Temel Kavramlar
Soyutlama (Abstraction)

- Bir şeyin en önemli, temel veya ayırt edici taraflarını öne çıkarıp daha az önemli, önemsiz veya saptırıcı ayrıntılarını bastıran veya görmezden gelen herhangi bir model.
- Ortak noktaları vurgulamak için farklılıkların kaldırılmasının neticesi.

Soyutlama (Abstraction) Nedir?

- Bir varlığın onu diğerlerinden ayıran temel özellikleridir.
- Bakanın perspektifine göre bir sınır çizer.
- Somut bir görünüm olmayıp bir şeyin mükemmel özünü ortaya koyar.

Abstraction bize ne sunar

- Soyutlama, bir varlığı diğer tüm varlık türlerinden ayıran temel özelliklerine odaklanarak karmaşıklığı yönetmemizi sağlar.
- Bir soyutlama, alana ve perspektife bağlıdır.
- Bir bağlamda önemli olan başka bir bağlamda önemsiz olabilir.
- OO, soyutlamalar kullanarak, problem alanından sistemimizi modellememize izin verir.(örneğin, sınıflar ve nesneler).

Abstraction örnekler

- Öğrenci, üniversitedeki sınıflara kayıtlı kimsedir.
- Profesör, üniversitede ders veren kimsedir.
- Ders, üniversite tarafından sunulan bir derstir.
- Ders programı, haftanın günleri ve saatleri dahil olmak üzere kurs için özel bir zamandır.

Soyutlama

- Öncelikli hedefi çıkarım/istidlal yapılarının (inference structures) geliştirilmesi olan matematik için
- enformasyonu/malumatı dikkate almamaktır. (information ignoring)
- Öncelikli hedefi etkileşim kalıplarının (interaction patterns) geliştirilmesi olan bilgisayar biliminde ise
- Enformasyonu kapamadır(örtmedir) (information hiding).
- Bilgisayar biliminin temel faaliyeti yazılım üretimidir ve bu faaliyet, esas olarak çıkarım yapılarının yaratılması ve kullanılmasıyla değil, etkileşimin modellemesiyle kendini gösterir.

Abstraction C. Sciences
Bilgisayar bilimi tarihi boyunca, soyutlama, önde gelen *bilgisayar bilimcileri* tarafından bilgisayar bilimlerinin temeli olarak görülmüştür.

> "[. . .] soyutlama gücü, yetkin bir programcının hayati faaliyetlerinden biri olarak görülmelidir "[33, s. 864].
> [33] Edsger W. Dijkstra. 1972. The humble programmer. 15, 10 (1972), 859–866. Communications of the ACM Statter)

Aho ve Ullman, *bilgisayar bilimini* bir soyutlama bilimi olarak tanımladılar :
"bir problem hakkında düşünmek için doğru modeli yaratmak ve onu çözmek için uygun teknikleri tasarlamak" (Alfred V. Aho and Jeffrey D. Ullman. 1992. Foundations of Computer Science. Computer Science Press, New York, NY. 1992).

Bilgisayar bilimcileri, soyutlama düzeyleri arasında hareket etme yeteneğinin bilgisayar bilimcileri ve bilgisayar bilimi uygulayıcıları için esas olduğunu vurgulamışlardır. Örneğin, Knuth, "doğal bilgisayar bilimcileri"ne atfen

- "onlar, soyutlama düzeylerini hızla değiştirebilen, eşyayı aynı anda "büyük" ve "küçük "olarak gören kimselerdir" der.

Benzer şekilde, Wing'e göre,

- "Bir *bilgisayar bilimcisi* gibi düşünmek [. . .] birden fazla soyutlama düzeyinde düşünmeyi gerektirir ".
- Büyük yazılım sistemlerinin tasarımı ve onaylanmasında kilit sorun, herhangi bir zaman diliminde dikkate alınması gereken karmaşıklık veya ayrıntı miktarını **azaltmaktır.** İki yaygın ve etkili çözüm yöntemi ayrıştırma ve soyutlamadır.

