Architectures in Context

Software Architecture
Chapter 2

Fundamental Understanding

- Architecture is a set of principal design decisions about a software system
- Three fundamental understandings of software architecture
  - Every application has an architecture
  - Every application has at least one architect
  - Architecture is not a phase of development

Every Application Has an Architecture

- By architecture we mean the set of principal design decisions made about a system
- The architecture of the WWW is based on the REST architectural style
- The architecture of a Unix shell is based on the pipe-and-filter style
- All applications have (good or bad) architectures because they all result from key design decisions
    - Where did the architecture of an app come from?
    - How can this architecture be characterized?
    - What are its properties? Is it a “good” or ”bad” one?

Every Application Has at Least One Architect

- The architect is the person, or in most cases, group who
  - Makes the principal decisions about the application
  - Establishes and (it is hoped) maintains the foundational design
- Were the architects always aware when they had made a fundamental design decision?
- Can they maintain the conceptual integrity of the design over time?
- Were alternatives considered at the various decision points?

Architecture is not a Phase of Development

- In a simplistic and inaccurate understanding, the architecture is a specific product of a particular phase in the development process, that is after requirements analysis and before the detailed design
- Treating architecture as a phase denies its foundational role in software development and confines architectures to consist of only a few design decisions
- More than “high-level design”, ”product design”, etc.
- Architecture is also represented, e.g., by object code, source code, …

Context of Software Architecture

- Requirements
- Design
- Implementation
- Analysis and Testing
- Evolution
- Development Process

Requirements Analysis

- Traditional SE suggests requirements analysis should remain unsullied by any consideration for a design
- However, without reference to existing architectures it becomes difficult to assess practicality, schedules, or costs
  - In building architecture we talk about specific rooms…
  - …rather than the abstract concept “means for providing shelter”
- In engineering new products come from the observation of existing solution and their limitations

New Perspective on Requirements Analysis

- Existing designs and architectures provide the solution vocabulary
- Our understanding of what works now, and how it works, affects our wants and perceived needs
- The insights from our experiences with existing systems
  - Help us imagine what might work and
  - Enable us to assess development time and costs
- → Requirements analysis and consideration of design must be pursued at the same time

Non-Functional Properties (NFPs)

- NFPs are the result of architectural choices
- NFP questions are raised as the result of architectural choices
- Specification of NFP might require an architectural framework to even enable their statement
- An architectural framework will be required for assessment of whether the properties are achievable

Core Observations

- Existing designs and architectures provide the vocabulary for talking about what might be
- Our understanding of what works now, and how it works, affects our wants and perceived needs, typically in very solution-focused terms
- The insights from our experiences with existing systems helps us imagine what might work and enables us to assess, at an early stage, how long we must be willing to wait for it, and how much we will need to pay for it

Design and Architecture

- The traditional design phase is not exclusively "the place or the time" when a system's architecture is developed–for that happens over the course of development–but it is a time when particular emphasis is placed on architectural concerns
- Since principal design decisions are made throughout development, designing must be seen as an aspect of many other development activities
- Architectural decisions are of many different kinds, requiring a rich repertoire of design techniques

Design and Architecture (cont’d)

- Design is an activity that pervades software development
- It is an activity that creates part of a system’s architecture
- Typically, in the traditional design phase, decisions concern
  - A system’s structure
  - Identification of its primary components
  - Their interconnections
- Architecture denotes the set of principal design decisions about a system
  - That is more than just structure

Architecture-Centric Design

- Traditional design phase suggests translating the requirements into algorithms, so a programmer can implement them
- Architecture-centric design
  - Stakeholder issues (e.g., use of open-source s/w)
  - Decision about use of COTS component
  - Overarching style and structure
  - Types of connectors for composing sub-elements
  - Package and primary class structure
  - Security and other non-functional properties
  - Deployment and post implementation issues

Design Techniques

- Basic conceptual tools
  - Separation of concerns
  - Abstraction
  - Modularity
- Two illustrative widely adapted strategies
  - Object-oriented design
  - Domain-specific software architectures (DSSA)

Object-Oriented Design (OOD)

- Objects
  - Main abstraction entity in OOD
  - Encapsulations of state with functions for accessing and manipulating that state
  - Numerous variations are found regarding the way objects are specified, related to one another, created, destroyed, etc.

Pros and Cons of OOD

- Pros
  - UML modeling notation (with simple notions of SA)
  - Design patterns that "codify" prior acquired know-how
- Cons
  - Provides only
    - One level of encapsulation (the object)
    - One notion of interface
    - One type of explicit connector (procedure call)
      - Even message passing is realized via procedure calls
    - No real notion of structure
  - OO programming languages might dictate important design decisions
  - OOD assumes a shared address space

Domain-Specific Software Architectures (DSSA)

- Capturing and characterizing the best solutions and best practices from past projects within a domain
- Production of new applications can focus on the points of novel variation
- Reuse applicable parts of the architecture and implementation
- Good technical support is required: the architecture of the previous generation of apps must be captured and refined for reuse
- Applicable for product lines
  - □ →Recall the Philips Koala example

Implementation

- The objective is to create machine-executable source code
  - That code should be faithful to the architecture
    - Alternatively, it may adapt the architecture
    - How much adaptation is allowed?
    - Architecturally-relevant vs. -unimportant adaptations
  - It must fully develop all outstanding details of the application

Faithful Implementation

- All of the structural elements found in the architecture are implemented in the source code
- Source code must not utilize major new computational elements that have no corresponding elements in the architecture
- Source code must not contain new connections between architectural elements that are not found in the architecture
- Is this realistic?
  Overly constraining?
  What if we deviate from this?

