Progark

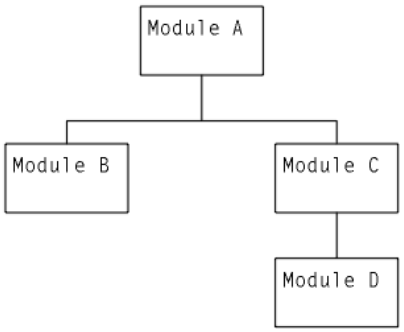
Oppsummering
av
Pensum

Chapter 1: Introduction

What is software architecture and what it isn't

- Software architecture: the set of sstructures needed to reason about the system, which comprimise software elements, relations among them, and properties of both.
- Structures, elements, relations, constraints, organisation and a system.
- **Structure:** a structure is a set of elements held together by a relation. There are thee types of architectural structures:

- **Module:** partitions a system into implementation units, which we call modules. Each module are assigned specific computional responsibilities, and are the basis of work assignments for programming teams.
- **Component and connector:** focus on the way the elements interact at runtime to do the system's function.
- **Allocation:** describe the mapping from software structures to the system's environments.
    - Organizational, developmental, installation, execution.



Architectural structures and views

- **View:** is a representation of a coherent set og architectural elements, as written by and read by system stakeholders. In short, a view is a representation of a structure.

- A module structure is the set of the modules
- A module view is the representation of the modules.

1) Module structures

- **Module structures** contain decisions as to how the system is to be constructed as a set of code or data units that have to be constructed or aquired.
- **Module structure views can answear questions like:**

- What are the primary functional responsability assiged to each module?
- What other software elements is a module allowed to use?

- **Some useful module structures:**

- **Decomposition structure:** how system is decomposed (modifiability)
- **Uses structure:** how units use other units (modifiability)
- **Layer structure:** units organized in layers (portability)
- **Class structure:** reasoning about collections of similar behavor (modifiability and reuse)
- **Data model structure:** static information structure

2) Component-and-connector structures

- **Component-and-connector (C&C)** structures embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).

- C&C structures views can help us aswear questions like:

- How does data progress through the system?
- What parts of the system can run in parallel?

- C&C structure:

- **Service structure:** the units are services that interoparate with each other by service coordination mechanisms such as SOAP.
- **Concurrency structure**

3) Allocation structures

- Allocation structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed.

- **Allocation structure views can help us to answear questions like:**

- What processor does each software element execute on?
- In what directories or files is each element stored during development, testing, and system building?
- What is the assignment of each software element to development teams?

- Allocation structures:

- Deployment structure
- Implementation structure:
- Work assignment structure

Architectural patterns

Structural “rules of thumb”

Chapter 2: Why is software architecture Important?

Thirteen reasons:

1. An architecture will anable a system's driving quality attributes
2. The decisions made in an architecture allow you to reaso about and manage change as the system evolves
3. The analysis of an architecture enables early prediction of a system's qualities
4. A documented architecture enhances communication among stakeholders
5. The architecture is a carrier if the earliset and hence most fundamental, hardest-to-change design decisions
6. An architecture defines a set of constraints on subsequent implementation
7. The architecture dictates the structure of an organization, or vice versa
8. An architecture can provide the basis for evoluntay prototyping
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule
10. An architecture can be created as a transferable reusable model that form the heart of a product line.
11. Architecture-based development forcuses attention on the assembly components, rather then simply on their creation
12. By restriction design atlernatives, architecture channels the creativity of developers, reducing design and system complexity
13. An architecture can be the fundation for training a new team member

