Implementing Architectures

Software Architecture

Lecture 15

Objectives

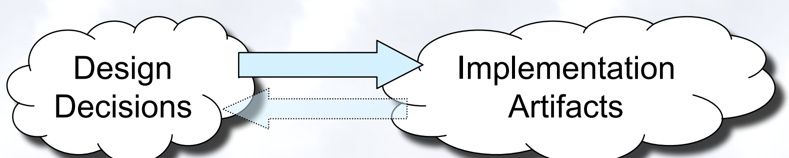
- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

The Mapping Problem

- Implementation is the one phase of software engineering that is not optional
- Architecture-based development provides a unique twist on the classic problem
  - It becomes, in large measure, a *mapping* activity



- Maintaining mapping means ensuring that our architectural intent is reflected in our constructed systems

Common Element Mapping

- Components and Connectors
  - Partitions of application computation and communication functionality
  - Modules, packages, libraries, classes, explicit components/connectors in middleware
- Interfaces
  - Programming-language level interfaces (e.g., APIs/function or method signatures) are common
  - State machines or protocols are harder to map

Common Element Mapping (cont'd)

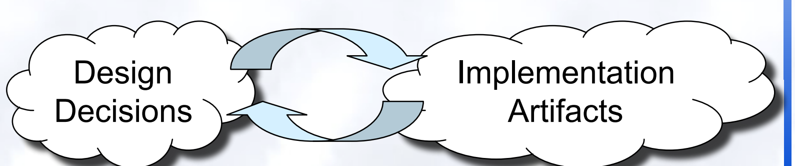
- Configurations
  - Interconnections, references, or dependencies between functional partitions
  - May be implicit in the implementation
  - May be externally specified through a MIL and enabled through middleware
  - May involve use of reflection
- Design rationale
  - Often does not appear directly in implementation
  - Retained in comments and other documentation

Common Element Mapping (cont'd)

- Dynamic Properties (e.g., behavior):
  - Usually translate to algorithms of some sort
  - Mapping strategy depends on how the behaviors are specified and what translations are available
  - Some behavioral specifications are more useful for generating analyses or testing plans
- Non-Functional Properties
  - Extremely difficult to do since non-functional properties are abstract and implementations are concrete
  - Achieved through a combination of human-centric strategies like inspections, reviews, focus groups, user studies, beta testing, and so on

One-Way vs. Round Trip Mapping

- Architectures inevitably change after implementation begins
  - For maintenance purposes
  - Because of time pressures
  - Because of new information
- Implementations can be a source of new information
  - We learn more about the feasibility of our designs when we implement
  - We also learn how to optimize them



One-Way vs. Round Trip Mapping (cont'd)

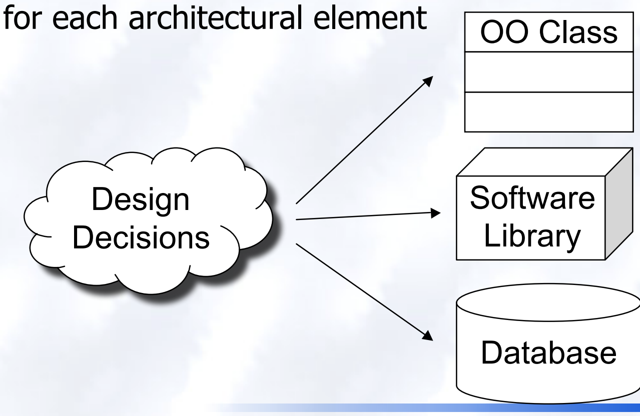
- Keeping the two in sync is a difficult technical and managerial problem
  - Places where strong mappings are not present are often the first to diverge
- One-way mappings are easier
  - Must be able to understand impact on implementation for an architectural design decision or change
- Two way mappings require more insight
  - Must understand how a change in the implementation impacts architecture-level design decisions

One-Way vs. Round Trip Mapping (cont'd)

- One strategy: limit changes
  - If all system changes must be done to the architecture first, only one-way mappings are needed
  - Works very well if many generative technologies in use
  - Often hard to control in practice; introduces process delays and limits implementer freedom
- Alternative: allow changes in either architecture or implementation
  - Requires round-trip mappings and maintenance strategies
  - Can be assisted (to a point) with automated tools

Architecture Implementation Frameworks

- Ideal approach: develop architecture based on a known style, select technologies that provide implementation support for each architectural element