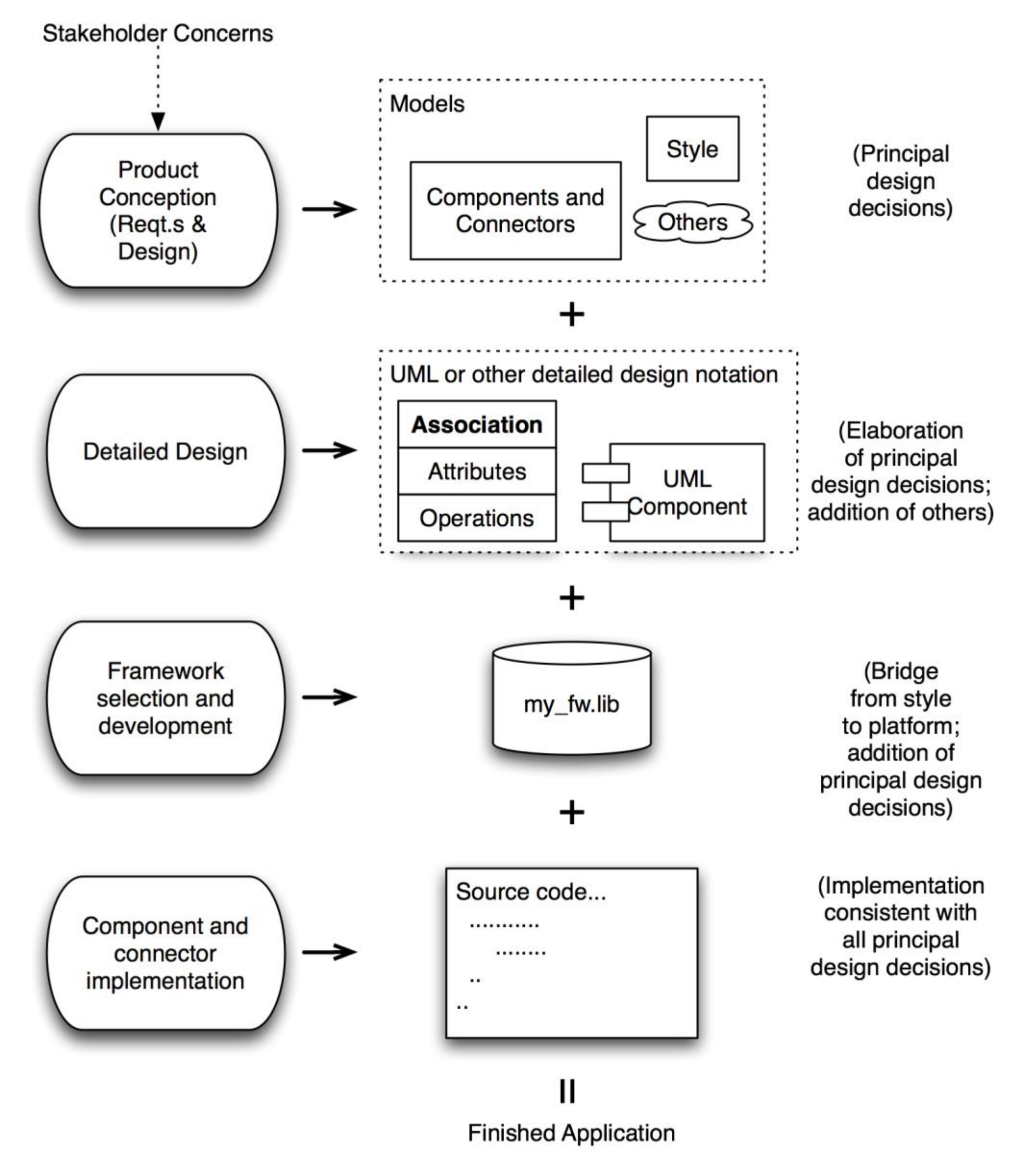
Unfaithful Implementation

- The implementation does have an architecture
  - It is latent, as opposed to what is documented
- Failure to recognize the distinction between planned and implemented architecture
  - Robs one of the ability to reason about the application’s architecture in the future
  - Misleads all stakeholders regarding what they believe they have as opposed to what they really have
  - Makes any development or evolution strategy that is based on the documented (but inaccurate) architecture doomed to failure

Implementation Strategies

- Generative techniques
  - E.g., parser generators
- Frameworks
  - Collections of source code with identified places where the engineer must “fill in the blanks”
- Middleware
  - CORBA, DCOM, RPC, …
- Reuse-based techniques
  - COTS, open-source, in-house
- Writing all code manually

How it all Fits Together


Stakeholder Concerns
Product Conception (Reqt.s & Design)
Models
Style
Components and Connectors
Others
(Principal design decisions)
+
Detailed Design
UML or other detailed design notation
Association
Attributes
Operations
UML Component
(Elaboration of principal design decisions; addition of others)
+
Framework selection and development
my_fw.lib
(Bridge from style to platform; addition of principal design decisions)
+
Component and connector implementation
Source code...
....
....
..
..
(Implementation consistent with all principal design decisions)
II
Finished Application

Analysis and Testing

- Analysis and testing are activities undertaken to assess the qualities of an artifact
- Traditionally, this is done after the code has been written
- The code is tested for functional correctness but occasionally also for NFPs (e.g., performance)
- The earlier an error is detected and corrected the lower the aggregate cost
- Rigorous representations are required for analysis, so precise questions can be asked and answered
    - Preferably by means of automated analysis aids

Analysis of Architectural Models

- A formal architectural model can be examined for internal consistency and correctness
- An analysis on a formal model can reveal
  - Component mismatch
  - Incomplete specifications
  - Undesired communication patterns
  - Deadlocks
  - Security flaws
- It can be used for size and development time estimations

Benefits of the Analysis of Architectural Models

- The structural architecture of an application can be examined for consistency, correctness and exhibition of desired nonfunctional properties
- The architectural model may be examined for consistency with requirements
- The architectural model may be used in determining and supporting analysis and testing strategies applied to the source code
- The architectural model can be compared to a model derived from the source code of an application

Evolution and Maintenance

- All activities that chronologically follow the release of an application
- Software will evolve
  - Regardless of whether one is using an architecture-centric development process or not
- The traditional software engineering approach to maintenance is largely ad hoc
  - Risk of architectural decay and overall quality degradation
- Architecture-centric approach
  - Sustained focus on an explicit, substantive, modifiable, faithful architectural model

Architecture-Centric Evolution Process

- Motivation (e.g., creating new versions of a product)
- Evaluation or assessment: the proposed change, as well as the existing application, must be examined to determine, for example, whether the desired change can be achieved and, if so, how
  - If an explicit architectural model that is faithful to the implementation is available, this step becomes easier
- Development of an approach by choosing among alternatives
- Once an approach has chosen, it is put into action, maintaining consistency between the SA and the code