Chapter 3: The many contexts of software architecture

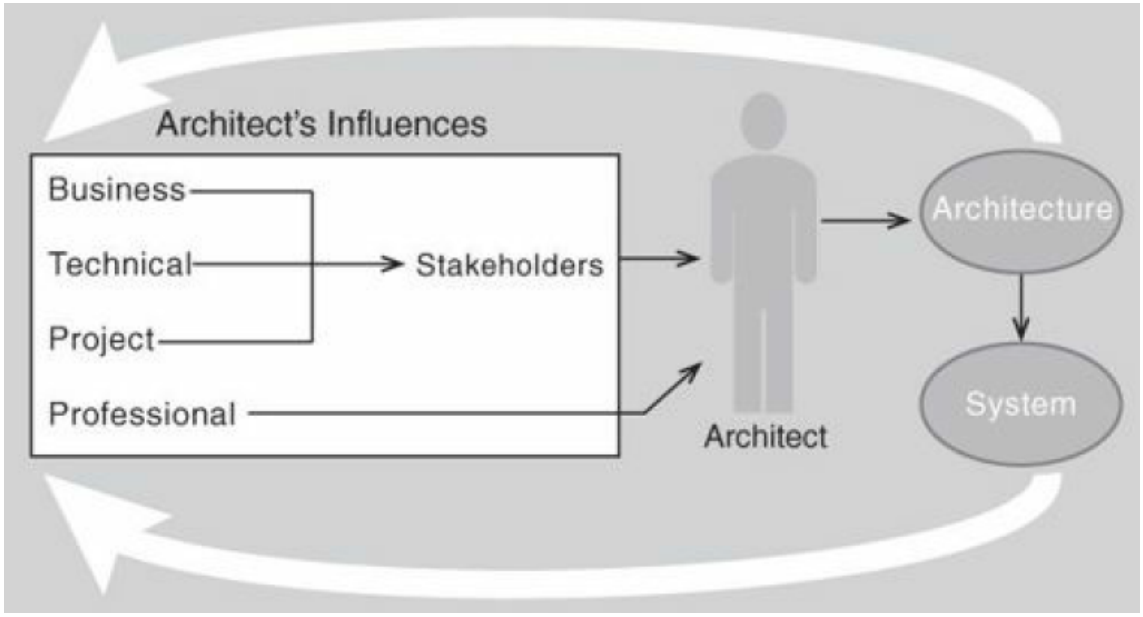
Architecture in a context:

- **Thnical context:** includes the achievement of quality attribute requirements
- **Project life-cycle context:** regardless of the software development methodology you use, you must perform specific activities
- **Business context:** the system created from the architecture must satisfy the business goals of a wide variety of stakeholders.
- **Professional context:** you must have certain skills and knowledge to be an architect, and there are certain duties that you must perform as an architect.

Stakeholders:

- A stakeholder is anyone who has a stake in the success of the system
- Stakeholders typically have different specific concerns that they wish the system to guarantee or optimize

How is architecture incfluenced/what do architectures influence?



Chapter 4: Understanding quality attributes

Architecture and requirements

- Functional requirements: state what the system must do, how it must behave or react to run-time stimuli
- Quality attribute requirements: annotate (quantify) functional requirement, e.g. how fast the function must be performed, how resilient it must be to erroneous input, how easy the function is to leard, etc.
- Constraints: a design decision with zero degree of freedom. A design decision that has already been made for you.

Functionality

Quality attribute considerations

- Example: If a functional requirement is “when a user presses the green button the options dialog appears”

- **A performance QA** annotation might describe how quickly the dialog will appeard
- **An availability QA** annotation might describe how often this function will fail, and how quickly it will be repaired
- **A usability QA** annotation might describe how easy it is to learn this funcrtion

