UNIT II | Reference: Software design, David Budgen, second edition, Pearson education,2003

SOFTWARE DESIGN

THE NATURE OF DESIGN PROCESS

The design process involves creating artifacts through the use of tools and communication. Tools, whether simple tools like stone axes or complex ones like compilers, are artifacts themselves and are used to create further artifacts. Design activity is inherent in the creation of any artifact, whether it's explicitly recognized or not.

Communication is crucial in translating a design into a product, conveying ideas to development teams. It also facilitates the transfer of expertise among designers and plays a role in the reuse of ideas, a key element in developing design expertise.

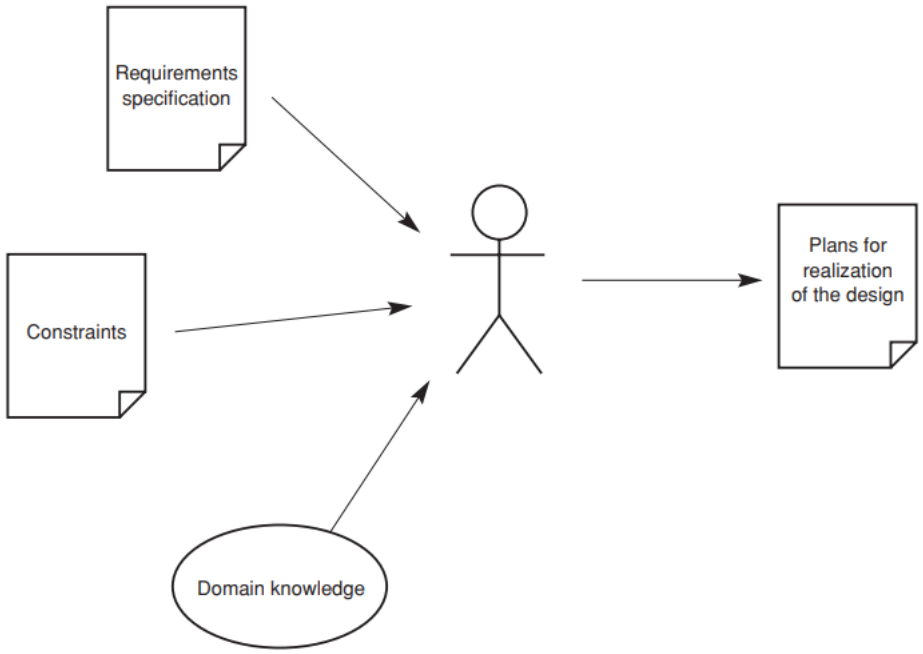
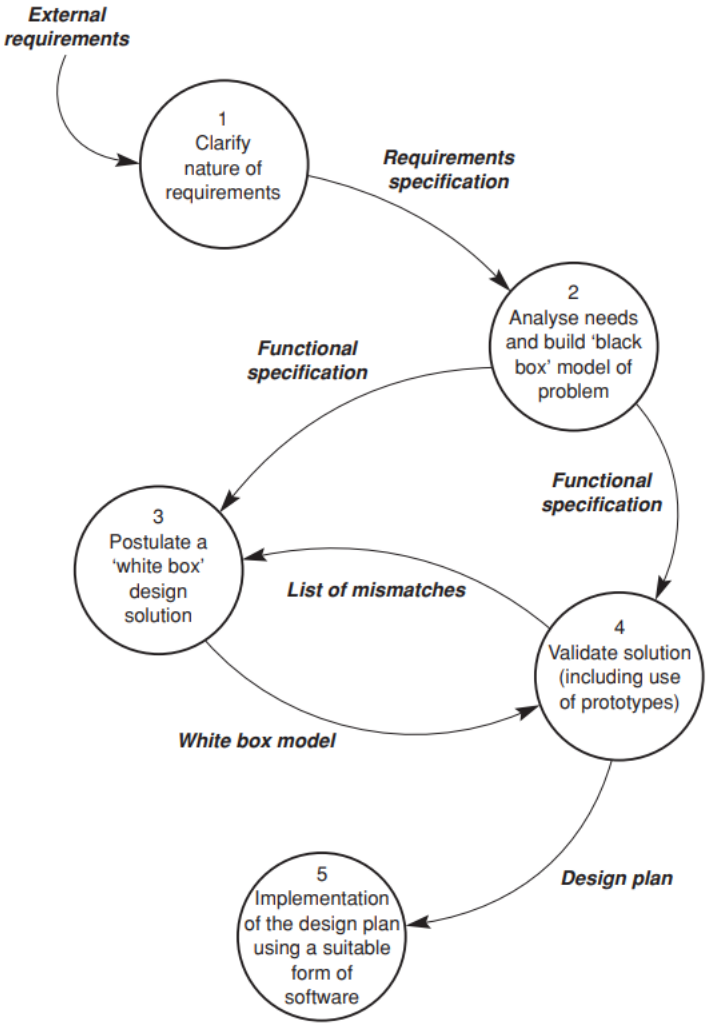


Figure 1.7 The designer's channels of communication.

Design is fundamental in various aspects of our lives, influencing the creation of everyday items like cars, houses, appliances, and more. Good design is essential for both safety-critical objects and everyday items, ensuring functionality, reliability, and user satisfaction.

The design process involves postulating a solution, building a model, evaluating it against requirements, and elaborating on the model for a detailed specification. Iterations and backtracking are common in the design process.



A model of the design process.

Design in software development, like in other disciplines, requires domain knowledge. Communication with stakeholders, including those providing requirements and those implementing the solution, is vital. Design plans encompass the static structure, data objects, algorithms, packaging, and interactions within the system.

Software design introduces complexity due to the need to model and describe both structure and behavior. Various design representations are used to provide different viewpoints on the system. The dynamic nature of software systems adds complexity to the design process, influencing how the system is constructed.

Design as a Problem-Solving Process

Design serves the fundamental purpose of solving problems, typically outlined in a requirements specification. The designer's role is to provide a description of how the requirement can be met.

Design is inherently a problem-solving task, involving the evaluation of options and decision-making based on complex criteria, including trade-offs between factors like size, speed, and ease of adaptation.

The ultimate requirement in design is fitness for purpose. Regardless of elegance or efficiency, the primary goals are to make the system work and perform its intended job effectively. While other factors and qualities matter, they are secondary to fulfilling the system's purpose.

Designers utilize tools such as design methods, patterns, and representations to aid in problem-solving.

Abstraction, a crucial concept in engineering, involves removing unnecessary details from a problem's description while retaining essential structural properties. Abstraction allows the creation of manageable models for large and complex systems, focusing on critical features. It encourages thinking about a system in abstract terms, emphasizing key items such as events, entities, or objects, and delaying detailed considerations until later stages in the design process. Mastering the skill of effective abstraction is essential for designers.

Design as a 'Wicked' Problem

The nature of the design process is such that it lacks an analytical form, leading to several effects, one of which is the presence of multiple acceptable solutions to a given problem. Design, therefore, is often not convergent, and it is characterized as a 'wicked' problem, a term introduced by Rittel and Webber. Wicked problems exhibit properties that make them complex and challenging:

1. Changing Nature of the Problem: A solution for one aspect of a wicked problem often changes the nature of the problem itself. This is exemplified in social planning problems, where improvements in one area may lead to the emergence of new, less tractable issues.

2. Ten Distinguishing Properties:

- No Definitive Formulation: Wicked problems lack a clear, definitive formulation. The specification and understanding of such problems are intertwined with ideas about solving them, challenging the separation of specification and design tasks.

- No Stopping Rule: There are no criteria to determine when a solution has been found. In software, the absence of quality measures makes it difficult to establish the "best" design.

- Not True or False, but Good or Bad: Solutions to wicked problems are subjective, existing in shades of grey. Software designs rarely have unequivocally right or wrong solutions.

- No Immediate or Ultimate Test: Wicked problems lack immediate or ultimate tests for solutions. Evaluating software involves multifaceted considerations, making it challenging to identify a universally "best" solution.

- One-Shot Operation: Every solution to a wicked problem is a one-shot operation; trial-and-error is limited. Large-scale software systems, vital for national activities, often lack resources for extensive exploration of options.

- No Enumerable Set of Potential Solutions: Wicked problems do not have an exhaustively describable set of potential solutions, similar to the diversity in software implementations.

- Every Wicked Problem is Essentially Unique: While similarities exist, every wicked problem is essentially unique. Techniques like design patterns aid reuse but require interpretation, preventing automation of design activities.

- Every Wicked Problem is a Symptom of Another Problem: Resolving one problem may reveal or create new difficulties, mirroring challenges in computer programming where one design choice may lead to new issues.

3. Herbert Simon's View: Herbert Simon's concept of well-structured and ill-structured problems aligns with the properties of an ill-structured problem (ISP). Software design shares characteristics with ISPs, diverging from the expectations of convergence seen in classical problem-solving approaches.

The design process, unlike scientific methods, doesn't necessarily converge to a single solution. Choices made during the design process can lead to complications, and the consequences of these choices may make further decisions more intricate. In some cases, the overall design may be deemed inadequate, requiring a holistic approach to problem-solving due to the interconnected nature of design choices. This challenges the traditional scientific approach where simplification and separation of problems lead to a stepwise convergence toward a solution. Wicked problems, especially prevalent in real-time systems, emphasize the difficulty of isolating and solving individual aspects without considering their combined impact.

In conclusion,

- the design process is concerned with describing how a requirement is to be met by the design product;
- design representation forms provide means of modelling ideas about a design, and also of presenting the design plans to the programmer;
- abstraction is used in problem-solving, and is used to help separate the logical and physical aspects of the design process;
- the software design problem is a ‘wicked’ one and this imposes constraints upon the way in which the process of design can be organized and managed.