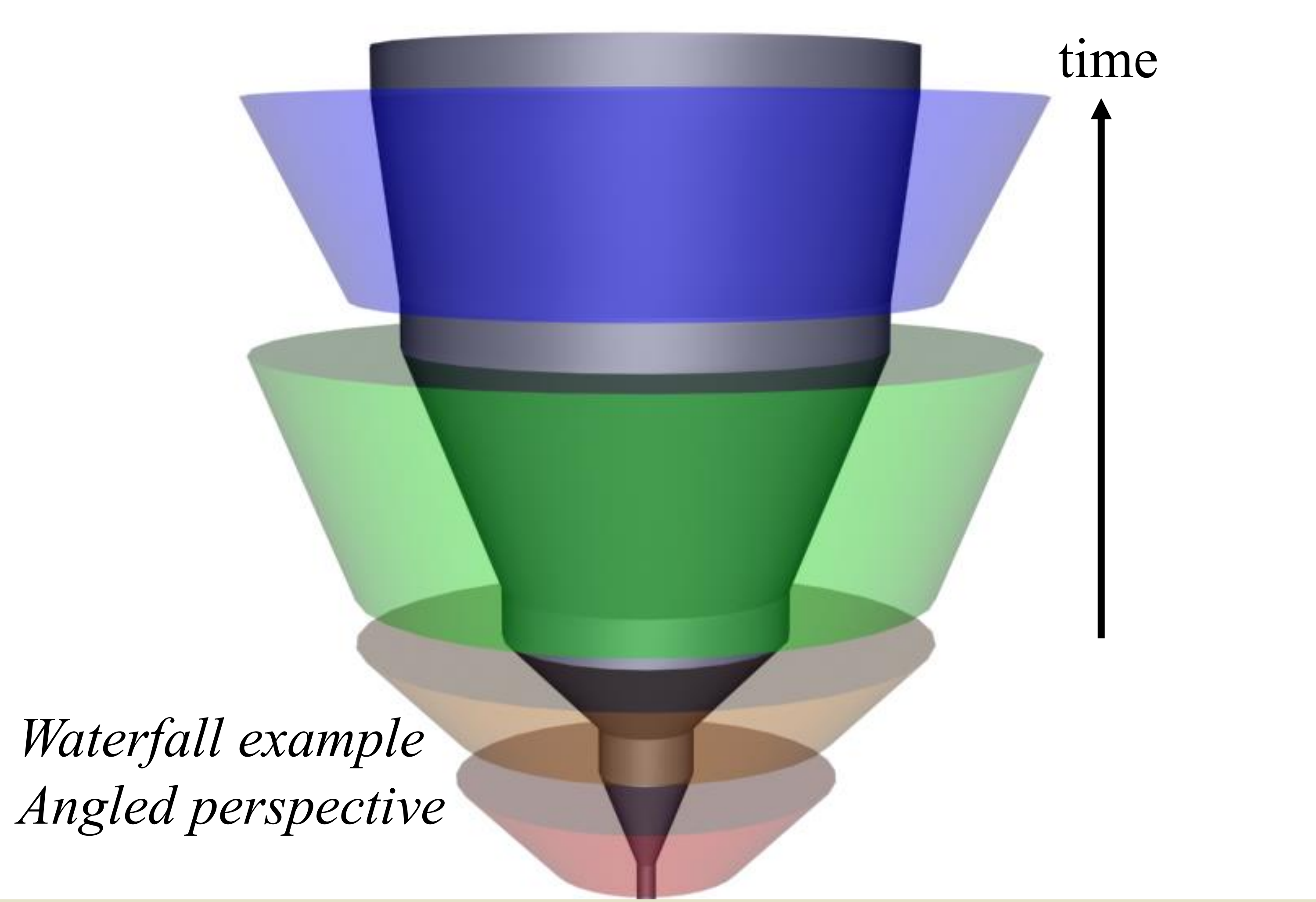
Processes

- Traditional software process discussions make the process activities the focal point
- In architecture-centric software engineering the product becomes the focal point
- No single "right" software process for architecture-centric software engineering exists
- Comparing, or even understanding, different strategies for software development, requires a means for describing these strategies
- A good descriptive formalism will also give effective prominence to the central role of the product's SA

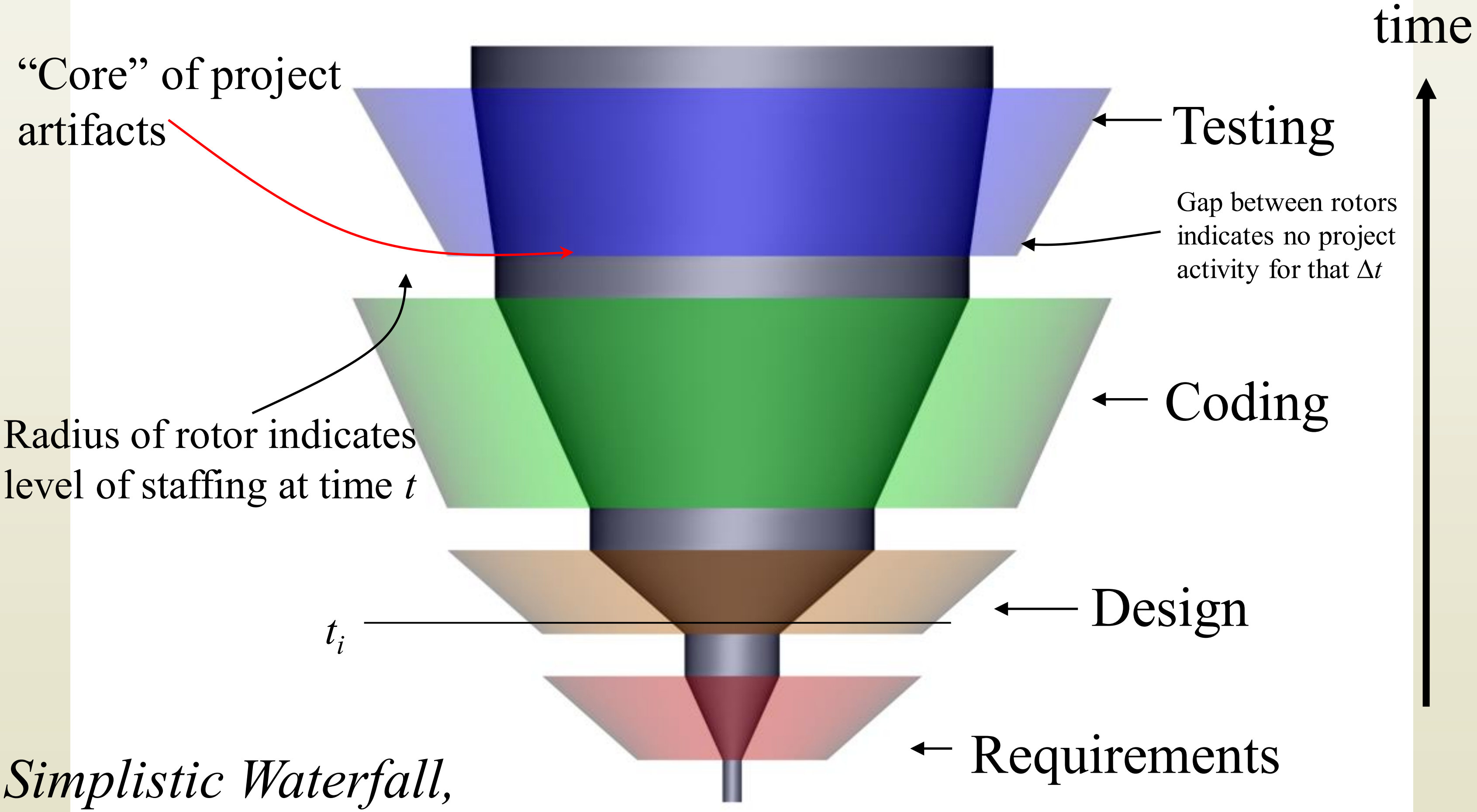
Turbine – A New Visualization Model

- Goals of the visualization
  - Provide an intuitive sense of
    - Project activities at any given time
      - Including concurrency of types of development activities
    - The "information space" of the project
    - Effort (e.g., labor hours expended) at any given time
    - Product state, e.g., total content of product development
  - Show centrality of the products
    - (Hopefully) Growing body of artifacts
    - Allow for the centrality of architecture
      - But work equally well for other approaches, including "dysfunctional" ones
  - Effective for indicating time, gaps, duration of activities
  - Investment (cost) indicators