Abstraction in C. Sciences/complexity

- Bir sorunun karmaşıklığı soyutlama yoluyla azaltılabilir.

Ayrıştırma:

- Bir görevi, onu iki veya daha fazla ayrılabilir alt göreve bölerek ayrıştırır.
- Ne yazık ki, birçok sorun için ayrılabilir alt görevler, bütünüyle çekip çevrilme bakımından hala çok karmaşıktır.

Abstraction in C. Sciences/ separation

- Soyutlama, bir modülün amacını uygulamasından ayırır.
- Modülerlik, bir çözümü modüllere ayırır; soyutlama, bir programlama dilinde uygulamadan önce her modülü açıkça belirtir.

Abstraction in C. Sciences/complexity and relevancy

- Soyutlama, bir nesnenin veya olayın niteliklerinden, belirli bir bağlamda ilgili olanları olmayanlardan ayırmak için bir mekanizma sağlayarak, herhangi bir zamanda anlaşılması gereken ayrıntı miktarını azaltmaya hizmet eder.
- "İyi bir soyutlama, okuyucu veya kullanıcı için önemli olan ayrıntıları vurgulayan ve en azından şu an için önemsiz veya dikkat dağıtıcı ayrıntıları bastıran bir soyutlamadır ".
- Soyutlamaların özü, belirli bir bağlamla ilgili bilgileri korumak ve bu bağlamla ilgisi olmayan bilgileri unutmaktır.
- Soyut bir veri yapısı, çeşitli alternatiflerle çalışma yapmaya izin vermek için gerekli esnekliği sağlar.

complexity and separation, proof of correctness

Berzins, Gray ve Naumann şunu ileri sürüyor:

- "Bir kavram, ancak, sonunda onu gerçekleştirmek için kullanılacak mekanizmadan bağımsız olarak tanımlanabildiği, anlaşılabildiği ve analiz edilebildiği takdirde bir soyutlama olarak nitelendirilir".
- karmaşık bir kavramı veya sistemi tanımlamak için bir dizi bağımsız soyutlama oluşturarak ilerletilebilir.

Soyutlama ve Sarmalama tamamlayıcı kavramlardır:

- Soyutlama, bir nesnenin gözlemlenebilir davranışına odaklanırken,
- Sarmalama, bu davranışa yol açan uygulamaya odaklanır.
- Soyutlama "insanların ne yaptıklarına dair düşünmelerine yardımcı olurken"
- Sarmalama "program değişikliklerinin sınırlı çabayla güvenilir bir şekilde yapılmasına izin verir"
- Sarmalama, farklı soyutlamalar arasında açık engeller sağlar ve böylece ilgilerin açık bir şekilde ayrılmasına götürür.

Soyutlama Paradigmasını Neden Seçiyoruz?

1. Karmaşıklığın azaltılması

- kullanıcılar ve uygulayıcılar için bir sorun ve yazılım mühendisliğinde önemli bir hedef –

karmaşık bir kavramı veya sistemi tanımlamak için bir dizi bağımsız soyutlama oluşturarak ilerletilebilir.

2. Örtme

Soyutlama, bir alt sistemin yalnızca dış arayüzlerdeki davranış yönlerini görünür bırakarak iç işleyişini gizlemek için kullanılabilir.

3. Maliyeti Düşürme

Değişiklikler, aksi takdirde olacaklarından daha az modülü etkilediğinden, bağımsız soyutlamaların yazılım onarımı ve geliştirmesi üzerinde etkisi vardır.

4. Soyutlamaların doğruluk ispatında da yeri vardır; Bunlar pratik büyüklükteki sistemler için yönetilebilir iseler karmaşık bir ispatın parçalarını ayırmak için soyutlamalardan faydalanmalıdır.

Software Design

Design is defined as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process" [ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary , ISO/IEC/IEEE, 2010.]

Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction.

A software design (the result) describes the software architecture — that is, how software is decomposed and organized into components — and the interfaces between those components.
It should also describe the components at a level of detail that enables their construction.