TRANSFERRING DESIGN KNOWLEDGE

Design is a creative, non-analytic process, diverging from the pursuit of a singular, scientifically determined solution. Studies on software designers indicate varying proficiency levels, with a limited

number of exceptional designers. Therefore, it is important to devise effective methods to impart design skills to a broader audience.

In Curtis et al., (1988), the exceptional designers were observed to possess three significant characteristics, these are:

1. Familiarity with the application domain, enabling them to map between problem structures and solution structures with ease. (A domain in this sense may be one such as data processing, real-time, telecommunication systems, and so on.)
2. Skill in communicating technical vision to other project members. This was 33 observed to be so significant a factor that much of the design work was often accomplished while interacting with others
3. Identification with project performance, to the extent that they could be found taking on significant management responsibilities for ensuring technical progress.

Notably, exceptional designers may not excel in programming skills.

Acquiring Domain Knowledge

- Accumulating domain knowledge often comes from experience in a specific problem domain.

- The process of acquiring this knowledge can potentially be accelerated and improved.

Opportunistic Design Activity

- Successful designers often engage in opportunistic design activities, deviating from planned strategies based on available information and experience.

- Such activities are structured and reflect the designer's experience and domain knowledge.

Accelerating Knowledge Transfer:

- Successful designers possess procedural knowledge about problem-solving within a domain.

- Design methods emerged in the 1970s to facilitate knowledge transfer by providing a framework for systematic design development.

Components of a Design Method:

- Representation Part: Provides descriptive forms for building models of the problem and solution, emphasizing both abstract and concrete properties.

- Process Part: Describes the organization of transformations between representation forms and includes heuristics for specific problem classes.

- Heuristics: Offer guidelines for organizing activities based on past experiences within a domain.

Design Pattern vs. Design Method:

- Design Pattern: A less procedural mechanism for knowledge transfer, focusing on reusable solution structures.

- Design Method: Provides a framework for systematic design development with representation forms, process organization, and heuristics.

Challenges in Design Methods:

- Design methods offer strategic guidance but may lack detailed solutions to issues.

- Creative acts in design involve recognition, problem restructuring, and procedural knowledge development.

- The process part of a design method structures design transformations, guiding the designer in identifying choices and evaluating consequences.

Trade-offs in Approaches:

- While alternative approaches offer better scope for reusing successful ideas, they may be less accessible to novices.

- Combining the strengths of different approaches in software development remains a challenging task.

CONSTRAINTS UPON THE DESIGN PROCESS AND PRODUCT

In practical design tasks, the freedom to create without limitations is rare. Design activities, including software design, are influenced by various constraints arising from the context in which they occur. These constraints shape not only the final product but also the design process itself. Examining constraints in the design of everyday objects, such as moving houses and designing garden sheds, illustrates the impact of functional, aesthetic, and structural limitations.

1. Functional and Aesthetic Constraints:

- In moving house design, constraints encompass functional aspects (e.g., blocking power outlets) and aesthetic considerations (e.g., clashes of style or color).

- The constraints largely dictate the form of the solution or product, emphasizing the need for a balance between functionality and aesthetics.

2. Software Design Constraints:

- Constraints in software design are diverse and may be influenced by the runtime environment (file structures, user interface aesthetics) and architectural style conventions.

- Conforming to chosen architectural styles (e.g., JavaBeans) and reusing existing software components may impose additional constraints on acceptable design styles and functionality allocation.

3. Implementation-Driven Constraints:

- One crucial constraint is the eventual form of implementation, with historical practices involving hardware and software selection before the design begins.

- Programming language choices are often influenced by external factors like programmer skills rather than problem-specific requirements.

4. Programming Language Paradigms:

- Imperative programming languages (COBOL, Java, etc.) have traditionally dominated software production.

- Design methods have implicitly assumed imperative language constructs, but the principles of design are not limited to the imperative paradigm alone.

5. Constraints on the Design Process:

- Process constraints may be related to designer skills, knowledge, or adherence to a specific architectural style to facilitate future maintenance.

- Some constraints on the product may lead to constraints on the design process, ensuring consistency in output forms.

6. Solution Space Constraints:

- Constraints can be viewed as forming bounds on the solution space, limiting the divergence of design choices.

- Initial specification documents identify many constraints, while others related to reuse expectations may emerge during the design process.

7. Influence of Constraints:

- Constraints effectively narrow the solution space by limiting the choices available to the designer.

- Inconsistencies in specifications may pose challenges, potentially making it impossible to find a solution.

8. Problem-Specific and Solution-Specific Constraints:

- Problem-specific constraints (e.g., user skill levels) and solution-specific constraints (e.g., architectural style) play distinct roles.

- While problem-specific constraints may not be easily incorporated into formal design processes, their influence should be considered throughout.

9. Tracking Constraints:

- Regular design reviews serve as a mechanism to identify and track constraints, ensuring an audit trail and facilitating constraint management.

RECORDING DESIGN DECISIONS

Recording design decisions is a crucial aspect of the design process, benefiting both the ongoing design task and the future maintenance of the system. While designers often document decisions, the rationale behind those decisions is frequently omitted. This lack of recorded rationale poses challenges for understanding the design, especially during system maintenance. The importance of recording decision rationale is emphasized in both the design phase and subsequent maintenance efforts.

1. Design Audit and Rationale Recording:

- Design audits, whether through peer reviews or formal evaluations, can encourage the recording of decision rationale.

- Systematic audits are more effective when accompanied by well-documented reasoning for decisions made during the design process.

2. Maintenance and Original Model Recapitulation:

- Maintenance designers, responsible for system modification or extension, require access to the original models used in system design.

- A reasonably complete model aids in effective implementation of changes, and possessing the rationale for the existing structure assists in evaluating the impact of modifications.

3. Quality Control Motivation:

- The primary motivation for recording design decision rationale is quality control, ensuring a comprehensive understanding of the design.

- A complete picture is essential for producing a reliable and high-quality design.

4. Support from Software Design Methods:

- Software design methods generally excel in supporting the recording of decisions related to product issues through diagrams or notations.

- However, they often fall short in documenting decision rationale and process matters.

5. Enforcing Rationale Recording:

- While recording decisions and their reasons can be incorporated into design practice, enforcement may be challenging without a robust quality control system.

6. Modeling Design Deliberation:

- Various models, such as the 'Potts and Bruns model,' have been proposed to understand the process of design deliberation.

- Tool support for recording decision rationale is limited, and there is a need for general tools in this regard.

7. Documentation as if Rational Process:

- Parnas and Clements suggest that documentation should appear as if a rational design process was followed, even if the actual process was not entirely rational.

- This approach facilitates knowledge transfer to new team members and simplifies system maintenance.

8. Benefits of Idealized Documentation:

- Idealized documentation aids new team members in understanding the project more easily.

- Simplifying the documentation structure makes system maintenance tasks more straightforward.

9. Open Source Software and Documentation:

- The emergence of open source software, exemplified by projects like Linux, highlights the importance of version control and documentation in managing collaborative efforts.

- Academic study on the differences between open source and traditional software development processes is limited.

DESIGNING WITH OTHERS

For effective software development, sound design is critical, especially in large-scale projects involving design teams. Collaborative design, however, introduces complexities and constraints that impact the process and the final product. While some successful systems have been the result of a few great designers' work, the reality for most projects involves a team.

When lacking a standout designer, team-based design introduces two major challenges:

1. Task Division: Determining how to split the design task among team members and establishing interfaces between different parts.

2. Integration: Bringing individual contributions together and negotiating between team members.

The trend toward modular designs has somewhat eased the task division issue, emphasizing the need for a fair balance of effort. Integrating diverse contributions, however, remains a mix of

technical and managerial challenges, where negotiation is necessary but shouldn't compromise the overall design integrity.

Research, such as Gerald Weinberg's work in "The Psychology of Computer Programming," has explored group organization effects on programming and design. The hierarchical "chief programmer" approach, with a central designer supported by a team, has been suggested but is seldom employed in practice.

Psychological factors influencing team behavior in designing, including team size, domain knowledge distribution, and organizational issues, have been studied. The need for operational scenarios from customers to understand system use behavior is emphasized, highlighting a broader organizational challenge.

In essence, the challenges of design teams lie more in group psychology than in design technology. Most widely used design methods do not offer specific guidance for team adaptation, recognizing the importance of learning, negotiation, and communication in the collective programming effort.

CONTEXT FOR DESIGN

The design process does not occur in isolation but is influenced by various factors within the context of software development. Here we, explore these external elements that impact the design process.

Different approaches exist for developing large software systems, each emphasizing various activities. The concept of a software life-cycle, despite its valuable role in organizing development activities, can inadvertently stifle creativity if overly institutionalized.

Table 3.1 Some examples of software development processes

Development process	Reasons for adoption	Typical domains
Linear (e.g. 'waterfall')	Where the needs are reasonably well determined and a 'longer' delivery time is acceptable.	Many 'made to measure' systems are developed in this way. This approach is one that has been used very widely and is still appropriate for systems that need to demonstrate any form of high integrity.
Incremental	Where there may be a need to demonstrate feasibility of an idea; establish a market position rapidly; or explore whether a market might exist. Providing a rapid time to market is often an important factor for adopting such an approach.	Widely used for 'shrink wrap' software of all sorts, such as office packages, games, operating systems. Also, where there is a need to interact closely with the end-users during development.
Reactive	Where evolution of a system is largely in response to what the developers perceive as being needed.	Open source software of all kinds is generally developed in this way, as are many websites.

The evolution of thinking about life-cycle models, exemplified by Royce's waterfall model, has addressed fundamental project questions. However, the model primarily guides where to ask questions rather than providing answers.