The Turbine Model



The Turbine Model



time

“Core” of project artifacts

Testing

Gap between rotors indicates no project activity for that Δt

Coding

Radius of rotor indicates level of staffing at time t

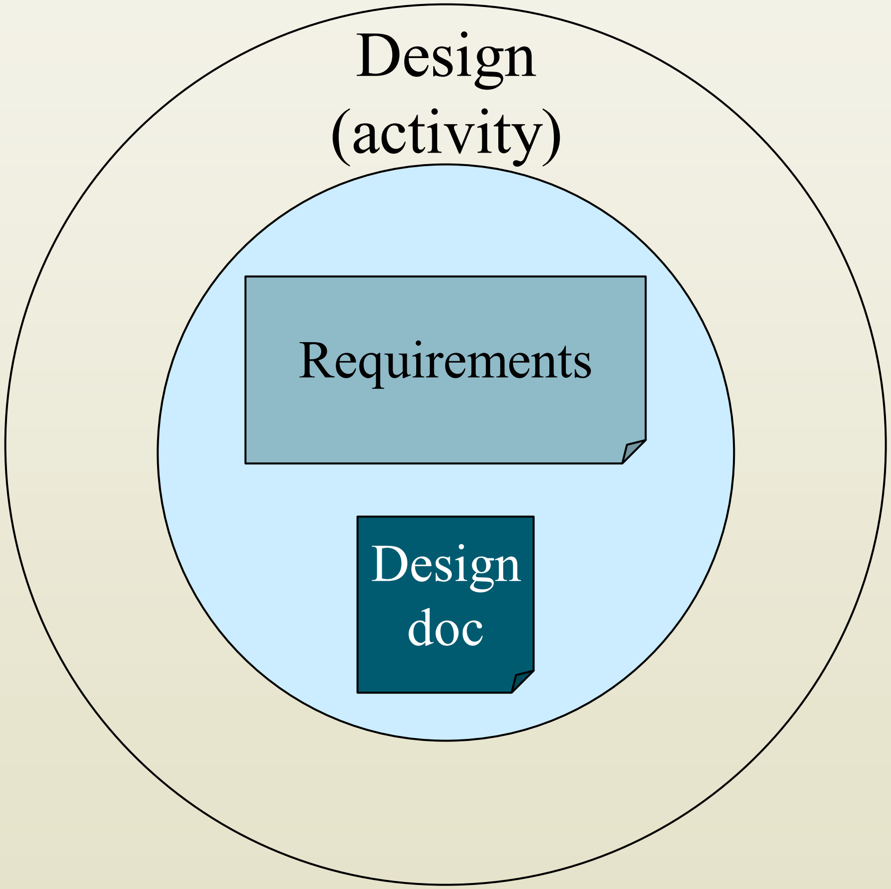
Design

t_i

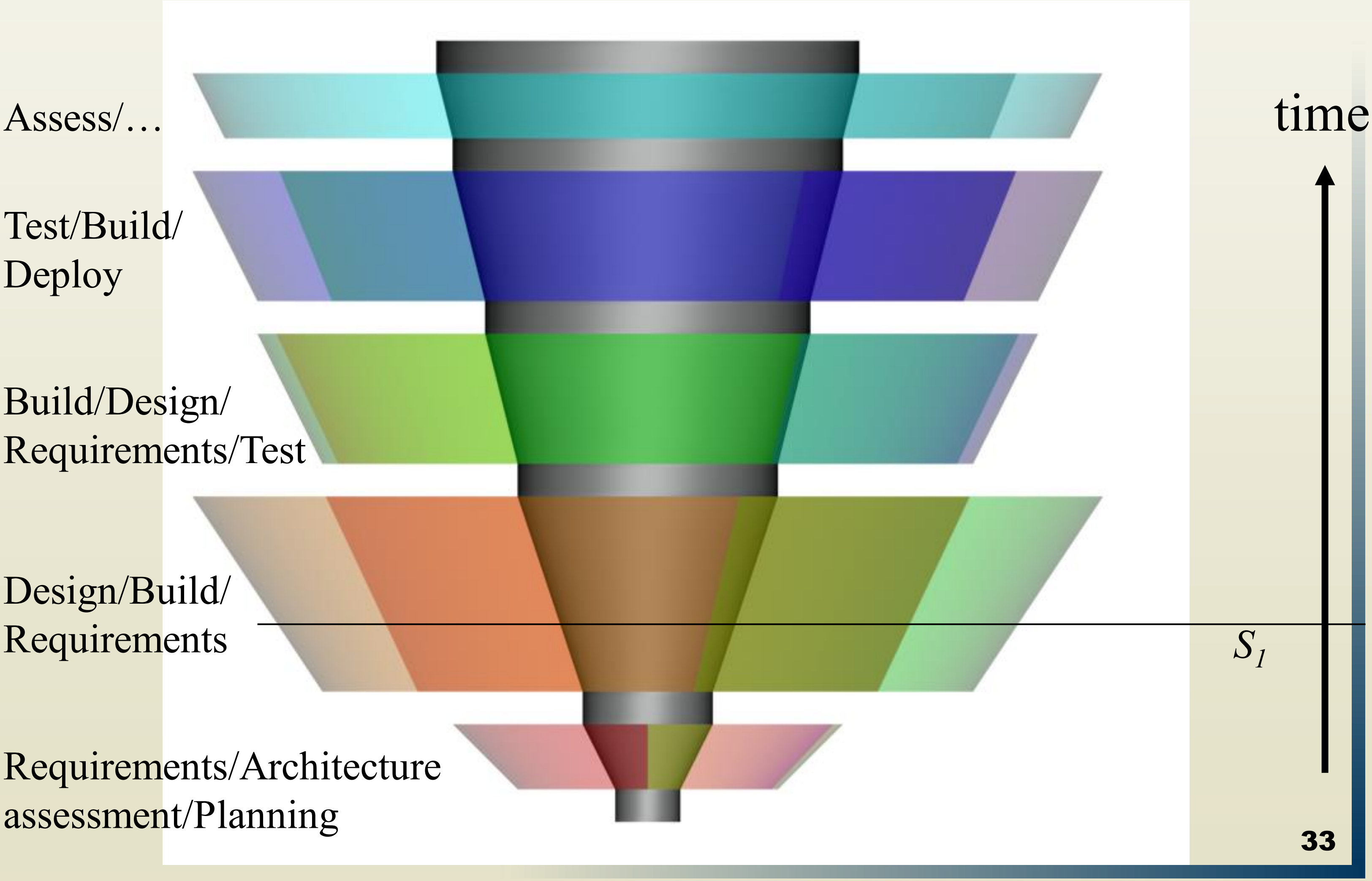
Requirements

Simplistic Waterfall, Side perspective

Cross-section at time t_i



A Richer Example



Assess/…

Test/Build/
Deploy

Build/Design/
Requirements/Test

