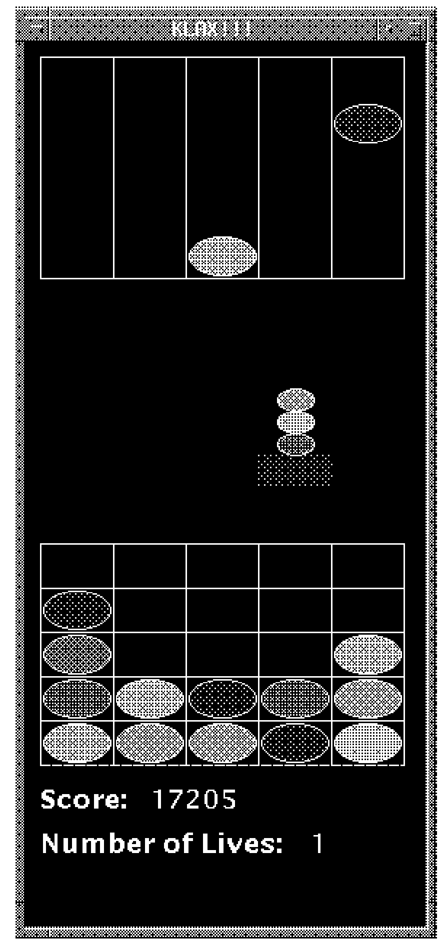
- no component-to-component links
- “one up, one down” rule for components
- connector-to-connector links are allowed
- “many up, many down” rule for connectors
- all communication by exchanging messages
- *substrate independence*

Internal Architecture of a C2 Component



Example C2-Style Architecture



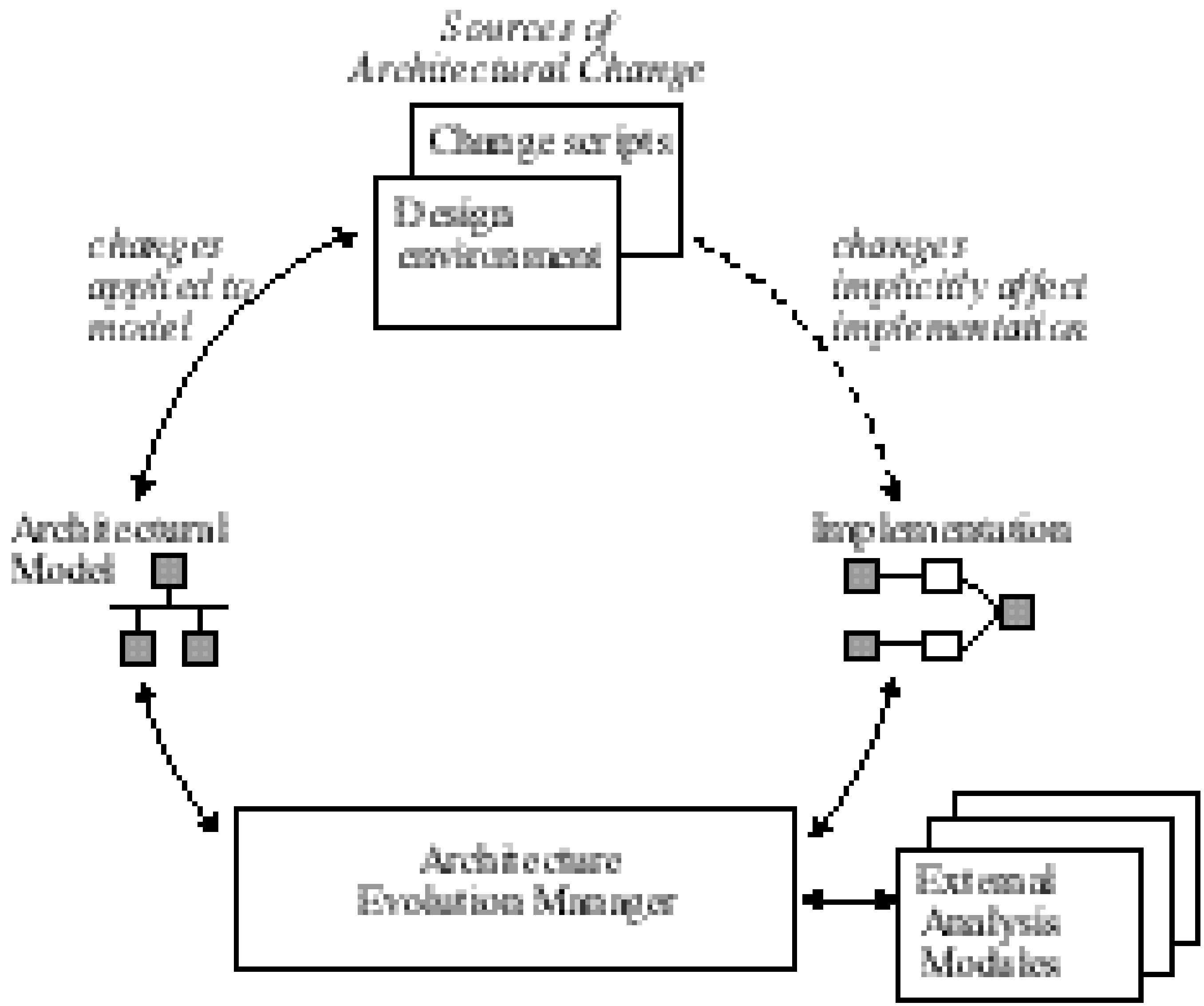


Architectural Evolution

- Principal Problems
  - Evolution of design elements
  - Evolution of system families
- Desired Solutions
  - Specification-time evolution
    - subtyping
    - incremental specification
    - system families
  - Execution-time evolution
    - replication, insertion, removal, and reconnection
    - planned or unplanned
    - constraint satisfaction

Execution-Time Evolution

- The role of architectures at run-time
- The special role of connectors
- The C2 project


Sources of
Architectural Change
Change scripts
Design
environment
changes
applied to
model
changes
implicitly affect
implementation
Architectural
Model
Implementation
Architecture
Evolution Manager
External
Analysis
Modules