- We can analyze and evaluate these models to determine whether or not they will allow us to fulfill the various requirements.
- We can also examine and evaluate alternative solutions and tradeoffs.
- Finally, we can use the resulting models to plan subsequent development activities, such as system verification and validation, in addition to using them as inputs and as the starting point of construction and testing.

Software Design
In a standard list of software life cycle processes, such as that in ISO/IEC/IEEE Std. 12207, Software Life Cycle Processes [2], software design consists of two activities that fit between software requirements analysis and software construction:

- Software architectural design (sometimes called high-level design): develops top-level structure and organization of the software and identifies the various components.
- Software detailed design: specifies each component in sufficient detail to facilitate its construction.

Softwar Design

- Software Design Fundamentals
- Key Issues In Software Design
- Software Structure and Architecture
- User Interface Design
- Software Design Quality Analysis and Evaluation
- Software Design Notations
- Software Design Strategies and Methods
- Software Design Tools

Software Design Fundamentals

- General Design Concepts
- Context of Software Design
- Software Design Process
- Software Design Principles

Key Issues In Software Design

- Concurrency
- Control and Handling of View
- Data Persistence
- Distribution of Components
- Error Exception Handling and Fault Tolerance
- Interaction and Presentation
- Security

Software Structure and Architecture

- Architectural Structures and Viewpoints

- Architectural Styles
- Design Patterns
- Architecture Design Decisions
- Families of Programs and Frameworks

User Interface Design
- General User Interface Design Principles
- User Interface Design Issues
- The Design of User Interaction Modalities
- The Desgin of Information Presentation
- User Interface Design Process
- Localization and Internationalization
- Metaphors and Conceptual Models

Software Design Quality Analysis and Evaluation
- Quality Attributes
- Quality Analysis and Evaluation Techniques
- Measures

Software Design Notations
- Structural Description (static view)
- Behavioral Description (dynamic view)

Software Design Strategies and Methods
- General Strategies
- Function Oriented (structured) Design
- Obejct-Oriented Design
- Data Structure-Centered Design
- Component-based Design (CBD)
- Other Methods

Software Architecture

Jonathan Aldrich, 2004; School of Computer Science; Carnegie Mellon University

What is Software Architecture?

Software architecture represents the high-level design of a software system, showing how desired system properties are achieved

Where Architecture Fits

- Requirements
  - What the system should do
  - What properties it should have
- Architecture
  - High-level design, how properties are achieved
- Detailed design
  - Lower-level design, how system functions
- Code
  - How the system actually works

How does architecture affect system **properties?**

- Modifiability / ease of change
- Consistency of results
- System cost
- Scalability of system
- Reliability of system

Factors to consider :

- Simplicity
- Efficiency
- Security

Architecture is an Abstraction

- Focus on principal design decisions
  - Structure - components and connections
  - Behavior - responsibilities of each component, high level algorithms
  - Interaction - rules governing how components communicate
  - Quality attributes - strategy for achieving
  - Implementation - language, platform, libraries, etc.
  - Any decision that impacts key stakeholder concerns or has global impact on the program

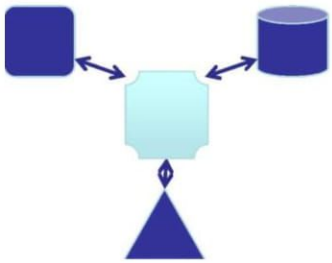
- Elide unimportant details
  - Decisions that are internal to a component
    - i.e. which other components cannot depend on
    - e.g. internal algorithms, data structures, local design patterns
  - AND do not impact key stakeholder concerns

Architecture is design, but not all design is architectural.

Architecture Benefits: System Properties

• Architecture is not about a *system's function*, but rather the *system's properties*
• Some properties and their consequences:
- Fitness: performance, reliability, security → competitive advantage
- Modifiability/ease of changing → business agility
- Reuse of code → reduced cost

Architecture-Organization Alignment



- Conway's Law

Any organization that designs a system...will inevitably produce a design whose structure is a copy of the organization's communication structure (Conway, 1968)

- Case example: product line
- Applications initially developed independently
- Desired reusable library to reduce cost, increase agility
- Failed to build library using existing teams
- Success required a team dedicated to the core library.