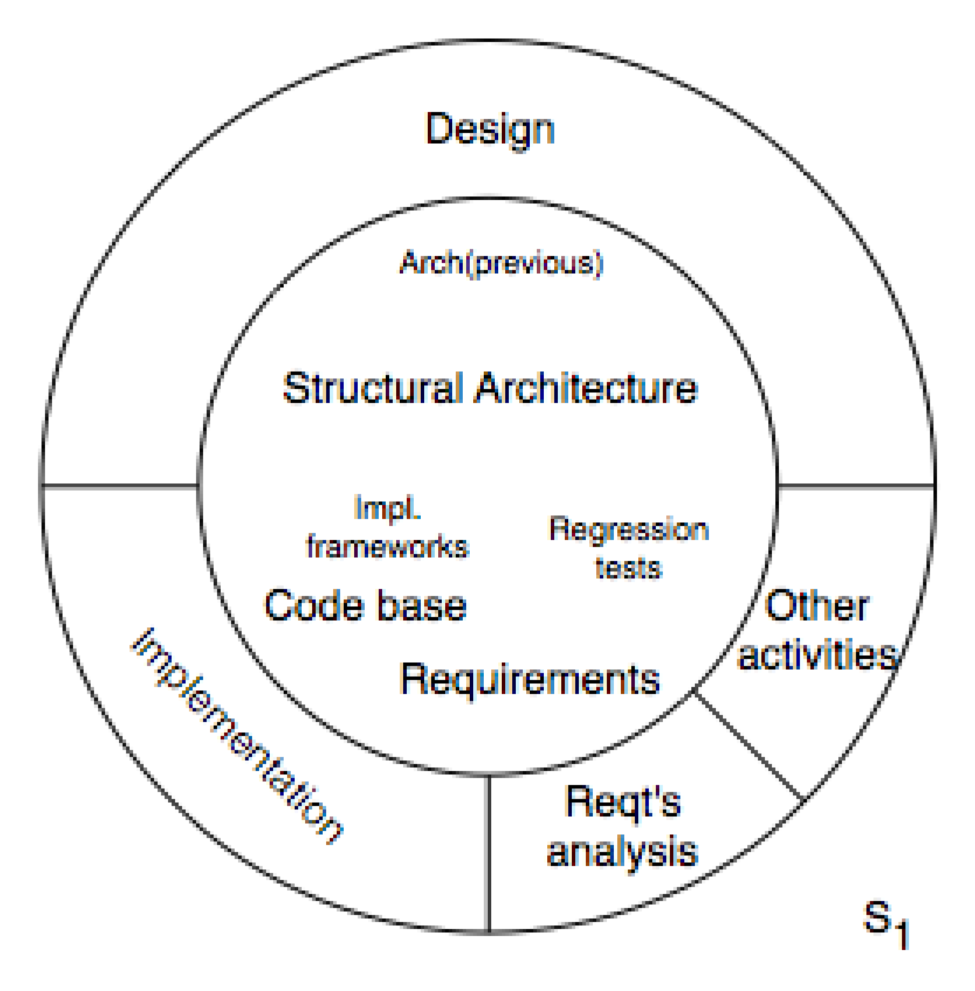
Design/Build/
Requirements

Requirements/Architecture
assessment/Planning

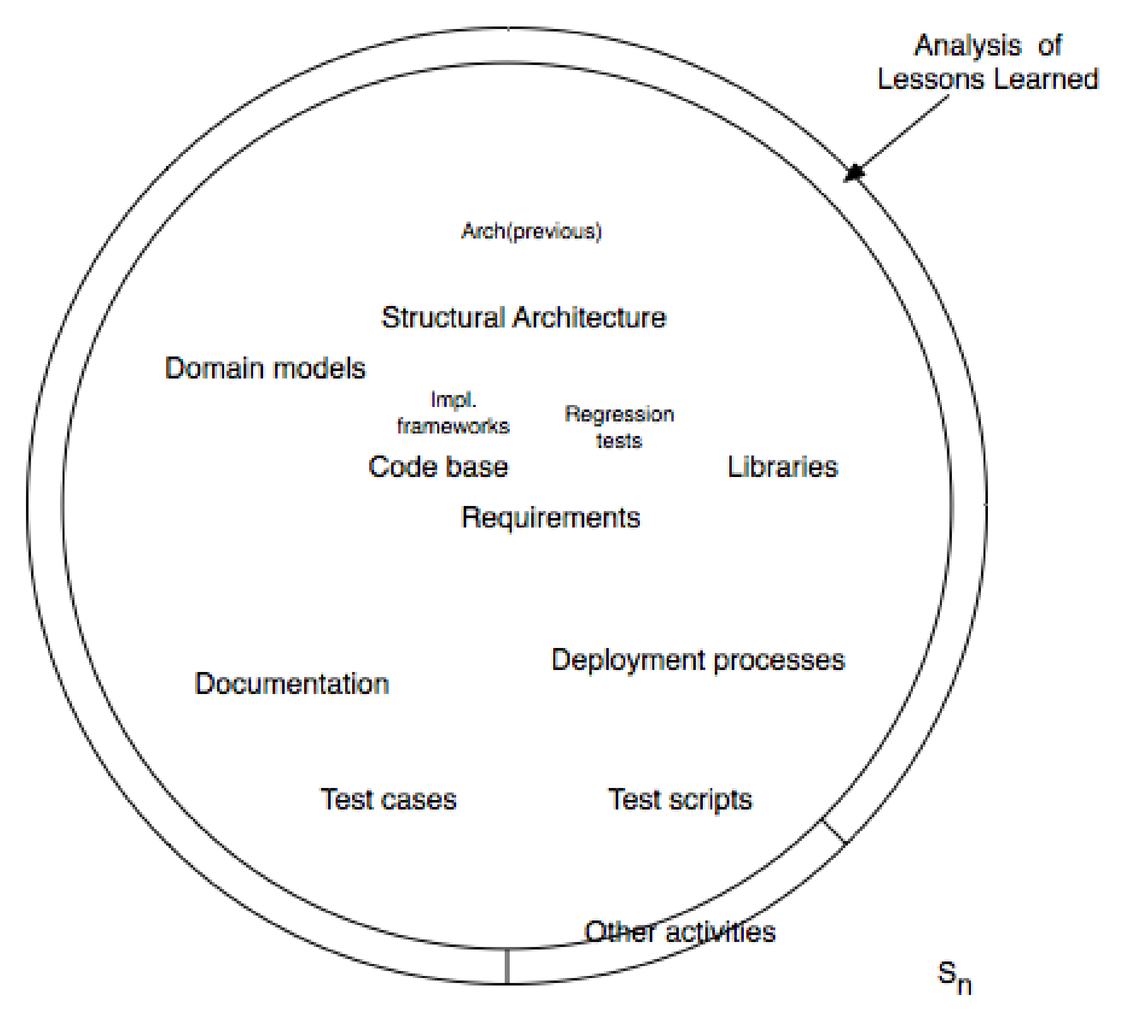
time

S_1

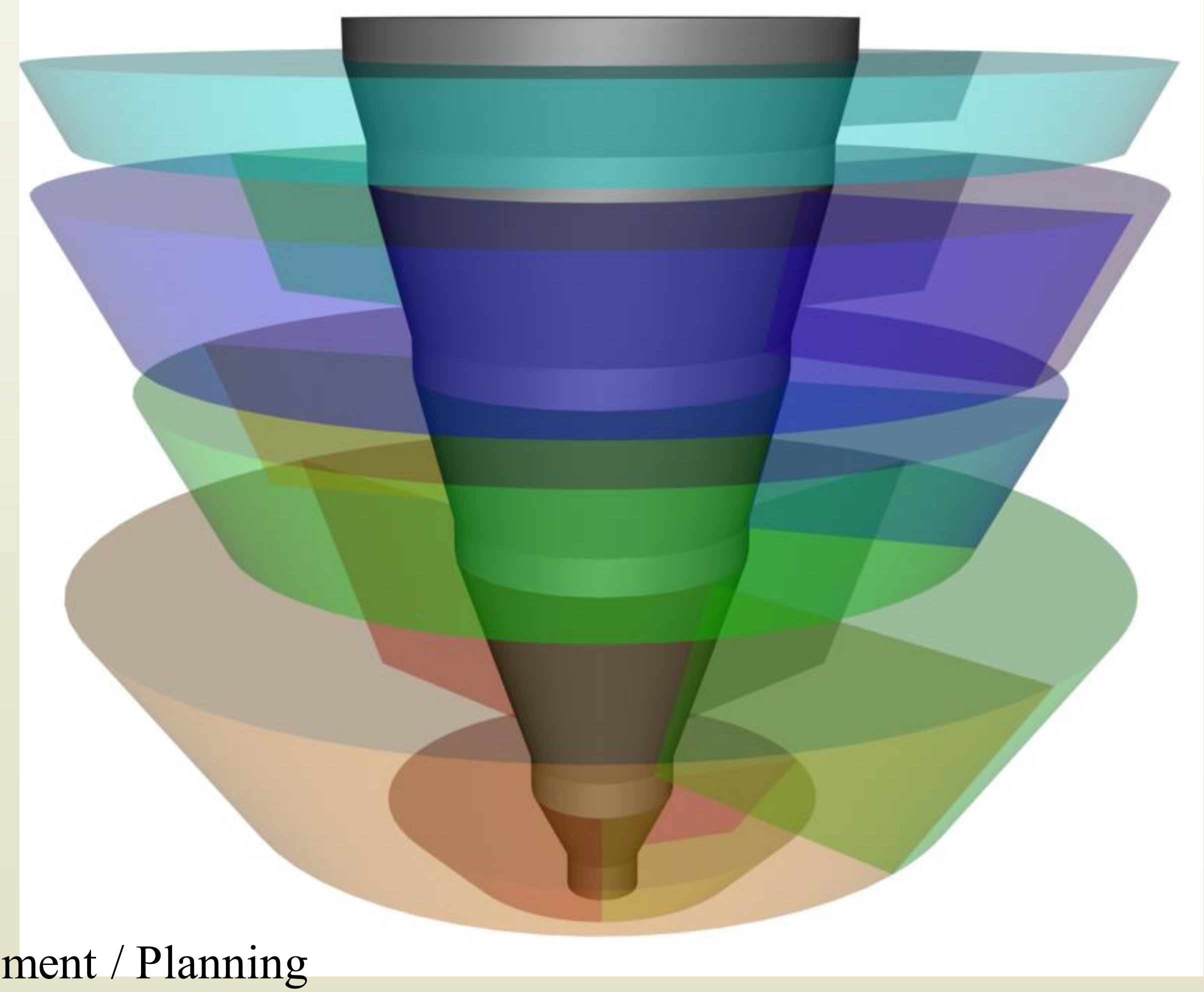
A Sample Cross-Section



A Cross-Section at Project End



Volume Indicates Where Time was Spent



Assess/…

Test/Build/
Deploy

Build/Design/
Requirements/Test

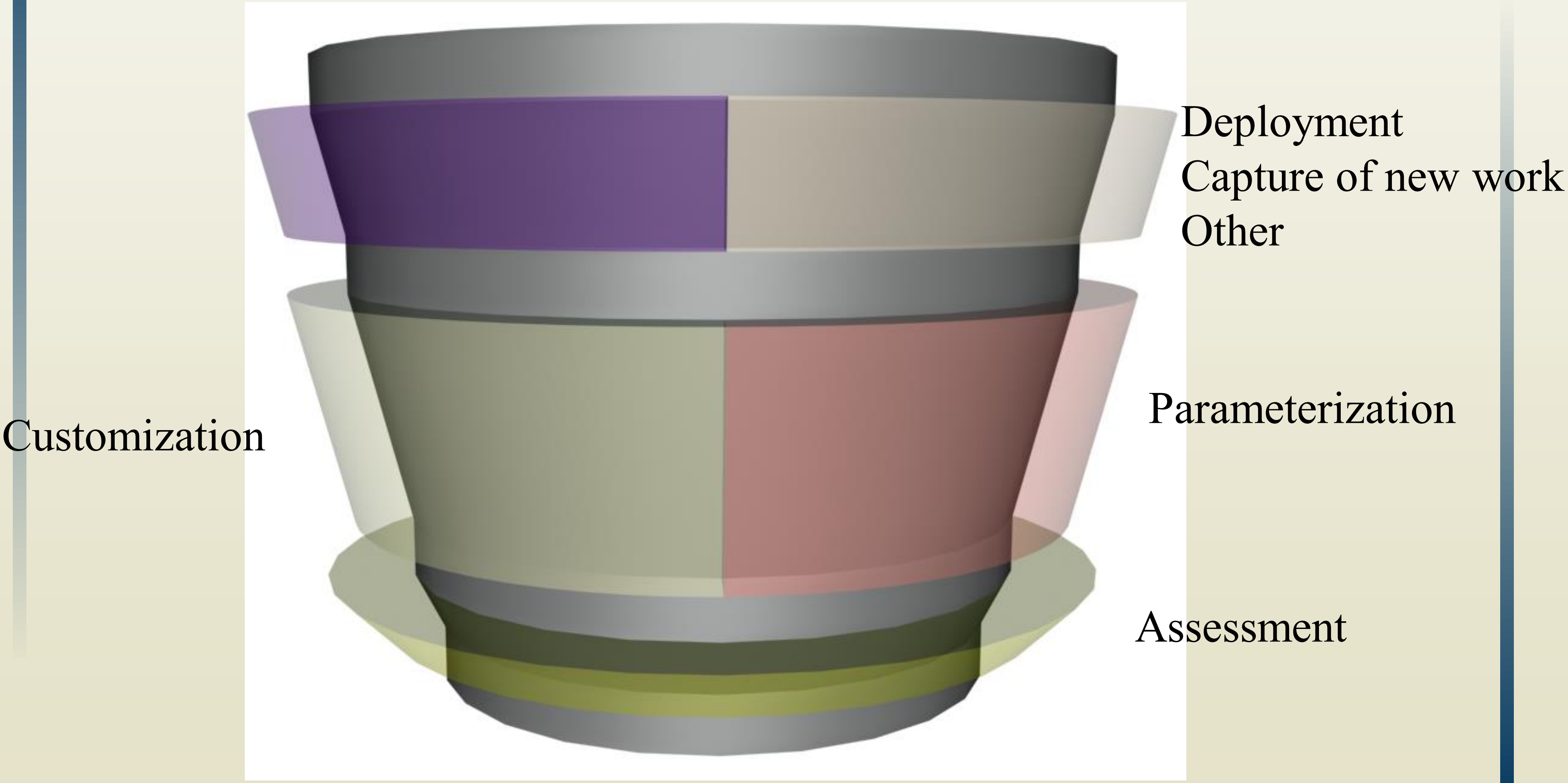
Design/Build/
Requirements

Requirements/
Architecture Assessment / Planning

Two Further Examples of the Use of the Turbine Model

- A technically strong product-line
  - The core is quite large right from the beginning, as it contains a multitude of reusable artifacts from preceding projects
  - The activities here are to
    - Assess the artifacts as to whether they are appropriate for the new product
    - Parameterize them to meet the new project's needs and perform any necessary customization
    - Integrate, assess, and deploy the product