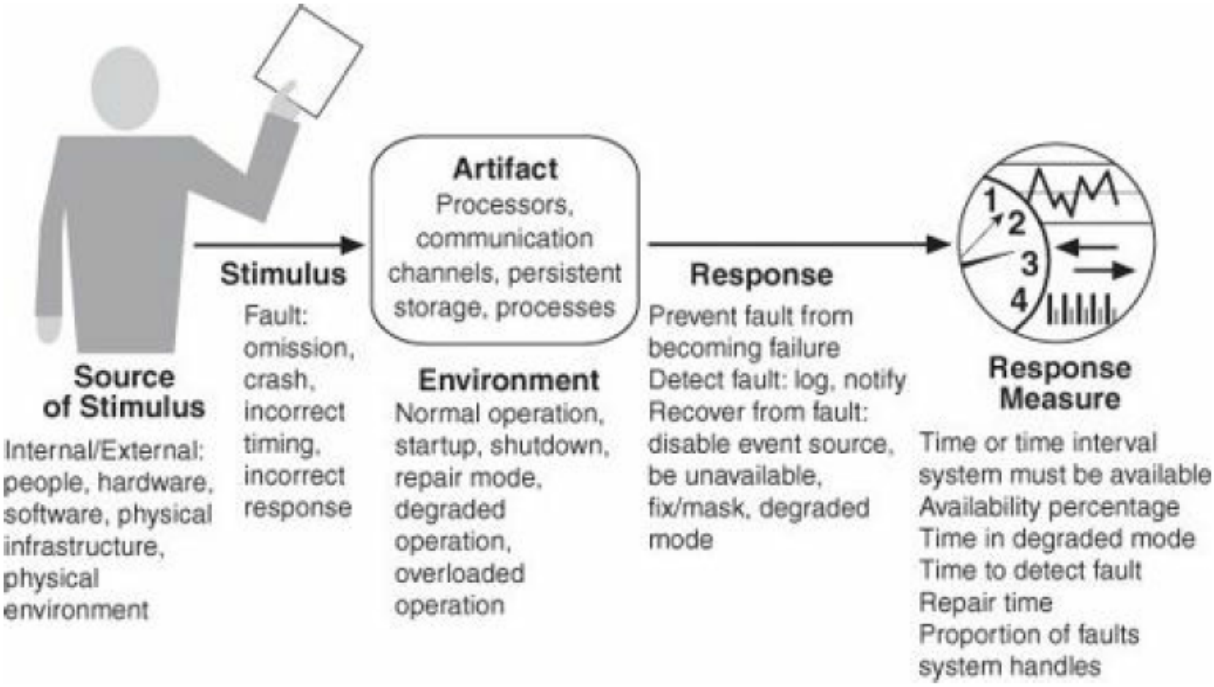
- Architectural or non architectural?

- **Usability:**
    - Making user interface clear and easy to use? NA
    - Ability to cancel operations? A
    - Provide radio buttoon or check box? NA
    - Undo operations? A
    - Re-use data previously entered? A
    - Screen layout? NA
- **Modifiability:**
    - How functionality is divided? A
    - By coding techniques within a module? NA
- **Performance:**
    - How algorithms are coded? NA
    - Communication necessary among comonents? A
    - What functionality is allocated to each component? A
    - Choice of algorithms to implement functionality? NA
    - How shared resources are allocated? A

Specifying quality attribute requirements

- Quality attribute scenarios:

- **Source of stimulus:** Human, computer, or other actor that generates the stimulus
- **Stimulus:** condition that needs to be considered when it arrives at a system
- **Environment:** the stimulus occurs within certain conditions
- **Artifact:** some artifact is stimulated
- **Response:** the activity undertaken after the arrival of the stimulus
- **Response measure:** when the response occur, it should be measured in some fashion so that the requirement can be tested.



Achieving Quality attributes through tactics

Guiding quality design decisions

- The seven categories of architectural design decisions are:

1. **Allocation of responsibilities**
    - identifying the important responsabilities including basic system functions, architectural infrastructure, and satisfaction of quality attributes
    - determing how these responsabilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors)
2. **Coordination model**
    - Determine the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency.
    - choosing the communication mechanisms that realize those properties. Important properties of the communication mechanisms inculude stateful vs. stateless, synchronous vs asynchronous, guaranteed vs non-guaranteed delivery, and performance-related properties such as throughput and latency.
3. **Data model**
    - Choosing the major data abstractions, their operations, and their properties.
    - metadata
    - organization of the data
4. **Management of resources**
    - determining which system element(s) manage each resource
    - identifying the resourcees that must be managed and determining the limits for each
5. **Mapping among architectural elements**
    - The assignment of runtime elements to processors
    - The assignment of items in the data model to data stores
6. **Binding time decisions**
7. **Choice of technology**

Chapter 5: Availability

What is availability?

- Availability refers to a property of software that it is there and ready to carry out its task when you need it to be
- This is a broad perspective and encompasses what is normally called reliability
- Availability builds on reliability by adding the notion of recovery (repair)
- Fundamentally, availability is about minimizing service outage time by mitigating faults.

Goal of availability tactics

- Availability tactics enable the system to endure faults so that services remain compliant with their specifications
- The tactics keep faults from becoming failures or at least bound the effects of the fault and make repair possible.