How to document a software architecture: **Architectural Views**

• Many possible "views" of architecture
- Implementation structures
  - Modules, packages
  - Modifiability, Independent construction, ...
- Run-time structures
  - Components, connectors
  - Interactions, dynamism, reliability,...
- Deployment structures
  - Hardware, processes, networks
  - Security, fault tolerance, ...

Why Document Architecture?

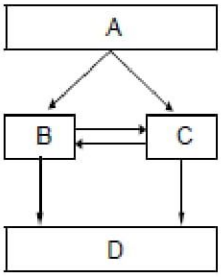
• Blueprint for the system
- Artifact for early analysis
- Primary carrier of quality attributes
- Key to post-deployment maintenance and enhancement

• Documentation speaks for the architect, today and 20 years from today
- As long as the system is built, maintained, and evolved according to its documented architecture

What is Wrong Today?

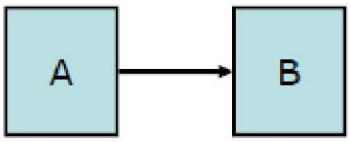
• In practice today's documentation consists of
- Ambiguous box-and-line diagrams

- Inconsistent use of notations
- Confusing combinations of viewtypes



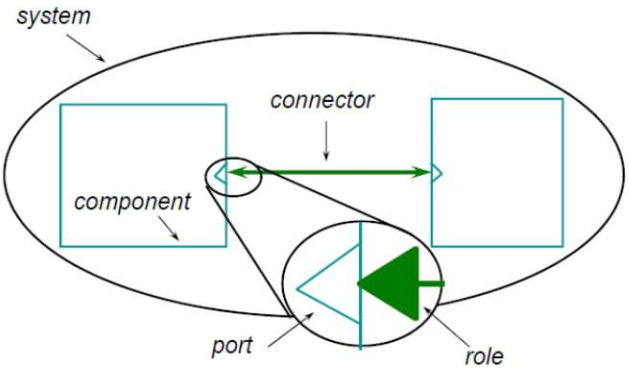
• Many things are left unspecified:
  - What kind of elements?
  - What kind of relations?
  - What do the boxes and arrows mean?
  - What is the significance of the layout?

What could the arrow mean?



• Many possibilities
  - A passes control to B
  - A passes data to B
  - A gets a value from B
  - A streams data to B
  - A sends a message to B
  - A creates B
  - ...