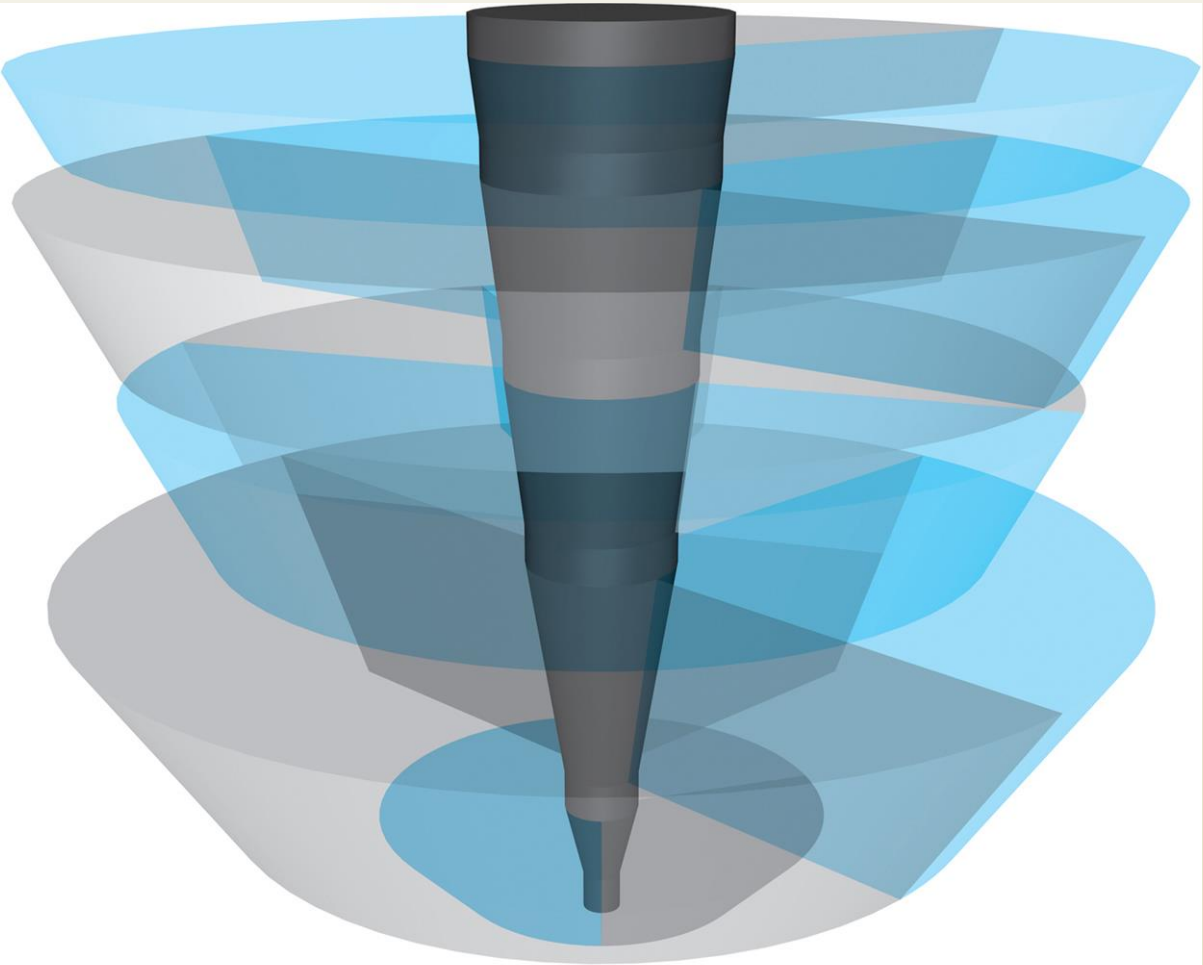
A Technically Strong Product-Line Project



Two Further Examples of the Use of the Turbine Model (cont’d)

- Agile development
  - Agile processes demonstrate and emphasize concurrency between a variety of kinds of development activities (e.g., requirements elicitation)
  - All the activities continue throughout the project
  - However, the skinny core indicates that the agile process denies development of any explicit architecture; rather the code is the architecture
  - The agile process starts with a core that is devoid of any architecture and terminates similarly

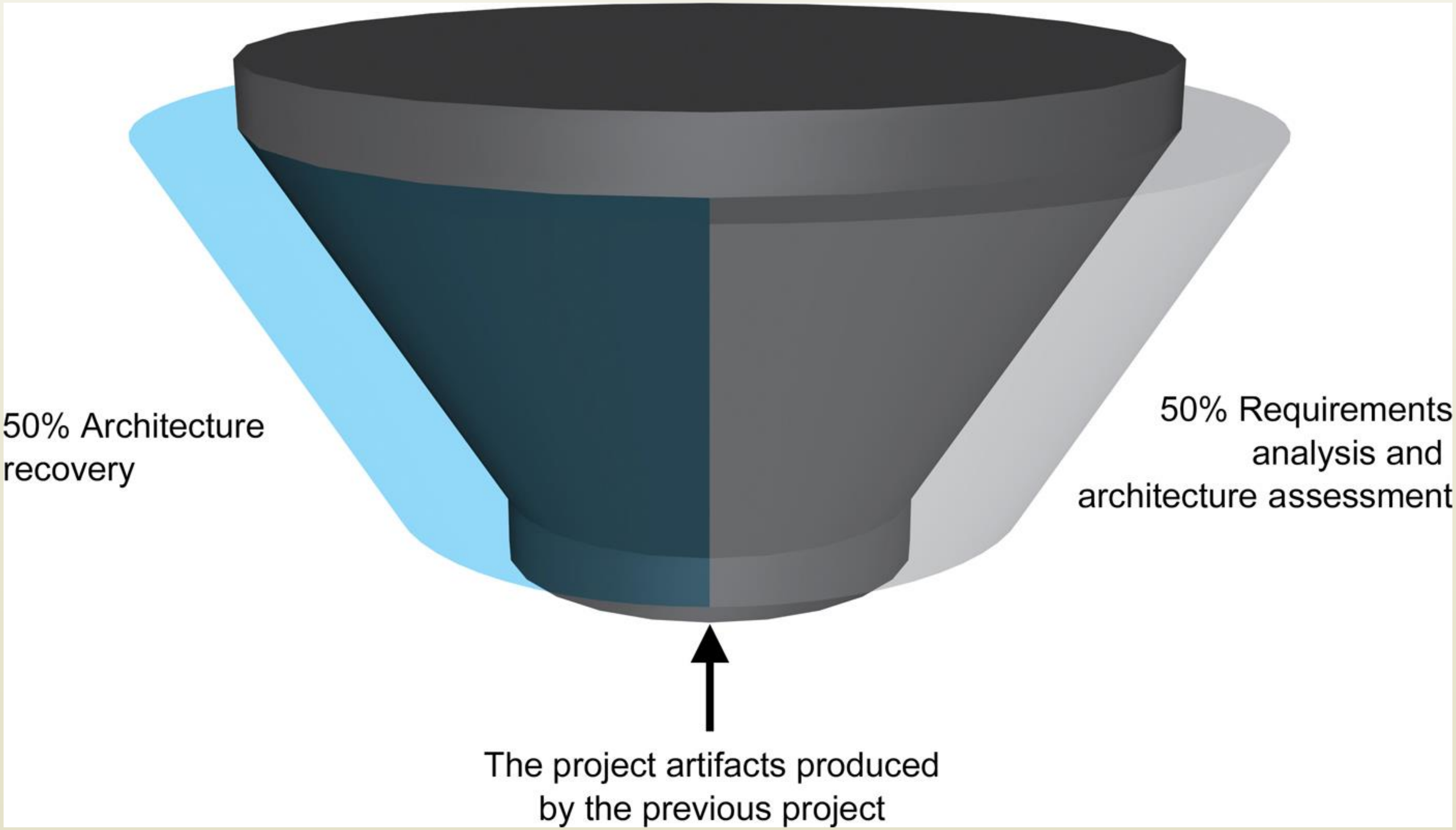
Turbine Visualization of an Agile Development Process