Availability general scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: ▪ Log the fault ▪ Notify appropriate entities (people or systems) Recover from the fault: ▪ Disable source of events causing the fault ▪ Be temporarily unavailable while repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g., 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

Chapter 6: Interoperability

What is interoperability?

- Is about the degree to which two or more systems can usefully exchange meaningful information.

Goal of interoperability tactics

- In order for two or more system to usefully exchange information, they must:
    - Know about each other. That is the purpose behind the locate tactics
    - Exchange information in a semantically meaningful fashion. That is the purpose behind the manage interfaces tactics. Two aspects of the exchange are
        - Provide services in the correct sequence
        - Modify information produced by one actor to a form acceptable to the secon actor.

Interoperability general scenario

Portion of Scenario	Possible Values
Source	A system initiates a request to interoperate with another system.
Stimulus	A request to exchange information among system(s).
Artifact	The systems that wish to interoperate.
Environment	System(s) wishing to interoperate are discovered at runtime or known prior to runtime.
Response	One or more of the following: ▪ The request is (appropriately) rejected and appropriate entities (people or systems) are notified. ▪ The request is (appropriately) accepted and information is exchanged successfully. ▪ The request is logged by one or more of the involved systems.
Response Measure	One or more of the following: ▪ Percentage of information exchanges correctly processed ▪ Percentage of information exchanges correctly rejected

Chapter 7: Modifiability

What is modifiability

- Is about change and our interest in it is in the cost and risk of making changes. To plan for modifiability, an architect has to consider three questions:
  - What can change?
  - What is the likelihood of the change?
  - When is the change made and who makes it?

Goal of modifiability tactics:

- Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes.

Modifiability general scenario:

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, . . .
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: ▪ Make modification ▪ Test modification ▪ Deploy modification
Response Measure	Cost in terms of the following: ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Calendar time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced

Chapter 8: Performance

What is performance?

- Performance is about time and the software system's ability to meet timing requirements
- When events occur - interrupts, messages, requests from users or other systems, or clock events marking the passage of time - time system, or some element of the system, must respond to them in time.
- Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence in discussing performance.

Goal of performance tactics:

- Tactics to control performance have as their goal to generate a response to an event arriving at the system within som time-based constraint..

Performance general scenario:

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

Chapter 9: Security

What is security?:

- Security is a measure of the system's ability to protect data and information from unnauthorized access while still providing access to people and systems that are authorized.
- An action taken against a computer system with the intention of doing harm is called attack and can take a number of forms.

- Security has three main characteristics, called CIA:

- Confidentiality
- Integrity
- Availability

- Other characteristics that support CIA:

- Authentication
- Nonrepudiation
- Authorization

Goal of security tactics:

- Secure installations have limited access to them. have means of detecting intruders, have deterrence mechanisms such as automatic locking of doors and have recovery mechanisms such as off-site backup.

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services, data within the system, a component or resources of the system, data produced or consumed by the system
Environment	The system is either online or offline; either connected to or disconnected from a network; either behind a firewall or open to a network; fully operational, partially operational, or not operational.
Response	Transactions are carried out in a fashion such that ▪ Data or services are protected from unauthorized access. ▪ Data or services are not being manipulated without authorization. ▪ Parties to a transaction are identified with assurance. ▪ The parties to the transaction cannot repudiate their involvements.

	▪ The data, resources, and system services will be available for legitimate use. The system tracks activities within it by ▪ Recording access or modification ▪ Recording attempts to access data, resources, or services ▪ Notifying appropriate entities (people or systems) when an apparent attack is occurring
Response Measure	One or more of the following: ▪ How much of a system is compromised when a particular component or data value is compromised ▪ How much time passed before an attack was detected ▪ How many attacks were resisted ▪ How long does it take to recover from a successful attack ▪ How much data is vulnerable to a particular attack

Chapter 10: Testability

What is testability?