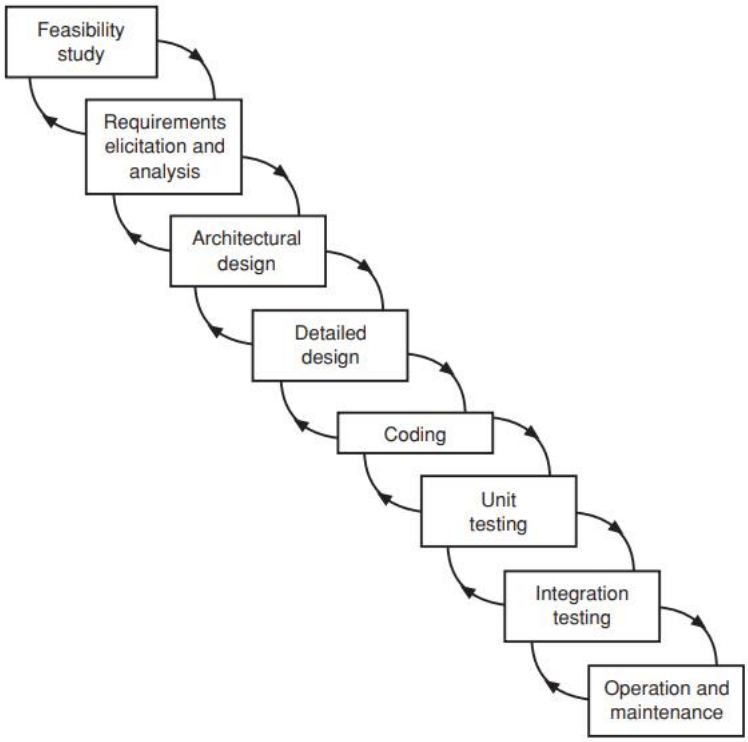


Figure 3.1 The waterfall model of the software life-cycle.

Development phases, as depicted in the waterfall model, involve:

1. Feasibility Study:Examining if a solution is achievable within constraints, providing initial consideration for design issues.

2. Requirements Elicitation:Identifying end-user needs independently, though increasing interdependence with design and implementation considerations.

3. Analysis: Formally modeling the problem, serving as a significant input to the design process.

4. Architectural Design: Defining the overall solution's form, such as choosing a distributed system and determining major elements and interactions.

5. Detailed Design: Developing descriptions of identified elements, involving feedback loops to ensure alignment with requirements.

6. Coding (Implementation): Translating abstract design plans into a realized system, involving new code or integrating existing components.

7. Unit and Integration Testing: Validating the implemented system against requirements and design.

8. Operation and Maintenance: Operating can resemble extended testing, while maintenance involves repeating earlier tasks to preserve system integrity.

Effort allocation varies, but a study suggests that "detailed design" consistently takes around 20% of total effort and time across different organizations.

The design task is closely tied to preceding phases, as illustrated by a simplified example of a library system. Requirements elicitation focuses on user needs, while analysis produces a solution-oriented model. The design's role becomes crucial at this 'watershed,' transitioning from stating problems to contemplating solutions.

Processes are rarely strictly sequential, and iterations are common due to revealed inconsistencies or omissions. In cases where there's no established market, user feedback is essential, and the potential for imaginative applications may emerge, diverging from the original developer's vision.

ECONOMIC FACTORS

The design process in software development involves navigating a vast solution space, making errors in design likely. These errors can manifest as self-inconsistency, inconsistencies with specifications, or misalignments with user needs. Detection of errors becomes increasingly costly as development progresses.

- Self-Consistency Errors: Revealed during design refinement, implementation, or testing, requiring adjustments based on the type of error.

- Specification Consistency Errors: Typically identified during testing, though prototypes may expedite detection. Verification involves checking design against specifications.

- User Needs Consistency Errors: Harder to detect, often requiring validation through prototypes. Boehm distinguishes verification ("Are we building the product right?") from validation ("Are we building the right product?").

Early error detection is crucial, as costs for correction rise significantly with each developmental stage. Boehm's study indicates a tenfold increase in fixing design errors detected during testing compared to implementation.

Prototyping is one approach for early evaluation, but potential side effects might reduce cost-effectiveness.

Economic Considerations:

1. Cost of System Changes: Software systems evolve, undergoing perfective, adaptive, and corrective maintenance. Around 65% of maintenance work involves perfective maintenance, emphasizing the need for systems that can be easily modified.

2. Maintenance Types:

- Perfective: Extending and improving the system.

- Adaptive: Meeting external change needs.

- Corrective: Fixing bugs in the operational system.

3. Structural Flexibility: Systems should allow easy modification to accommodate changes, aligning with principles common in other engineering branches.

Organizational budgeting tends to favor short-term considerations, limiting the adoption of effective long-term practices, despite the potential to increase initial costs for decreased long-term expenses.

ASSESSING DESIGN QUALITIES

The context is crucial when it comes to assessing design quality in software, the challenge lies in the abstract and complex nature of properties.

Framework for Assessment:

- Quality Concepts: Abstract ideas of what constitutes good and bad properties of a system.

- Design Attributes: Measurable characteristics of design entities, providing mappings between abstract properties and countable features.

- Counts: Identification of lexical features for obtaining metric values.

Example Metric:

- Cyclomatic Complexity: Measures the structural complexity of program code based on the number of possible control paths.

Expanded Framework:

- Use: Identifies the purpose of measurements, differentiating between static and dynamic properties.

- Quality Factors: Determines quality concepts associated with the purpose (ilities).

- Quality Criteria: Relate requirements-oriented properties to solution-oriented properties, mapped onto chosen metrics.

Economic Issue:

- Early Detection of Errors: Crucial for cost-effectiveness; errors detected later in development stages incur significantly higher costs.

Prototyping: Offers an approach for early evaluation but may have side effects impacting cost-effectiveness.

System Changes Cost:

- Perfective Maintenance: Extending and improving the system.

- Adaptive Maintenance: Meeting external change needs.

- Corrective Maintenance: Fixing bugs. Perfective maintenance dominates (65%).

Structural Flexibility: Systems should allow easy modification to adapt to changes.

The 'ilities':

1. Reliability: Concerned with dynamic characteristics, predicting completeness, consistency, and robustness.

2. Efficiency: Measured through resource use (processor time, memory, etc.), involves making projections.

3. Maintainability: Design should allow future modification, relates to implementation factors and detailed design.

4. Usability: Includes HCI design, cognitive elements, and consistency in providing a clear mental model.

Cognitive Dimensions:

Table 4.1 The cognitive dimensions

Dimension	Interpretation
Abstraction	Types and availability of abstraction mechanisms
Hidden dependencies	Important links between entities are not visible
Premature commitment	Constraints on the order of doing things
Secondary notation	Extra information provided via means other than formal syntax
Viscosity	Resistance to change
Visibility	Ability to view components easily
Closeness of mapping	Closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Error-proneness	Notation invites mistakes
Hard mental operations	High demand on cognitive resources
Progressive evaluation	Work-to-date can be checked at any time
Provisionality	Degree of commitment to actions or marks
Role-expressiveness	The purpose of a component is readily inferred

- Premature Commitment: Making early design decisions constraining later ones.

- Hidden Dependencies: Relationships not fully visible between components, impacting modification during maintenance.

- Secondary Notation: Additional information conveyed through means other than official syntax.

- Viscosity: Resistance to change, indicative of design quality and interdependence of modules.

Assessment Challenges:

- Dynamic Attributes: Often considered more important but challenging to assess compared to static attributes.

- Analogy with Musical Notation: Similarities in assessing static attributes, but ultimate judgment requires execution (hearing).

Conclusion:

- Measuring Abstractions: Difficult; metrics analysis schemes seek detailed information, potentially conflicting with experienced designers' preference for consistent abstraction levels.

- Dynamic Attributes Assessment: Essential for assessing fitness for purpose, but challenging due to existing focus on static attributes.

- Cognitive Dimensions: Offer additional insight into design quality, emphasizing factors like premature commitment, hidden dependencies, secondary notation, and viscosity.

QUALITY ATTRUBUTES OF THE DESIGN PRODUCT

Key Challenges:

1. Identify design attributes related to quality.

2. Extract information from design documents.

Design Attributes:

1. Simplicity:

   - Good designs are simple, meeting objectives without unnecessary embellishments.

   - Simplify without oversimplifying.

   - Assess complexity using measures like control flow, structure, and comprehension.

2. Modularity:

   - Break down problems into smaller, replaceable components.

   - Use well-defined interfaces.

- Benefits: ease of replacement, better comprehension, and potential for reuse.
- Assess using measures like coupling and cohesion.

3. Information-Hiding:
- Keep details within modules; don't expose to the outside.
- Use access procedures to control external access.
- Balances integrity and flexibility.
- Relates to reliability and maintainability.

Assessing Design Quality:

- Technical reviews, like walkthroughs, are crucial.
- Identify weaknesses and potential issues.
- No quantified measure of quality, but effective for extracting and assessing information.

Parnas and Weiss's Design Goals:

1. Well-structured and consistent.
2. Simple yet not oversimplified.
3. Efficient resource usage.
4. Adequate meeting requirements.
5. Flexible to accommodate changes.
6. Practical with suitable module interfaces.
7. Implementable with current technology.
8. Standardized using familiar notation for documentation.

Conclusion:

Effective design balances simplicity, modularity, and information-hiding, assessed through technical reviews, aiming for well-defined, practical, and implementable solutions.

ASSESSING THE DESIGN PROCESS

Design Process Quality:

- Design products result from some design process, structured or not.

- Process quality doesn't guarantee product quality but is crucial for understanding design quality.

- Design involves building a model and exploring ways to realize it.

Software Process Maturity Model:

1. Initial. The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success in a project depends upon individual effort.
2. Repeatable. Basic project management processes are established and used to track cost, schedule and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3. Defined. The software process for both management and engineering activities is documented, standardized and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software.
4. Managed. Detailed measures of the software process and product quality are collected. Both the software process and the products are quantitatively understood and controlled using detailed measures.
5. Optimizing. Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies.