Two Further Examples of the Use of the Turbine Model (cont’d)

- Agile development (cont’d)
  - The problems with this approach are made clear when a follow-on project is required in order to meet new demands
  - Unless the same development team is employed, a significant ring of activity will be required at the beginning of the project to simply understand the existing code base and the latent architecture, so that planning on how to meet the new needs can proceed

Initial Portion of a Turbine Model of a Phase-2 Agile Process


50% Architecture recovery
50% Requirements analysis and architecture assessment
The project artifacts produced by the previous project

Visualization Summary

- It is illustrative, not prescriptive
- It is an aid to thinking about what’s going on in a project
- Can be automatically generated based on input of monitored project data
- Can be extended to illustrate development of the information space (artifacts)
  - The preceding slides have focused primarily on the development activities

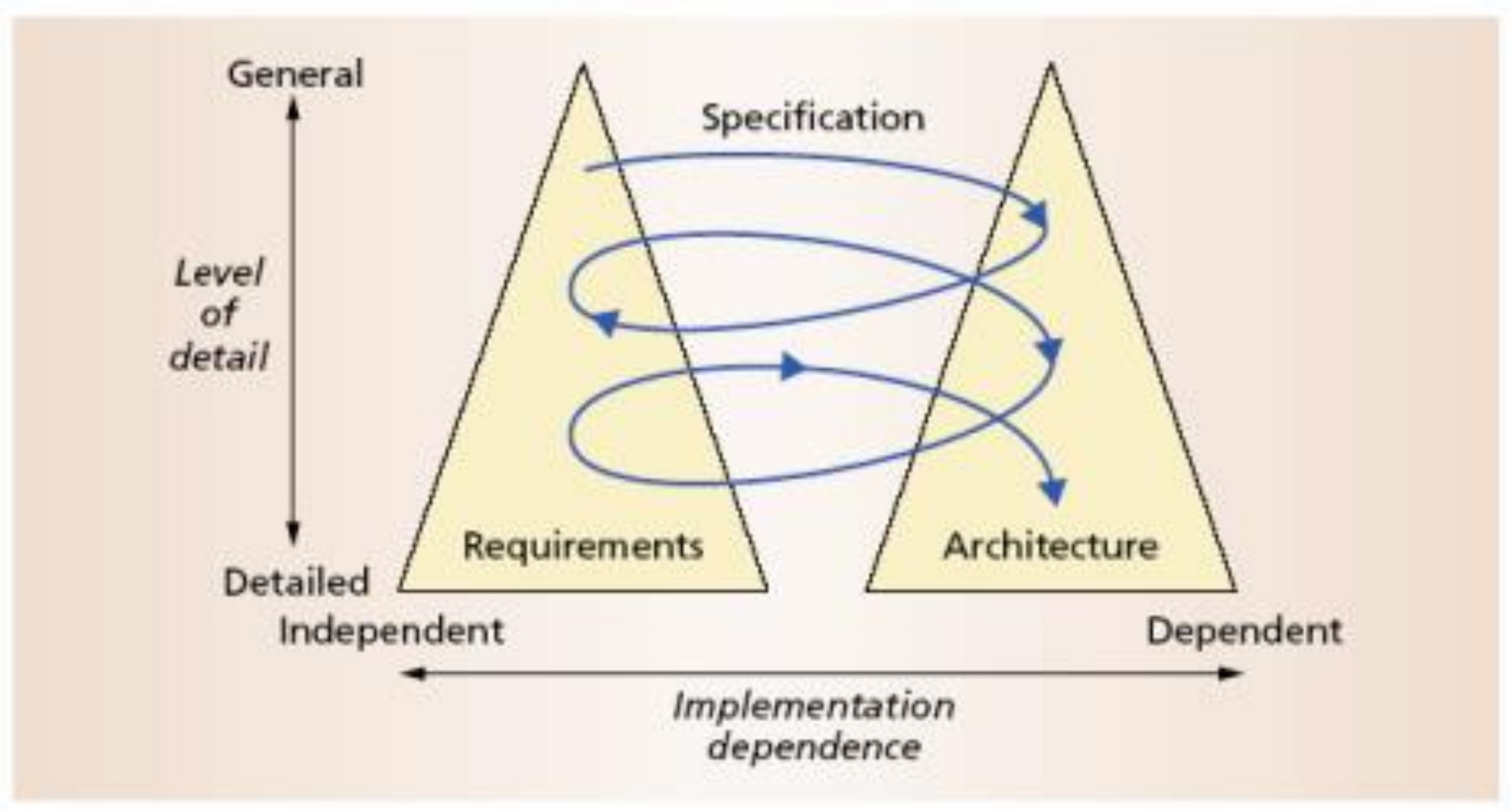
Processes Possible in this Model

- Traditional, straight-line waterfall
- Architecture-centric development
- DSSA-based project
- Agile development
- Dysfunctional process

Other Processes and Process Models

- One of the best is the Twin Peaks model by Bashar Nuseibeh
- It emphasizes the co-development of requirements and architectures, incrementally elaborating details
- It is representative of recent work in requirements engineering that is now giving much more prominence to the role of design and existing architectures in the activity of product conception

The Twin Peaks Model



Summary

- A proper view of software architecture affects every aspect of the classical software engineering activities
- The requirements activity is a co-equal partner with design activities
- The design activity is enriched by techniques that exploit knowledge gained in previous product developments
- The implementation activity
  - Is centered on creating a faithful implementation of the architecture
  - Utilizes a variety of techniques to achieve this in a cost-effective manner

Summary (cont’d)

- Analysis and testing activities can be focused on and guided by the architecture
- Evolution activities revolve around the product’s architecture
- An equal focus on process and product results from a proper understanding of the role of software architecture