Representing C&C Views



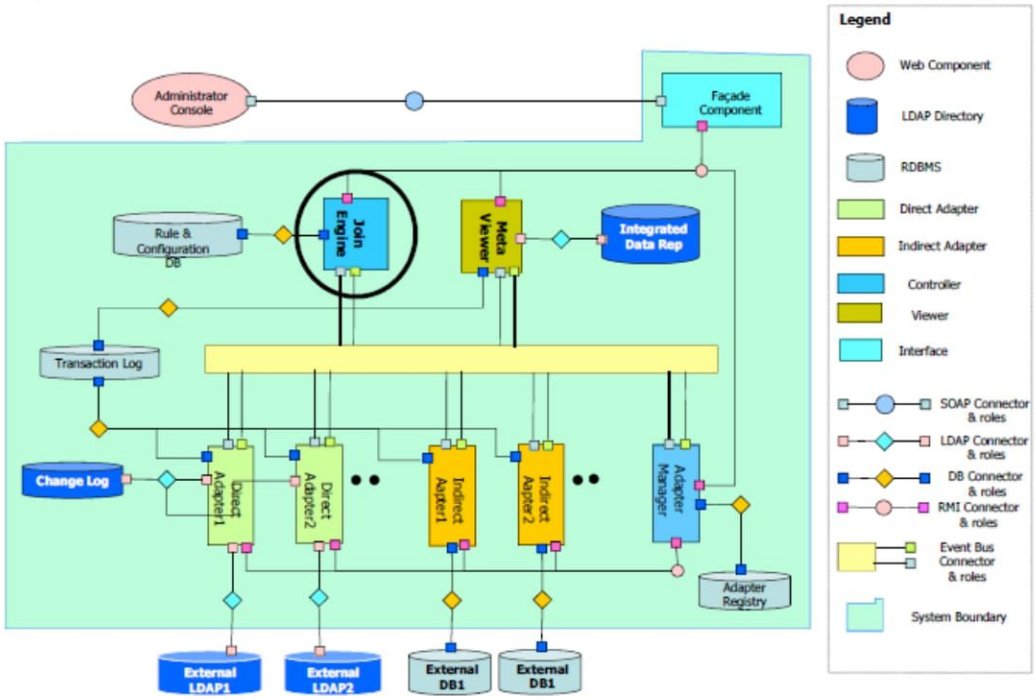
Guidelines: Avoiding Ambiguity

• Always include a legend
• Define precisely what the boxes mean
• Define precisely what the lines mean
• Don't mix viewtypes unintentionally
  - Recall: Module (classes), C&C (components)
• Supplement graphics with explanation
  - Very important: rationale (architectural intent)
• Do not try to do too much in one diagram
  - Each view of architecture should fit on a page
  - Use hierarchy

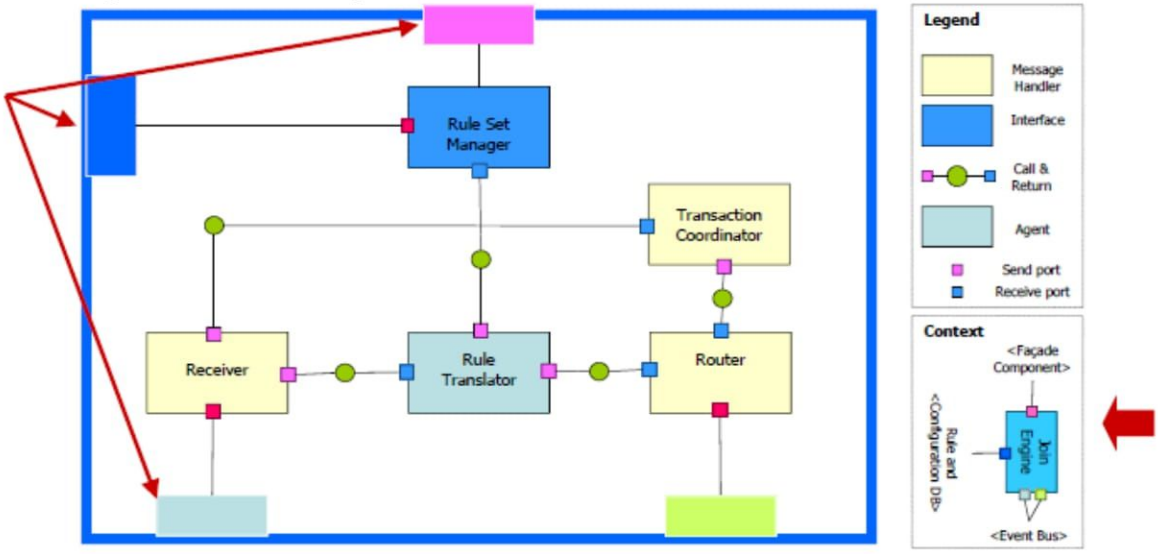
Technique: Hierarchy

- Use hierarchy to define elements in more detail in separate views
- Helps keep an architectural description manageable

Top-level C&C View



Showing Details of Component



Conclusion: Key Takeaways

- Architecture captures high-level design of software
  - Structure and communication
  - Key design decisions
- Enables desired properties of system
  - Reuse → reduce cost
  - Modifiability → business agility
  - Fitness for use → competitive advantage