Challenges in Design Process:

- Design phase is more problem-oriented than coding or testing.

- Difficult to achieve repeatable design due to its creative nature.

Improving Design Process Quality:

- Include input from:
  - Domain knowledge.
  - Method knowledge.
  - Experience from similar projects.

Input Methods:

1. Technical Reviews: Gain knowledge from others' experiences.

2. Management Reviews: Focus on effort and deadline estimates.

3. Prototyping: Provides domain knowledge and experience.

Management Review:

- Assesses design development and provides a framework for other inputs.

- Reporting and tracking mechanisms are crucial for monitoring and addressing quality issues.

Documentation Quality:

- High-quality documentation and configuration procedures aid in reusing experience.

- Use of standard notations and design practices is influential but doesn't guarantee quality.

In Conclusion,

Design process quality is essential for understanding design quality. While challenging to quantify, input from technical and management reviews, along with prototyping, improves the process. Documentation quality and the use of standard practices play influential roles.

REPRESENTING ABSTRACT IDEAS

Role of Abstraction in Design:

- Abstraction helps focus on crucial features during design.

- Designers need ways to represent abstract ideas about problems and solutions.

Purposes of Representation:

- Capture designer's solution ideas.

- Explain ideas to others.

- Check consistency and completeness.

Benefits of Representation:

- Aids concentration by reducing information overload.

- Acts as an 'external memory device.'

Levels of Abstraction:

- Representations can vary from a 'black box' description to detailed 'white box' descriptions.

- Software design uses various forms of box and line notations.

Representation in Design Methods:

- Design methods use representations for specific steps.

- Use of representations is not limited to formal methods; many designs occur without a defined method.

Viewpoints and Consistency:

- Representations are linked to viewpoints.

- Different viewpoints capture specific properties of the design model.

- Consistency is essential where viewpoints intersect.

Viewpoint Concept:

- Viewpoints are projections of the design model.

- Representations describe attributes or characteristics, not the design itself.

Comparison with Requirements Engineering:

- 'Viewpoint' in this context differs from its use in requirements engineering.

- Here, it focuses on projecting properties of the model; in requirements, it's more user-centric.

Consistency Maintenance:

- Projections need consistency to maintain a coherent design model.

- Checking consistency involves ensuring harmony between different sets of design properties.

Limitation in Checking 'Correctness':

- The design model is only captured through representations.

- Checking consistency between representations is possible, but absolute correctness is challenging.

DESIGN VIEW POINTS

Distinctive Nature of Software:

- Software is dynamic and abstract, posing challenges in representation.

- Describing both static and dynamic properties is essential in the design model.

Classification of Design Representations:

- Broadly classified into constructional, behavioural, functional, and data-modelling forms.
- Each class addresses specific aspects of software design.

Constructional Forms:

- Focus on how software structuring forms are used in the final system.
- Describe outcomes of design practices like methods.
- Examples include data specification, threads of execution, and packaging constructs.
- Primarily concerned with static structures rather than runtime behavior.

Behavioural Forms:

- Concerned with causal links between events and system responses.
- More abstract than constructional forms, dealing with system operations.
- Examples often resemble finite-state machines, handling dynamic relationships.
- Used for black box and white box modeling roles.

Functional Forms:

- Address the challenge of describing what a system does.
- Difficult due to problem-driven nature but crucial for detailed design.
- Needed to describe runtime behavior of program elements.

Data-Modelling Forms:

- Important for detailed design, especially in dealing with data structures.
- Capture dependencies such as type, sequence, and form.
- Often considered more in analysis than design due to detailed nature.
- Primarily used in white box roles.

Derived Viewpoints:

- Some transformations applied to the design model to generate new representation forms.
- Examples include using interpreters to execute designs for behavioral aspects.

Classification Variations:

- Various ways to classify design descriptions, e.g., Kruchten's '4+1 View.'

- Attributes may not be uniquely confined to one viewpoint.

THE ARCHITECTURE CONCEPT

What is Architecture?

- The term "architecture" has been informally used in software development.

- Analogies to building architecture exist, where the role of an architect is associated with abstract design tasks.

- Examples include computer architecture, naval architects, landscape architects, and computer architects.

- Architectural style provides a framework for design tasks, describing the form and constraints of a family of architectural instances.

Roles of Architectural Style:

1. Abstract Description: Provides a high-level, abstract description of the overall form of a design solution.

2. Mismatch Identification: Assists in identifying potential mismatches between different elements in a system.

3. Notation Choice: Helps in selecting appropriate constructional notations for describing a solution.

4. Implicit Design Processes: Imposes a particular architectural style based on the chosen design method.

5. Impacts on Software Development:

  - Understanding: Provides an abstract vocabulary for understanding high-level system design.

  - Reuse: Influences the idea of components and supports reusability.

  - Evolution: Guides the expected dimensions of system evolution.

  - Analysis: Offers a framework for checking various features of a design.

  - Management: Aids in planning, especially for future changes.

Classifying Architectural Styles:

- Classification scheme includes components, connectors, control mechanisms, data communication, data-control interaction, and design reasoning.

- Various classification schemes exist, like those by Perry and Wolf, Shaw and Garlan, and Shaw and Clements.

Table 6.1 Major categories of software architectural style

Category	Characteristics	Examples of Styles
Data-flow	Motion of data, with no 'upstream content control' by the recipient	Batch sequential Pipe-and-filter
Call-and-return	Order of computation with a single thread of control	Main program/subprograms 'Classical' Objects
Interacting processes	Communication among independent, concurrent processes	Communicating processes Distributed Objects
Data-centred repository	Complex central data store	Transactional databases Client–Server Blackboard
Data-sharing	Direct sharing of data among components	Hypertext Lightweight threads

Examples of Architectural Styles:

1. Pipe and Filter:

   - Components: Processes acting as filters.

   - Connectors: Dataflow mechanisms.

   - Control Mechanism: Data-driven.

   - Data Communication: Through data streams.

   - Data-Control Interaction: Limited.

   - Design Reasoning: Varies.

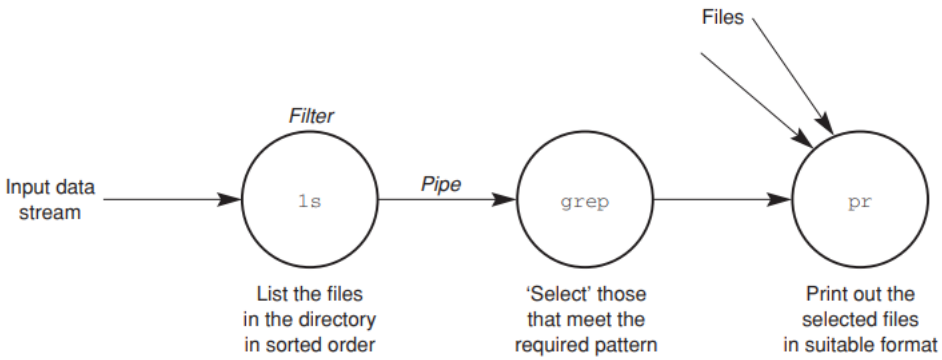


Figure 6.1 The pipe-and-filter style: Unix processes.

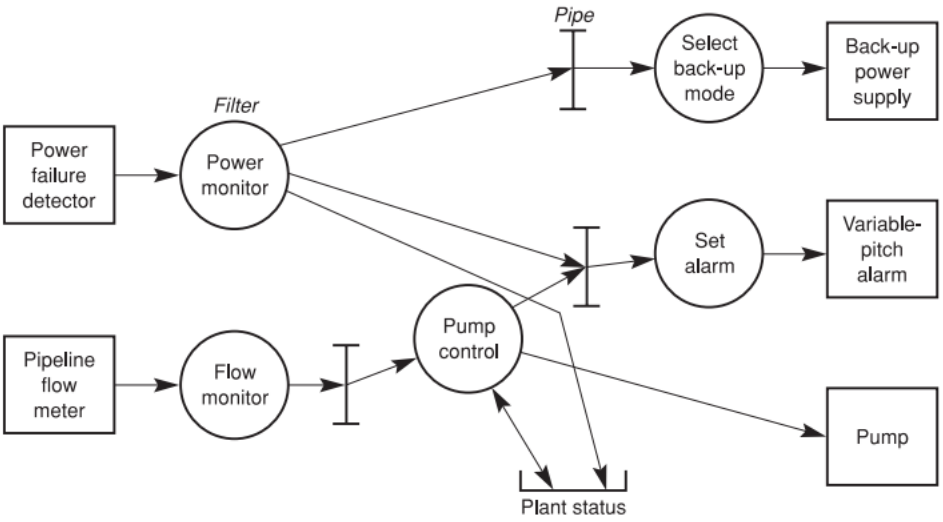


Figure 6.2 The pipe-and-filter style: MASCOT diagram.

Table 6.2 **The pipe-and-filter architectural style**

Feature	Instantiation in pipe and filter
Components	Data transformation processes.
Connectors	Data transfer mechanisms (e.g. Unix pipes, files, etc.).
Control of execution	Typically asynchronous, control is transferred by the arrival of data at the input to a process. Upstream processes have no control of this.
Data communication	Data is generally passed with control.
Control/data interaction	Control and data generally share the same topology and control is achieved through the transfer of data.
Design reasoning	Tends to employ a 'bottom-up' approach using *function* due to the emphasis placed upon the filters (components). A design method such as JSP (Chapter 14) may generate this style of solution.

2. Call and Return:

- Components: Main program and subprograms.
- Connectors: Procedure calls and returns.
- Control Mechanism: Hierarchy.
- Data Communication: Through parameters.
- Data-Control Interaction: Limited.
- Design Reasoning: Procedural.