- Software testability refers to the ease with which software can be made to demonstrate its faults through testing
- If a faut is present in a system, then we want it to fail during testing as quick as possibile
- For a system to be properly testable, it must be possible to control each component's inputs and then to observe its outputs

Goal of testability:

- The goal of tactics for testability is to allow for easier testing when in increment of software development

Testability general scenario:

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	One or more of the following: execute test suite and capture results, capture activity that resulted in the fault, control and monitor the state of the system
Response Measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage, probability of fault being revealed by the next test, time to perform tests, effort to detect faults, length of longest dependency chain in test, length of time to prepare test environment, reduction in risk exposure (size(loss) × prob(loss))

Chapter 11: Usability

What is usability?

- Is concerned with how easy it is for the user to accomplish a desied task and the kind of user support system the system provides
- The cheapest and easiest way to improve a system's quality
- Usability comprimises the following areas:
    - Learning the system features
    - Using a system efficiently
    - Minimizing the inpact of errors
    - Adapting the system to user needs
    - Increasing confidence and satisfaction

Goal of usability tactics:

Usability general scenario:

Portion of Scenario	Possible Values
Source	End user, possibly in a specialized role
Stimulus	End user tries to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system.
Environment	Runtime or configuration time
Artifacts	System or the specific portion of the system with which the user is interacting
Response	The system should either provide the user with the features needed or anticipate the user's needs.
Response Measure	One or more of the following: task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs

Chapter 12: Other quality attributes

Other important Quality attributes:

- **Variability:** is a special form of modifiability. It refers to the ability of a system and its supporting artifacts to support the production of a set of variants that differ from each other in a preplanned fashion

- Portability: is also a special form of modifiability. Portability refers to the ease with which software that build to run on one platform can be changed to run on a different platform

- Development distributability: is the quality of designing the software support distributed software development

- Scalability: horizontal scalability refers to adding more rescourses to logical units such as adding another server to a cluster. Vertical scalability refers to adding more resources to a physical unit such as adding more memory to a computer.

- Deployability: is concerned with how an exacutable arrives at a host platform and how it is invoked.

- Mobility: deals with the problems of movement and affordances of a platform (e.g size, type of display, type of input device, etc).

- Monitorability: deals with the ability of the operations staff to monitor the system while it is executing.

- Safety: software safety is about the software's ability to avoid entering states that cause or lead to damage, injury, or loss of life, and to recover and limit the damage when it does enter a bad state.

- Conceptual integrity: refers to consistency in the design of the architecture. It contributes to the understandability to the architecture.

- Marketability: some systems are marketed by their architectures, and these architectures sometimes carry a meaning all their own, independent of what other quality attributes they bring to the system (e.g service-oriented, cloud-based, etc).

- Quality in use: qualities that pertain to use of the system by various stakeholders.

Chapter 13: Patterns and Tactics

What is a pattern?

- An architectural pattern establishes a relationship between:
  - **A context:** a recurring, common situation in the world that gives rise to a problem
  - **A problem:** the problem, appropriately generalized, that arises in the given context
  - **A solution:** a successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:
    - a set of element types (for example, data repositories, processes, and objects)
    - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
    - A topological layout for the components
    - A set of semantic constraints covering topology, element behaviour, and interaction mechanisms.

Pattern categories:

1. **Module patterns** (layered, ..)
2. **Component and connector patterns** (broker, MVC, pipe-and-filter, client-server, peer-to-peer, Service-oriented architecture, publish-subscribe, shared-data,
3. **Allocation patterns** (map-reduce, multi-tier,..)

Relationships between tactics and patterns:

- Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom
- MVC, for example utilizes the tactics:
  - Increase semantic coherence, Encapsualtion, Use an intermediary and Use run time binding

Tactics augment patterns:

- Patterns solve a specific problem but are natural or have weaknesses with respect to other qualities. Consider the broker pattern:
  - May have performance bottlenecks, but using tactics to increase resources will help performance
  - May have a single point of failure, but by using a tactic to maintain multiple copies will help availability

How does this process end?

Each use of tactic introduces new concerns. Each new concern causes new tactics to added. Are we in an infinite progression? - No, eventually the side-effects of each tactic become small enough to ignore.

Guest lecture - Design patterns

What is design?