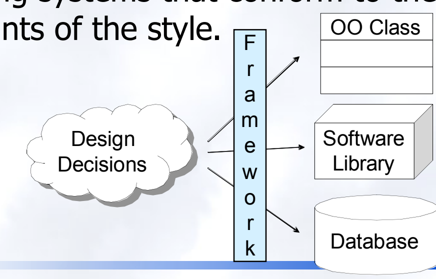


Architecture Implementation Frameworks

- This is rarely easy or trivial
  - Few programming languages have explicit support for architecture-level constructs
  - Support infrastructure (libraries, operating systems, etc.) also has its own sets of concepts, metaphors, and rules
- To mitigate these mismatches, we leverage an *architecture implementation framework*

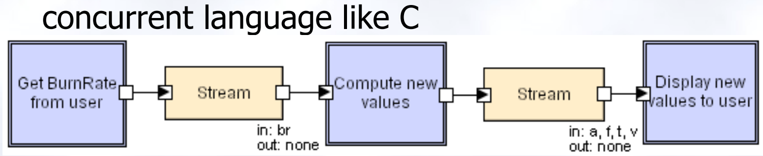
Architecture Implementation Frameworks

- **Definition:** An *architecture implementation framework* is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style *in code*, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style.



Canonical Example

- The standard I/O ('stdio') framework in UNIX and other operating systems
  - Perhaps the most prevalent framework in use today
  - Style supported: pipe-and-filter
  - Implementation technologies supported: concurrent process-oriented operating system, (generally) non-concurrent language like C



More on Frameworks

- Frameworks are meant to assist developers in following a style
  - But generally do not *constrain* developers from violating a style if they really want to
- Developing applications in a target style does not *require* a framework
  - But if you follow good software engineering practices, you'll probably end up developing one anyway
- Frameworks are generally considered as underlying infrastructure or substrates from an architectural perspective
  - You won't usually see the framework show up in an architectural model, e.g., as a component

Same Style, Different Frameworks

- For a given style, there is no one perfect architecture framework
  - Different target implementation technologies induce different frameworks
    - stdio vs. iostream vs. java.io
- Even in the same (style/target technology) groupings, different frameworks exist due to different qualitative properties of frameworks
  - java.io vs. java.nio
  - Various C2-style frameworks in Java

Evaluating Frameworks

- Can draw out some of the qualitative properties just mentioned
- Platform support
  - Target language, operating system, other technologies
- Fidelity
  - How much style-specific support is provided by the framework?
    - Many frameworks are more general than one target style or focus on a subset of the style rules
  - How much enforcement is provided?

Evaluating Frameworks (cont'd)

- Matching Assumptions
  - Styles impose constraints on the target architecture/application
  - Frameworks can induce constraints as well
    - E.g., startup order, communication patterns …
  - To what extent does the framework make too many (or too few) assumptions?
- Efficiency
  - Frameworks pervade target applications and can potentially get involved in any interaction
  - To what extent does the framework limit its slowdown and provide help to improve efficiency if possible (consider buffering in stdio)?

Evaluating Frameworks (cont'd)

- Other quality considerations
  - Nearly every other software quality can affect framework evaluation and selection
    - Size
    - Cost
    - Ease of use
    - Reliability
    - Robustness
    - Availability of source code
    - Portability
    - Long-term maintainability and support

Middleware and Component Models

- This may all sound similar to various kinds of middleware/component frameworks
  - CORBA, COM/DCOM, JavaBeans, .NET, Java Message Service (JMS), etc.
- They are closely related
  - Both provide developers with services not available in the underlying OS/language
  - CORBA provides well-defined interfaces, portability, remote procedure call…
  - JavaBeans provides a standardized packaging framework (the bean) with new kinds of introspection and binding

Middleware and Component Models (cont'd)

- Indeed, architecture implementation frameworks *are* forms of middleware
  - There's a subtle difference in how they emerge and develop
  - Middleware generally evolves based on a set of *services* that the developers want to have available
    - E.g., CORBA: Support for language heterogeneity, network transparency, portability
  - Frameworks generally evolve based on a particular *architectural style* that developers want to use
- Why is this important?

Middleware and Component Models (cont'd)

- By focusing on *services*, middleware developers often make other decisions that substantially impact architecture
- E.g., in supporting network transparency and language heterogeneity, CORBA uses RPC
  - But is RPC necessary for these services or is it just an enabling technique?
- In a very real way, middleware induces an architectural style
  - CORBA induces the 'distributed objects' style
  - JMS induces a distributed implicit invocation style
- Understanding these implications is essential for not having major problems when the tail wags the dog!