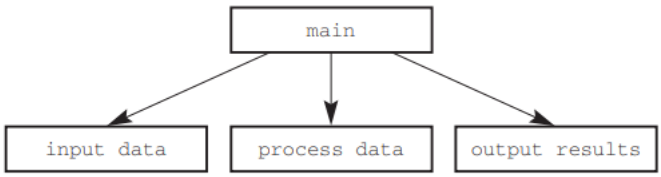


Figure 6.3 The call-and-return style: subprogram invocation.

Table 6.3 The call-and-return architectural style

Feature	Instantiation in call and return
Components	Subprogram units.
Connectors	Subprogram invocation (calling).
Control of execution	Sequencing is controlled through the calling hierarchy and (in detail) the algorithms in the components.
Data communication	Data is passed via parameters and can also be accessed directly (global access).
Control/data interaction	This is relatively limited, beyond the linking of parameters and return inform within the 'calling stack'.
Design reasoning	Encourages use of a 'top-down' strategy, based upon *function*. A design method such as the 'traditional' Structured Analysis/Structured Design will produce solutions that employ this style (Chapter 13).

3. Data-Centred Repository:

- Components: Central mechanism for data storage.
- Connectors: Interaction through queries.
- Control Mechanism: Centralized data control.
- Data Communication: Through queries.
- Data-Control Interaction: Centralized control.
- Design Reasoning: Data-centric.

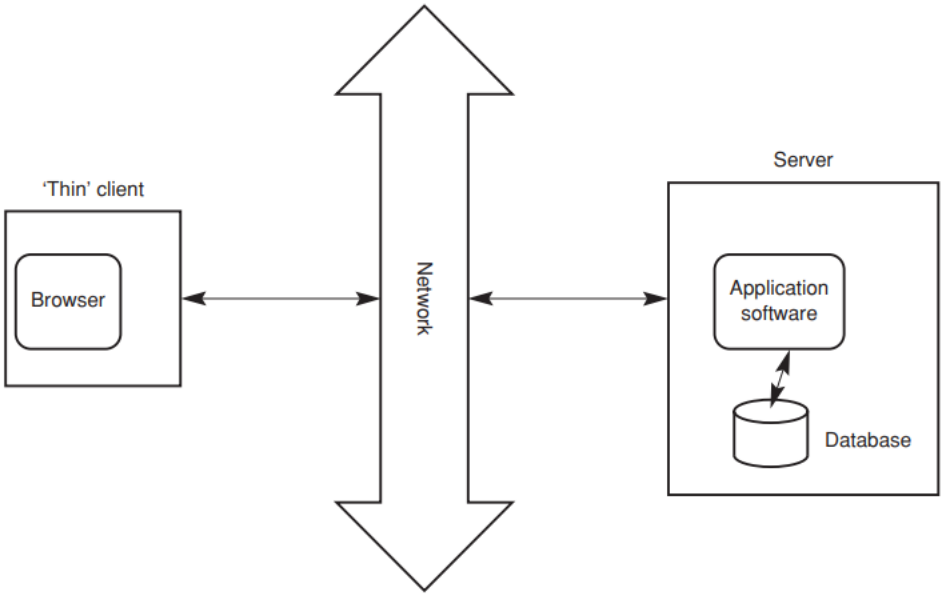


Figure 6.4 The data-centred repository style: simple client–server system.

Table 6.4 **The data-centred repository architectural style**

Feature	Instantiation in data-centred repositories
Components	Storage mechanisms and processing units.
Connectors	Transactions, queries, direct access (blackboard).
Control of execution	Operations are usually asynchronous and may also occur in parallel.
Data communication	Data is usually passed via some form of parameter mechanism.
Control/data interaction	Varies quite widely. For a database or a client–server system, these may be highly synchronized, whereas in the case of a blackboard there may be little or no interaction.
Design reasoning	A *data modelling* viewpoint is obviously relevant for database and client–server systems. The wider variation of detail that occurs in this style tends to preclude the widespread use of more procedural design approaches.

Role in Knowledge Transfer:

- Provides a framework and vocabulary for top-level design ideas.

- Determines the choice of design strategy and notation.

- Assists in preserving architectural integrity during later changes.

DESIGN METHODS

- Design methods provide a procedural description of how to generate design solutions for a given problem.

- They are more detailed and prescriptive in software design compared to other domains.

- Possible reasons for their prominence in software design include the rapid development of the computing industry, the need for peer-to-peer knowledge transfer, and the invisibility of software.

- Design methods implicitly produce a design solution embodying a particular architectural style.

- There is a unique expectation in software that following a design process will ensure a sound design product.

- Studies suggest that experienced designers use design methods more when lacking confidence in their domain knowledge.

- Designers tend to adapt and modify design methods based on their experience.

- The popularity and importance of design methods have varied over the years, possibly due to a growing maturity of design expertise and increasing complexity of software structures.

- While design methods continue to play a useful role in transferring knowledge to novice designers, their value in peer-to-peer exchange may be reduced.

DESIGN PATTERNS

- Design patterns are associated with the object-oriented architectural style but can, in principle, be employed with other styles.

- The concept is rooted in the work of architect Christopher Alexander and has been adapted for object-oriented software by the pattern community.

- A design pattern is a generic solution to a problem likely to be part of a more extensive problem, conveying information about the problem and a strategy for addressing it.

- Design patterns embody a master/apprentice model for transferring knowledge, educating about both problems and solutions.

- They facilitate peer-to-peer exchange of design knowledge in smaller chunks compared to design methods.

- Expert designers often employ the concept of 'labels for plans,' recognizing and labeling sub-problems they know how to solve.

- Patterns have limitations, such as the need for accessibility and recognizability, which can be challenging with a large set of patterns.

- The pattern concept recognizes and embodies a useful level of reuse of design knowledge.

- Experimental findings suggest that design patterns can be used to produce solutions with a consistent style, especially in terms of enabling future change.

- The design pattern concept is not restricted to detailed design but extends to programming (referred to as an idiom) and architectural levels.

DESIGN REPRESENTATIONS

Table 7.1 The selection of black box design representations

Representation form	Viewpoints	Design characteristics
Data-Flow Diagram	Functional	Information flow, dependency of operations on other operations, relation with data stores
Entity–Relationship Diagram	Data modelling	Static relationships between design entities
State Transition Diagram	Behavioural	State-machine model of an entity
Statechart	Behavioural	System-wide state model, including parallelism (orthogonality), hierarchy and abstraction
Structure Diagram (Jackson)	Functional, data modelling, behavioural	Form of sequencing adopted (operations, data, actions)
UML: Class Diagram	Constructional	Interactions between classes and objects
UML: Use Case Diagram	Behavioural and functional	Interactions between a system and other 'actors'
UML: Activity Diagram	Behavioural and functional	Synchronization and coordination of system activities

Table 7.2 The selection of white box design representations

Representation form	Viewpoints	Design characteristics
Structure Chart	Functional and constructional	Invocation hierarchy between subprograms, decomposition into subprogram units
Class and Object Diagrams	Constructional	*Uses* relationships between elements, interfaces and dependencies
Sequence Diagrams	Behavioural	Message-passing sequences, interaction protocols
Pseudocode	Functional	Algorithm form

Developing a Diagram:

- The section explores the development and checking of diagrammatic forms, focusing on the use of tabular structures.

- Tabular representations, such as State Transition Tables (STT), can assist in checking for completeness and consistency in diagrams.

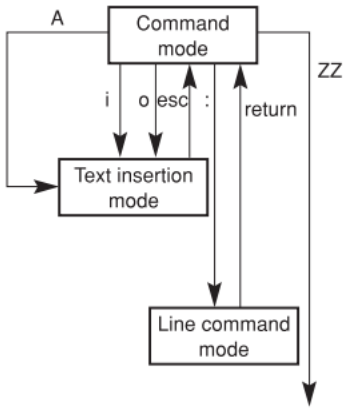
- Both tabular forms and diagrams are complementary, with tables being better suited for large-scale descriptions, while diagrams aid in visualizing relationships.

- Two examples are briefly examined: transforming a State Transition Diagram (STD) into an STT and developing an Entity Life-History Diagram (ELHD) from an ELH matrix.

A Tabular Representation of a Diagram:

- The State Transition Table (STT) is a tabular form of the State Transition Diagram (STD).

(a) The STD form



(b) The STT

	A	i	o	esc	:	return	ZZ
Command mode (C)	T	T	T		L		Exit
Text insertion mode (T)				C			
Line command mode (L)						C	

Figure 7.36 An STT describing the Unix **vi** text editor, with corresponding STD.

-

	stack	cleared to land	abort landing	touch down	take off	abort take off	park	cleared for take off
in flight	stacked	landing approach						
stacked		landing approach						
landing approach			in flight	on runway				
on runway					in flight	on ground	on ground	
on ground								on runway

Figure 7.37 An STT describing an aircraft in an air traffic control zone.

- The STT represents states along the left column and events as column headers, with entries denoting final states after an event in a specific state.

- STT aids in checking model completeness and correctness, identifying questions about the model, and accommodating structure extensions more readily.

Developing a Diagram from a Tabular Representation:

- The Entity Life-History Diagram (ELHD) development process in SSADM involves using an ELH matrix.

ENTITIES	EVENTS					
	Post bulletin	List b-b contents	Read bulletin	Delete expired bulletin	Mail bulletin author	Save bulletin in file
Bulletin	*		*		*	*
Bulletin board	*	*				
User	*	*	*		*	*
Daemon				*		

Figure 7.38 Example of an Entity Life-History Matrix (SSADM).

- Steps include listing entities, identifying events(External, Temporal, Internal), creating the ELH matrix, and drawing the ELHD.

- The ELH matrix helps identify links between events and entities, guiding the creation, modification, or deletion of entries.

- The iterative process of drawing the ELHD involves several evolutionary stages, with tables assisting in listing elements and identifying dependencies.