Purpose of design

- Create reusable code?
- Create maintainable and extensible code?
- Portability, flexibility, transparency?
- Keywords:
    - High cohesion: doing one thing.
    - Loose coupling:

Design patterns vs. architectural patterns vs. idioms

- Design patterns are a level below architectural patterns
- The lowest level are idioms
    - low/level pattern specific to a programming language
    - Describes how to implement particular aspects of components or the reletionships between them using features of given language

GRASP patterns

- Describe fundamental principels of assigning responsabilities to objects, expressed as patterns
- Examples
    - **Expert:** assign a responsability to the information expert - the class that has the information necessary to fulfil the responsability
    - **Creator:** Assign class B the responsability to create an instance of class A if B aggregates, contains, records instances, or closely uses A Objects, or has the initalising data that will be passed A when it is created.
    - **High cohesion:** assign a responsability so that cohesion remains high
    - **Loose coupling:** Assign a responsability so that coupling remains low
    - **Controller:** assign the responsability for handling a system event massage to a class

Design pattern types - GOF:

- **Creational (e.g Factory, Singleton)**
  - Encapsulates creational knowledge for an object in a method, a class or another object
- **Structural (e.g Composite, Proxy, Facade)**
  - Concerned with how classes and objects are composed to form larger structures
  - Structural class patterns use inheritance to compose interfaces or implementations.
- **Behavioural (e.g Observer, State, Strategy)**
  - Are concerned with algorithms and the assignment of responsabilities between objects
  - Describes not just patterns of objects or classes but also the patterns of communication between them.

(OBS: the design patterns are in a separate document!)

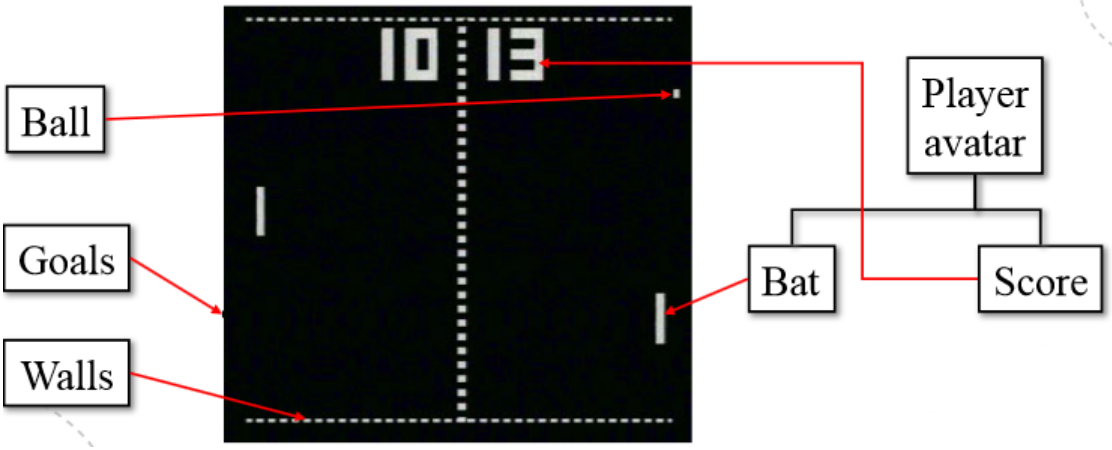
Pattern languages:

- Some patterns may be applied in a sequence from one to another
- A pattern language connects patterns together
- A pattern language is a collection of patterns that build on each other to generate a system

Extra chapter: Project and game architecture

Creating a game architecture

1. **Find tokens:**
    - Tokens are objects related to the gameplay: Playable objects, non-playable character (NPC), game environment objects, environment, score, etc.
    - Tokens in pong:



2. **Analyse interaction and events:**
    - Create token interaction matrix
    - Trace events in the game (event diagram)
    - Create finite-state machine diagrams for NPCs, game world, etc
    - Starting point for component&connector view as well as logical view

3. **Create logical view using tokens:**
    - Tokens can be used to sketch gameplay logical view based on token interaction matrix.