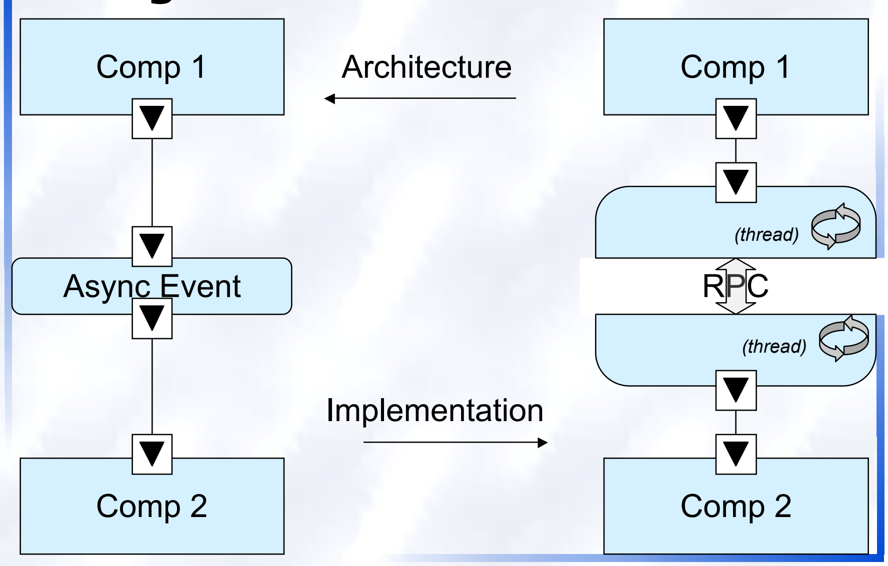
Resolving Mismatches

- A style is chosen first, but the middleware selected for implementation does not support (or contradicts) that style
- A middleware is chosen first (or independently) and has undue influence on the architectural style used
- Strategies
  - Change or adapt the style
  - Change the middleware selected
  - Develop glue code
  - Leverage parts of the middleware and ignore others
  - Hide the middleware in components/connectors

Use the middleware as the basis for a framework

Software Architecture Foundations, Theory, and Practice

Hiding Middleware in Connectors



Building a New Framework

- Occasionally, you need a new framework
  - The architectural style in use is novel
  - The architectural style is not novel but it is being implemented on a platform for which no framework exists
  - The architectural style is not novel and frameworks exist for the target platform, but the existing frameworks are inadequate
- Good framework development is extremely difficult
  - Frameworks pervade nearly every aspect of your system
  - Making changes to frameworks often means changing the entire system
  - A task for experienced developers/architects

New Framework Guidelines

- Understand the target style first
  - Enumerate all the rules and constraints in concrete terms
  - Provide example design patterns and corner cases
- Limit the framework to the rules and constraints of the style
  - Do not let a particular target application's needs creep into the framework
  - "Rule of three" for applications

New Framework Guidelines (cont'd)

- Choose the framework scope
  - A framework does not necessarily have to implement all possible stylistic advantages (e.g., dynamism or distribution)
- Avoid over-engineering
  - Don't add capabilities simply because they are clever or "cool", especially if known target applications won't use them
  - These often add complexity and reduce performance

Software Architecture Foundations, Theory, and Practice

New Framework Guidelines (cont'd)

- Limit overhead for application developers
  - Every framework induces some overhead (classes must inherit from framework base classes, communication mechanisms limited)
  - Try to put as little overhead as possible on framework users
- Develop strategies and patterns for legacy systems and components
  - Almost every large application will need to include elements that were not built to work with a target framework
  - Develop strategies for incorporating and wrapping these

Concurrency

- Concurrency is one of the most difficult concerns to address in implementation
  - Introduction of subtle bugs: deadlock, race conditions…
  - Another topic on which there are entire books written
- Concurrency is often an architecture-level concern
  - Decisions can be made at the architectural level
  - Done carefully, much concurrency management can be embedded into the architecture framework
- Consider our earlier example, or how pipe-and-filter architectures are made concurrent without direct user involvement

Generative Technologies

- With a sufficiently detailed architectural model, various implementation artifacts can be generated
  - Entire system implementations
    - Requires extremely detailed models including behavioral specifications
    - More feasible in domain-specific contexts
  - Skeletons or interfaces
    - With detailed structure and interface specifications
  - Compositions (e.g., glue code)
    - With sufficient data about bindings between two elements

Maintaining Consistency

- Strategies for maintaining one-way or round-trip mappings
  - Create and maintain traceability links from architectural implementation elements
    - Explicit links in a database, in architectural models, in code comments can all help with consistency checking
  - Make the architectural model part of the implementation
    - When the model changes, the implementation adapts automatically
    - May involve "internal generation"
  - Generate some or all of the implementation from the architecture