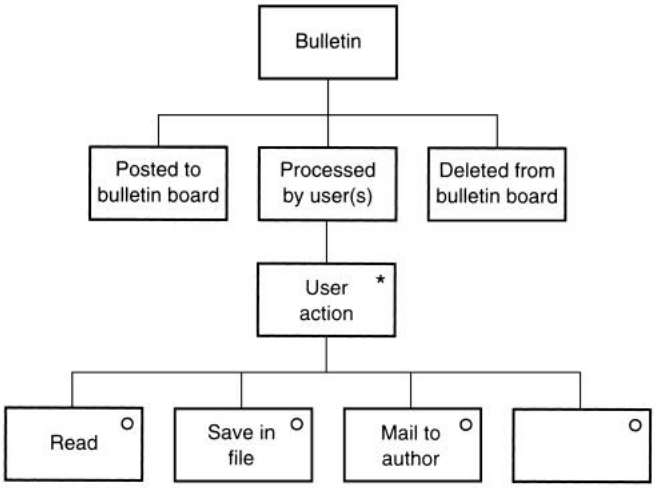


Figure 7.39 Example of an Entity Life-History Diagram (SSADM).

RATIONALE FOR DESIGN METHODS

What is a Software Design Method?

1. Motivations for Software Design Methods:

- Software design methods have been essential in addressing challenges like the rapid growth of the software industry, continuous technological change, and the need for frameworks to structure ideas in an inherently invisible medium.

2. Design Process as Navigation Through a Solution Space:

- The design process is seen as navigating through a solution space, with each step providing the designer with multiple options. A design method aims to guide designers in making choices and assessing the consequences of their decisions.

3. Understanding a Design Method:

- A design method is defined as a structured and procedural way of doing things. It is not as prescriptive as methods for routine tasks like making tea but provides a general strategy and guidelines for the design process.

4. Declarative and Procedural Knowledge:

- Successfully employing a design method involves declarative knowledge (describing tasks at each step) and procedural knowledge (processes and strategies for making choices).

5. Two Categories of Software Design Methods:

- Formal Methods: Use mathematical notations for representation, allowing consistency checking and rigorous transformations. Process parts may be less refined.

- Systematic Methods: Use diagram-based representations and generally have less mathematical rigor. Process parts are less prescriptive, allowing more flexibility.

6. Heuristics in Design Methods:

- Heuristics are sets of guidelines or techniques for handling specific situations encountered during the design process. They are experiential and contribute to reusing the experience of others.

<u>The Support that Design Methods Provide:</u>

1.Reasons for Using Design Methods:

- Design methods provide an artificial framework to assist in thinking about invisible elements, and they are crucial for representing complex software structures.

- Scale is a significant factor, as large software-based systems benefit from a systematic and structured approach provided by design methods.

2. Two Aspects of Benefit:

- Benefits of using design methods are categorized into technical issues (problem-related aspects of design) and management issues.

3. Technical Issues:

- Knowledge Transfer: Design methods assist less experienced or unfamiliar designers by providing guidance in formulating and exploring mental models.

- Consistent Structure: Design methods help produce systems with consistent structures, crucial when multiple designers contribute to a project.

- Documentation for Maintenance: Standardized records and representations aid maintenance teams in understanding and preserving the system's intentions.

- Cognitive Load Management: Design methods reduce cognitive load, lowering the likelihood of errors and ensuring comprehensive consideration of all problem factors.

4. Design Virtual Machine (DVM):

- Each design method provides a DVM, a high-level framework to express ideas about program forms. The DVM is crucial in the transition from architectural design to physical design.

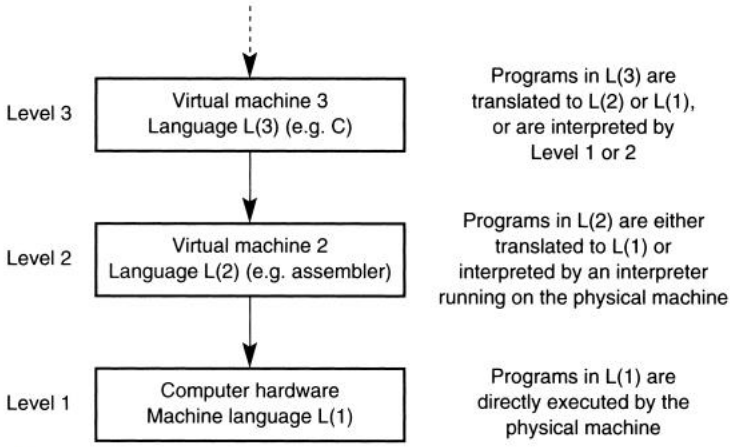


Figure 8.3 The use of virtual machines in the design process.

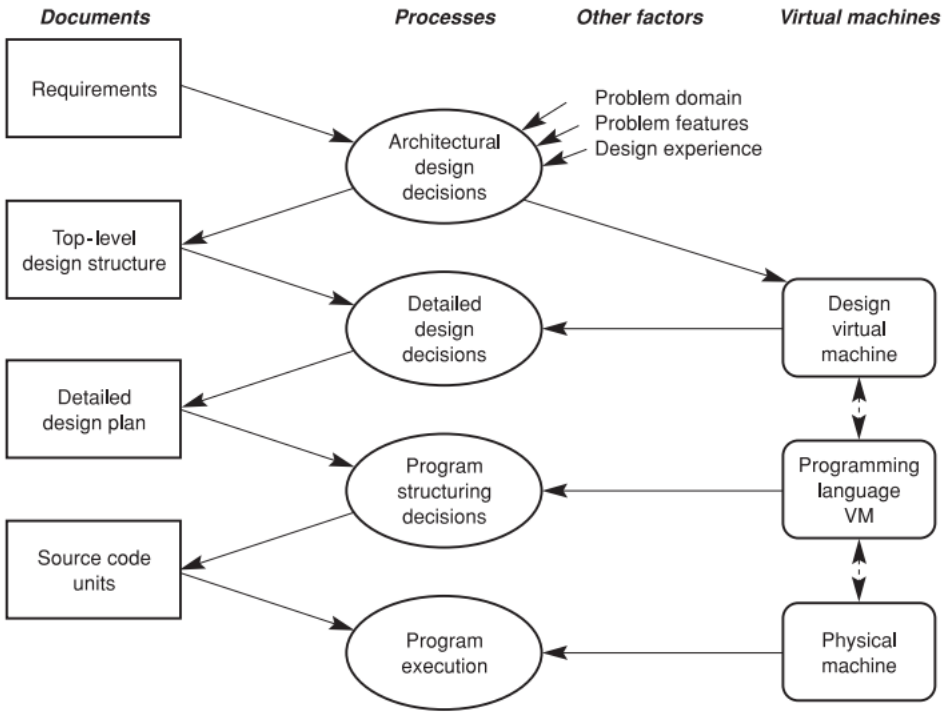


Figure 8.4 The link between the DVM and the virtual machine levels used on a computer. (Each method has a design virtual machine embodied within it, helping to determine such aspects as the set of viewpoints used, the strategy, architectural assumptions, etc.)

5. Architectural Style and DVM:

- DVM is characterized by architectural style, viewpoints, relationships, and the basic strategy of the method. It forms assumptions about the general form of the solution, providing a framework for the designer.

6. Ideal Matching of DVM:

- Ideally, the DVM should match the virtual machine of the eventual implementation form. Mismatches, as seen with the introduction of the Ada language, can create challenges.

7. Management Benefits:

- Documentation: Design methods provide a systematic framework for recording decisions, ensuring consistency and aiding teams on large projects.

- Progress Milestones: Design methods help identify important progress milestones, structuring both the design process and product.

8. Designing for Change:

- Design methods encourage planning for change, a crucial aspect for maintaining and modifying existing systems. They help designers explore solutions in a structured manner that considers future modifications.

In summary, design methods offer both technical and management benefits, providing a structured and systematic approach to software development, aiding understanding, consistency, and adaptability.

Why Methods Don't Work Miracles:

This section addresses the limitations of design methods and emphasizes that while they offer valuable guidance in the software design process, they do not provide a one-size-fits-all solution. The main points discussed include:

1. Nature of Design Methods:

- Design methods offer a framework for organizing the development of a design, recommended representation forms, and guidance on the design process and criteria.

- The emphasis is on providing guidance, but the actual choices and decisions in a design task depend on the specific problem at hand.

2. Analogy with Cooking Recipe:

- The analogy of using a recipe for cooking is presented. A recipe provides a process for producing a dish, similar to how a computer program is a process for various tasks.

- Designing software is compared to designing a recipe, requiring insight into the problem domain, awareness of materials, and knowledge of system capabilities.

3. Design Process vs. Recipe:

- Design methods are processes used to design another process, not recipes themselves.

- While a design method can offer guidance on organizing and presenting instructions, it cannot dictate specific details like ingredient choices or oven temperature.

4. Focus on Problem Domains:

- Focusing a method on a particular domain, like data processing or information retrieval, can provide tighter guidance for certain decisions.

- However, most methods aim to be applicable to reasonably wide domains to optimize their use.

5. Sequential Actions in Design Methods:

- Design methods often define sequences of actions to be performed by a designer, but this may provide only an inadequate approximation to expert behavior.

- Experienced designers may work on multiple threads of action simultaneously at different levels of abstraction.

6. Limitations and Recognizing Them:

- The purpose is not to discourage the use of design methods but to highlight their limitations.

- Design methods have constraints, and users should recognize these limitations instead of having exaggerated expectations.

In summary, the section emphasizes that design methods are valuable tools for providing guidance in the software design process, but users must recognize their limitations and avoid expecting them to be a cure-all solution for every design challenge

Problem Domains and Their Influence:

1. Modeling the Real-World Problem:

- Design methods generally expect designers to start the process by creating a model of the real-world problem to be solved.

- The form of this model is based on the underlying Design Virtual Machine (DVM), strongly influencing the derived solution, including its architectural style.