	Bat	Ball	Wall	Goal	Score
Bat	X				
Ball	Collision event: Deflection	X			
Wall	Collision event: Stop	Collision event: Deflection	X		
Goal	X	Collision event: Trigger Goal event	X	X	
Score	X	X	X	Goal event: Goal score	X

Architectural patterns (with focus on robot and game)

- **Pipe-and-filter:** used to manipulate a data strem
    - Typical usage: 3D graphics engine, data conversion, compiler, workflow systems, Game loop, Intercepting filter pattern, AI approach
    - Positive: Vertical division of problem into units, Each unnit is responsible for well-defined task, Simple process flow, Logical division of components, Easy to replace with better filters
    - Challenging: Real-time performance can be a problem depending on the filters and One filter (bottleneck) can delay the whole process
- **Layered pattern:** this pattern divide the different parts of the system into different abstraction levels.
    - Positive: Divide complex problems into simpler problems through abstraction, each part does not to handle the whole complex model, each layer adds more complexity, fits to incremental implementation
    - Challenging: the layers must be planned carefully, can be hard to decide which layer to put functionality into, introduce performance overhead, require that the problem can be decomposed into distinct sub-problems.
- **Blackboard:** is a central database where all components can publish and subscribe info objects. Components can place observers that look for certain characteristics. Often transaction management or info objects. Info objects can be inserted, duplicated, read, and removed.
    - Positive: Exceptions, wiretrapping and monitors can be implemented as modules that watch the database, support runtime-update of components, can provide security, transaction support,
    - Challenging: All control flow must go via blackboard, even if direct interaction is more natural, possible challenge of performance due to everything through blackboard, blackboard is the bottleneck of the system.
- **Task control pattern**
    - Task tree
    - Hierarchical task tree
- **MVC**

Chapter 14: Quality Attribute Modeling and Analysis

Life-Cycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based analogy	Low	Low–High
Requirements	Back-of-the-envelope	Low	Low–Medium
Architecture	Thought experiment	Low	Low–Medium
Architecture	Checklist	Low	Medium
Architecture	Analytic model	Low–Medium	Medium
Architecture	Simulation	Medium	Medium
Architecture	Prototype	Medium	Medium–High
Implementation	Experiment	Medium–High	Medium–High
Fielded System	Instrumentation	Medium–High	High

Chapter 15: Architectures in Agile Projects

Chapter 16: Architecture Requirements

- Gathering architecturally significant requirements from:
  - Requirement documents
  - Interviewing stakeholders
  - Understanding business goals
  - Utility tree
  - Trying methods together

Capter 17: Designing an architecture

Extra chapter: Software architecture documentation

- Definitions:

- Software architecture: the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both
- Architectural description: a collection of products to document an architecture
- Architecural view: a presetation of a whole system from the perspetive of a related set of concerns.

- Who is interested in the architecture documentation?

- architect, user, requirement engineer, designer, implementers, testers, integrators, managers, etc.

- IEEE 1471 standard: prescribe the least common multiple on what should be included in a complete *architectural description*:

- Architectural documentation: AD ID, version, overview information
- Stakeholders and their concerns
- Selection of architectural viewpoints (views): what views will be in the document, which notations will be sed (UML), etc
- Architectural views: inconsistencies between views
- Architectural rationale: a motivation/reason for the design

- Views as presented in the book:

- Module views
- Component and connector views
- Allocation views

- 4+1 view model: specifies how software architecture can be described using 5 different, but related, viewpoints

- **Logical view:** Object model of the design
- **Process view:** Captures the concurrency and synchronization process
- **Physical view:** describe software into hardware mapping
- **Development:** Static organization of software in its development environment
- **Scenario:** illustrate a few selected use cases

View	Logical	Process	Development	Physical	Scenarios
Components	Class	Task	Module, Subsystem	Node	Step, Scripts
Connectors	Association, inheritance, containment	Redez-vous, Message, Broadcast, RPC, etc.	Compilation dependency, “with” clause, “include”	Communication medium, LAN, WAN, bus etc.	
Containers	Class category	Process	Subsystem (library)	Physical subsystem	Web
Stakeholders	End-user	System designer, integrator	Developer, manager	System designer	End-user, developer
Concerns	Functionality	Performance, availability, SW fault-tolerance, integrity	Organization, reuse, portability, line-of-product	Scalability, performance, availability	Understand-ability

Chapter 20: Reconstruction and conformance

What is reconstruction of a software architecture?

Reconstrction process:

1. **Information extraction:** extract information from various sources
2. **Database construction:** convert information from step 1 to a database format
3. **View fusion:** combines information in the database
4. **Reconstruction:** building abstractions and various representations