2. Viewpoints in Model-Building:

- Various viewpoints, such as function, information flow, data structure, actions, data objects, and time ordering of actions, are used to map the characteristics of the original problem in the initial model.

- The choice of viewpoints depends on the problem domain.

3. Specialized Methods and Problem Domains:

- Some methods, like those for compiler design, may be highly specialized for specific domains.

- However, most problem domains are more general, and methods may be applicable across various classes of problems.

4. Choosing Architectural Style:

- The question of whether the problem characteristics should determine the design approach or if the choice of architectural style should drive the design approach is discussed.

- Pragmatic factors, compatibility needs, designer expertise, and prevailing fashions may influence the choice.

5. Problem Domain Classifications:

- Batch systems, reactive systems, and concurrent systems are identified as three general problem domain classifications.

- These classifications are not mutually exclusive, and a system might exhibit characteristics of more than one.

6. Influence on Design Goals:

- The classification of problem domains based on how they influence the designer's goals is considered.

- Examples include problems where the form and content of data guide actions (e.g., transaction processing) and problems where the existence and content of data guide actions (e.g., real-time systems).

7. Critical Aspects in Different Domains:

- Different problem domains have different critical aspects, such as correctness verification in systems processing financial transactions and ensuring correct performance within critical time intervals in real-time systems.

8. Design Practices for Critical Aspects:

- Each domain requires design practices that focus on the critical aspects, as illustrated in Figure 8.5.

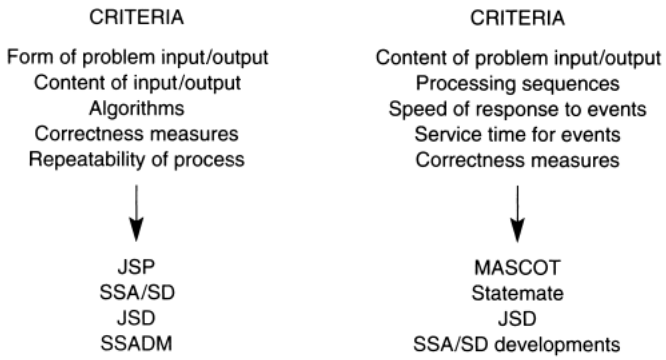


Figure 8.5 Some design criteria and related design methods.

DESIGN PROCESSES AND STRATEGIES

THE ROLE OF STRATEGY IN DESIGN METHODS

The role of strategy in design methods is pivotal, influencing every aspect of the software development process.

Introduction to Design Components:

- Design methods consist of three fundamental components: representation, process, and heuristics. The representation part involves various forms, determining the Design Virtual Machine (DVM). The process part, closely linked with representation, offers procedural guidelines for model development. Heuristics provide guiding principles.

Symbolic Procedural Model:

- A symbolic procedural model, as depicted in Figure 9.1, uses shapes - oblongs (representation), ovals (procedural steps), and arcs (step sequences). This model, demonstrated in Figure 9.2, maintains a hierarchical structure for transformations, emphasizing their interdependence.

Transformation and Elaboration Steps:

- Design processes encompass two essential steps: transformation and elaboration. Transformation involves creatively modifying the system model, often requiring major decisions. Elaboration focuses on restructuring within the current viewpoint, adding information or gaining insight into the design plan.

Detailed Process Model - Potts and Bruns:

- Potts and Bruns propose a detailed process model with five entity types and eight binary relationships. This model aids in the analysis of design decisions. Relationships include modifying artifacts, raising issues, reviewing artifacts, responding to issues, supporting/opposing positions, citing artifacts, and contributing to steps.

Strategic Influences on Design:

- Strategy plays a crucial role in shaping the initial model created to describe the problem. The choice of strategy is influenced by factors like the problem domain, specific constructional and behavioral characteristics, and likely forms of implementation.

Evolution of Software Design Ideas:

- A historical perspective reveals the evolution of software design ideas. Initially centered around physical modularity achieved through subprogram mechanisms, later concepts like information-hiding led to a more logical view of modularity.

Classification of Design Methods:

- Design methods fall into four broad categories: decompositional (top-down), compositional (entity-based), organizational (organization-driven), and template-based (standard strategy). Each strategy has unique characteristics influencing the design approach.

Handling Ancillary Aspects:

- Design strategies prioritize core activities in the early stages, deferring ancillary aspects like human–computer interactions (HCI) and error handling. These issues are addressed during later design refinement and elaboration stages.

Challenges in HCI and Error Handling:

- Incorporating HCI and error-handling strategies early in the design process can lead to model complexity. Designers face choices between incorporating these aspects later in the process, potentially conflicting with architectural decisions, or restarting the design process from an earlier stage.

Viewpoints-Centered Notation:

- To aid in modeling procedural steps, a 'viewpoints-centered' notation is introduced. This notation emphasizes the explicit distinction between transformation and elaboration steps, providing a structured approach to analyzing design methods.

In essence, the role of strategy in design methods is multifaceted, encompassing the structuring of design processes, influencing decision-making at various stages, and responding to the evolving landscape of software design concepts.

DESCRIBING THE DESIGN PROCESS- THE D-MATRIX

The D-Matrix notation serves as a powerful tool for describing the design process, emphasizing the synthesis of transformation, elaboration, and the viewpoints model.

The D-Matrix (Design Matrix) formalism is introduced to provide an abstract representation of the state of the design model during its evolution. It is method-independent and can describe both method processes and opportunistic design activities.

Composition of Design Elements:

- Design specifications are composed of 'design elements,' each described by a set of attributes representing the four major viewpoints: function, behavior, data model, and construction. The attributes depend on the design approach and architectural style.

Matrix Representation:

- At any point in the design process, the design model is represented by a matrix with columns for elements and rows for attributes corresponding to viewpoints. The matrix offers a compact notation for design states.

$$\begin{bmatrix} Db_1 & Df_1 & Dd_1 & Dc_1 \\ Db_2 & Df_2 & Dd_2 & Dc_2 \\ \vdots & \vdots & \vdots & \vdots \\ Db_n & Df_n & Dd_n & Dc_n \end{bmatrix}$$

Projection of Design Representations:

- Specific design representations, like STDs, can be seen as projections from the set of attributes along a single row, for example: $\{Ds_1, Ds_2, \ldots\ldots, Ds_n\}$.

Handling Hierarchical Representations:

- The notation can handle hierarchical representations (e.g., DFD or Statechart) by extending the matrix columns to represent sub-elements.

Describing Design Transformations:

- The design process involves a transformation, converting a requirements specification into a complete design description. The D-Matrix notation expresses this transformation in a concise form.

Elaboration and Transformation Steps:

- Design processes in procedural methods consist of elaboration (Ei) and transformation (Ti) steps. Elaboration involves restructuring within the current viewpoint, while transformation results in a significant modification, often a creative decision.

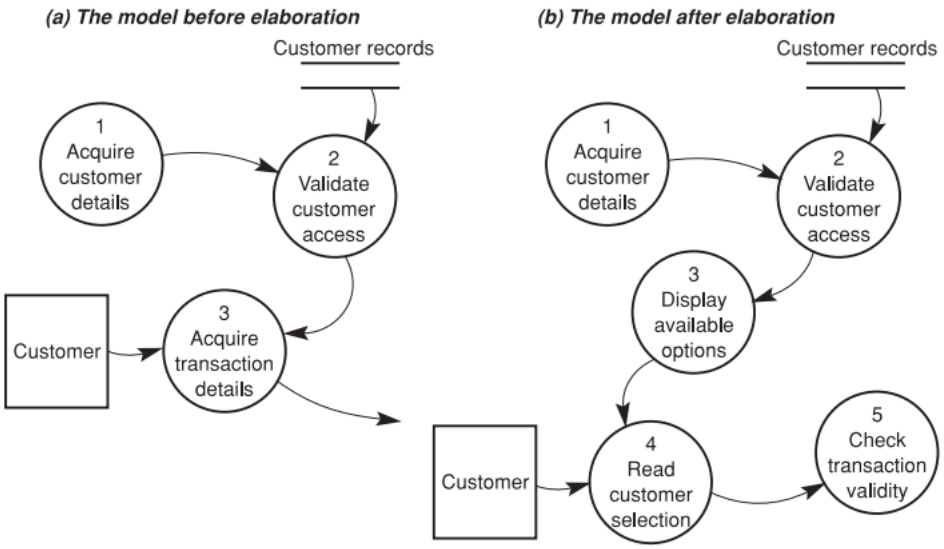


Figure 9.5 A typical elaboration step.

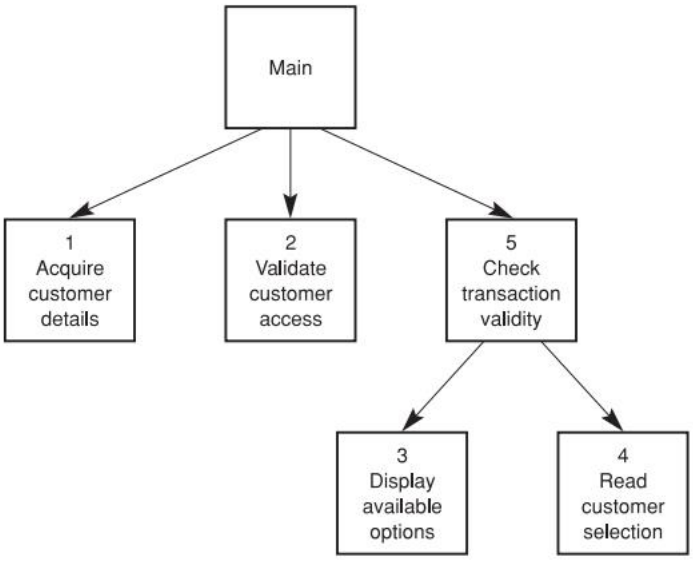


Figure 9.6 A typical transformation step. (Transforming the functional model of Figure 9.5(b) into a constructional viewpoint.)

Pragmatic Use of Notation:

- Despite its mathematical appearance, the notation is pragmatic and serves as a shorthand to make explicit how design methods guide the development of a design solution.

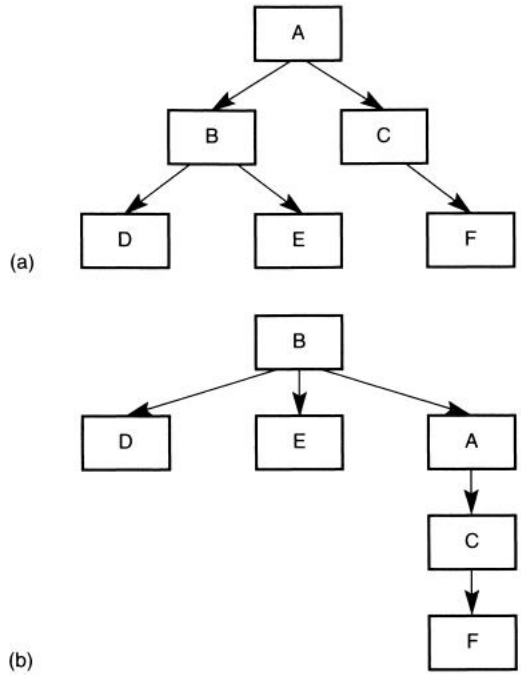
In summary, the D-Matrix provides a versatile and concise notation for capturing the evolving states of a design model, offering a method-independent framework for describing design processes and facilitating the analysis of design methods.

DESIGN BY TOP-DOWN DECOMPOSITION

The top-down (decompositional) approach to software design has a rich history, rooted in early programming languages like assembler and FORTRAN. These languages emphasized action-oriented structuring through subprograms but lacked support for complex data structures. Niklaus Wirth (1971) coined the term 'stepwise refinement' for this strategy, which involves subdividing the main task into smaller tasks until they are considered elemental enough to be implemented as subprograms. Early programming languages focused on action-oriented structuring, leading to a natural design strategy of breaking down tasks into subtasks.

Stepwise Refinement:

- Also known as 'divide and conquer,' stepwise refinement involves breaking down a problem into smaller subtasks until they are suitable for implementation.



Decompositional solutions to a simple problem. (a) Solution 1. (b) Solution 2.

- The success of this approach is highly dependent on how the original problem is described, influencing the designer's initial choice of subtasks.

- Small differences in the decomposition of a task can lead to functionally equivalent but structurally different solutions, introducing a degree of instability.

- Given the early commitment to basic structures, it is crucial to explore design options thoroughly at each stage of decomposition to avoid potential issues later in the process.

- Functional decomposition lacks a universally applicable criterion for determining the suitable size of a task, contributing to the challenging nature of 'wicked problems' in design.

Duplication Concerns:

- The strategy may result in duplication of functionality, especially when low-level operations are defined. Explicit checks are required to identify and resolve any duplications.

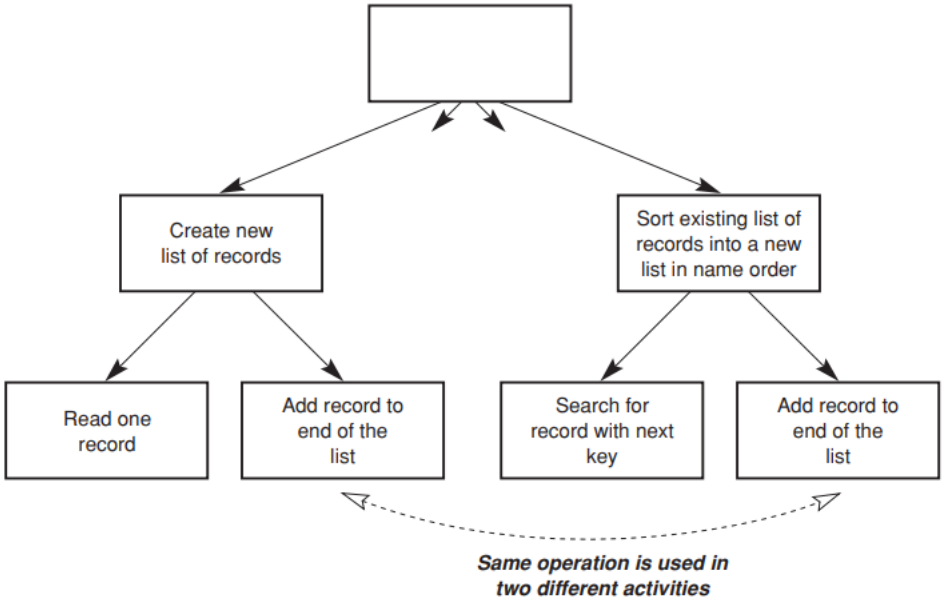


Figure 9.8 Example of duplication of low-level functions when using a decompositional structure.

- Despite its challenges, the top-down strategy has played a significant role in the development of design methods. Even in methods employing different strategies, initial design steps often involve functional decomposition.

Example Method - SSA/SD:

- The Structured Systems Analysis and Structured Design (SSA/SD) method, developed by Yourdon and colleagues, is an example of a method rooted in the top-down strategy. Even in more modern methods like SSADM, a top-down aspect remains in the process model, highlighting the continued relevance of this strategy in certain design contexts.

DESIGN BY COMPOSITION

In contrast to the top-down strategy that primarily focuses on identifying system operations, design by composition takes a reverse approach. Here, the emphasis lies on constructing the designer's model by developing descriptions of specific entities or objects recognized in the problem, along with their relationships. This compositional strategy involves grouping elements together to form a complete and detailed model of the solution. The top-down approach concentrates on identifying system operations, primarily considering the functional viewpoint during the design process.

Compositional Strategy

- In methods like JSD and object-oriented approaches, entities are identified by analyzing the initial problem description, and the design model is constructed by elaborating entity descriptions and their interactions.

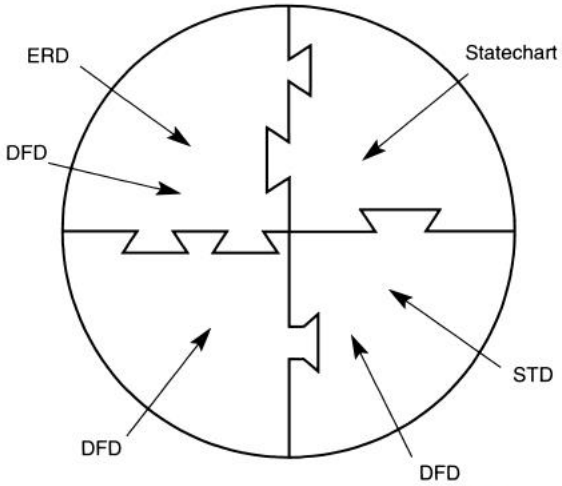


Figure 9.9 The compositional design strategy. An important feature is the design of the interfaces between the elements of the model.

D-Matrix Application:

- The D-Matrix is used to model these processes, with an increased likelihood of grouping columns of the matrix to create new abstractions and elaborating complex combinations of viewpoints.

JSP as a Compositional Method:

- JSP is cited as another method employing a compositional strategy, where the model of the problem is created using a data-modelling viewpoint, assembling descriptions of input and output data stream structures.

- The compositional strategy might be less directly intuitive, but this perception is argued to be influenced by familiarity. Training in compositional methods can offset this bias, as evidenced by experiences reported in the literature.

Advantages of Compositional Methods:

- More stability: Similar solutions are likely, regardless of the user, due to the design strategy relating the solution's structure to the problem's structure.

- Gradual progression: Design development tends to be more gradual compared to top-down methods.

- Better verification: Compositional methods facilitate a good verification process between design and the original specification, thanks to an explicit 'audit trail' between solution objects and problem objects.

The compositional strategy provides opportunities to use multiple viewpoints, leveraging important design concepts such as modularity and information-hiding. The emphasis on 'grouping' during the elaboration of a design model allows for grouping elements by considering different relations between them.

ORGANISATIONAL INFLUENCES UPON DESIGN

In the context of software design, an organizational design method is characterized by a process strongly influenced by non-technical factors, originating from the nature and structure of the organization implementing the method. Although these factors may not directly impact the representational part of the method, they can extend or formalize its effects. To comprehend these methods, it is essential to delve into the rationale behind their adoption and use.

- International agencies and government bodies are prominent customers for large software-based systems, ranging from defense real-time systems to applications like stock control or taxation.

- Organizations often provide staff with career structures aligned with overall organizational needs rather than project needs. Staff at all levels may be transferred in and out of project teams based on factors independent of project status.

- The British Civil Service serves as an example, where career progression is tied to fixed intervals, irrespective of a project's state. Similar structures exist in various countries.

<u>Standard methods</u>

-In the 1980s and 1990s, there was an increased emphasis on using 'standard' methods for analysis and design to control large software-based projects in such organizations.

- Standard methods offer benefits such as minimal disruption during staff changes, consistency in technical direction despite project manager changes, improved management of maintenance through standardized documentation, and better planning and control based on past experiences.

- Standard methods can be criticized for their bureaucratic nature, potentially hindering exploration of creative options and risking the loss of overall technical direction.

Examples of Organizational Methods:

1. SSADM (Structured Systems Analysis and Design Method)
2. MERISE (French equivalent to SSADM)
3. HOOD (Hierarchical Object-Oriented Design)
4. MASCOT (Modular Approach to Software Construction, Operation, and Test).

- The development and use of organizational methods appear to have been more common in Europe than in the USA or elsewhere, as per available information.

The organizational category is considered essentially independent of strategy. A method can be both top-down and organizational or compositional and organizational based on its domain of application.

These organizational design methods aim to align software development practices with the broader structures and processes of the organizations implementing them.