Software Engineering 2
Alloy

```
-- show predicate
pred show {}
-- will try to find an example
run show for 3 but 1 Book

pred show2[b: Book] {
    #b.addr > 1
    #Name.(b.addr) > 1
}

-- dynamic analysis
pred add[b, b': Book, n: Name, a: Addr] {
    b'.addr = b.addr + n -> a
}
pred showAdd[b,b': Book, n: Name, a: Addr] {
    add[b, b', n, a]
    #Nmae.(b'.addr) > 1
}
run showAdd

-- assertations and counterexamples
assert delUndoesAdd {
    all b, b', b'': Book, n: Name, a: Addr |
        add[b, b', n, a] and del[b', b'', n] ↩
            implies
            b.addr = b''.addr
}
-- will try to find a counterexample
check delUndoesAdd for 3

-- functions
fun lookup[b: Book, n: Name]: set Addr {
    n.(b.addr)
}
assert addLocal {
    all b,b': Book, n,n': Name, a: Addr |
        add[b,b',n,a] and n != n' implies
            lookup[b,n'] = lookup[b',n']
}

-- set operators
Name = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecUsed = {(N2)}
Name - RecUsed = {(N1)}
RecUsed in Alias = false
RecUsed in Name = true
Name = Group + Alias = true

Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}
Name->Addr = {(N0,A0),(N0,A1),(N1,A0),(N1,A1)}

-- dot join
p = {(a,b), (a,c), (b,d)}
q = {(a,d,c), (b,c,c), (c,c,c), (b,a,d)}
p.q = {(a,c,c),(a,a,d)}
   -- (a,b) . (b,c,c) => (a,c,c)
   -- (a,b) . (b,a,d) => (a,a,d)
   -- (a,c) . (c,c,c) => (a,c,c)
   -- match last col of p with first of q and ↩
       remove them
-- box join
a[b] = b.a
-- for binary relations
~r = r transposed
^r = r + r.r + r.r.r + ...
*r = iden + ^r
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}
~next = {(N1,N0), (N2,N1), (N3,N2)}
^next = {(N0, N1), (N1, N2), (N2, N3),
         (N0, N2), (N1, N3), (N0, N2)}
*next = {(N0, N0), (N1, N1), (N2, N2)} + ^next

-- restriction and override
Name = {(N0), (N1), (N2)}
Alias = {(N0), (N1)}
Addr = {(A0)}
address = {(N0, N1), (N1, N2), (N2, A0)}

address :> Addr = {(N2, A0)} -- range restriction
Alias <: address = {(N0, N1), (N1, N2)} -- domain ↩
    restriction
address :> Alias = {(N0, N1)}
workAddress = {(N0, N1), (N1, A0)}
address ++ workAddress = {(N0, N1), (N1, A0), (N2,↩
     A0)} -- override
m' = m ++ (k -> v) -- update map with (k,v)

-- set definition
{n: Name | no n.^address & Addr}

-- if-else
f implies e1 else e2
-- let binding
all n: Name |
    let w = n.workAddress, a = n.address |
        (some w implies a = w else a = n.↩
            homeAddress)

-- enum
abstract sig Color {}
one sig GREEN extends Color {}
one sig RED extends Color {}
all s: Semaphore | s.color = RED

-- seq
sig Path { positions: seq Position }
p.positions -- Int -> Position
p.positions[i] -- access the i-th position
p.positions.inds -- set of all the indexes
univ.(p.positions) -- set of all the elements
(p.positions).hasDups -- true if there are ↩
    duplicates
```

World and Machine

The *machine* is the portion of system to be developed (software and hardware). The *world* (aka the environment) is the portion of the real world affected by the machine.

Phenomena can be *shared* between world and machine. Shared phenomena can be controlled by the machine and observed by the world, or viceversa.

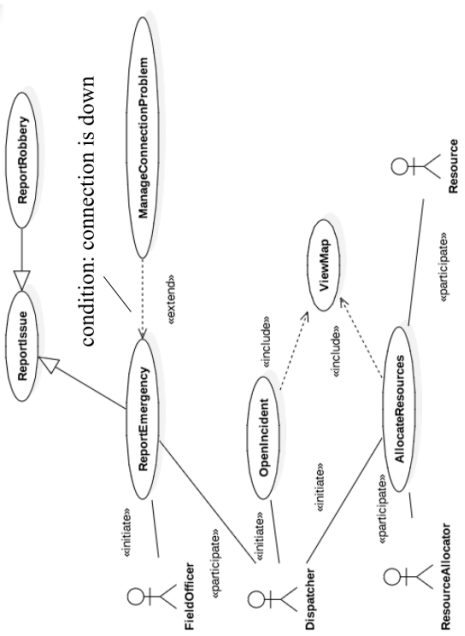
Goals are prescriptive assertions formulated in terms of world phenomena.
Domain assumptions are descriptive assertions assumed to hold in the world.
Requirements are prescriptive assertions formulated in terms of shared phenomena.

Example of interaction betweem world and machine:

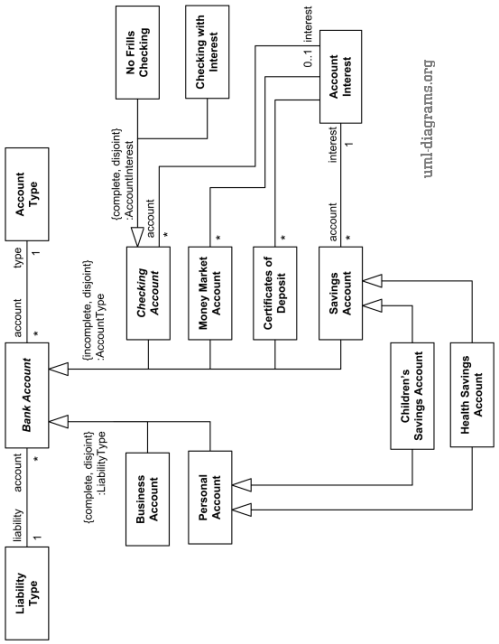
Phenomenon	Shared	Who controls it
User wants to buy some milk	N	W
User inserts a coin in the machine	Y	W
The machine compares the inserted coin with the last received one	N	M
The machine rejects the inserted coin	Y	M
The machine accepts the inserted coin	Y	M
User inserts a fidelity card	Y	W
The machine checks and accepts the fidelity card	Y	M
The machine sees that amount needed to buy a bottle of milk is reached	N	M
The machine delivers the bottle of milk	Y	M
The machine updates the current amount of money	Y	M
The user goes home with the milk	N	W
The user wants to receive the money back	N	W
The user asks for the money back	Y	W
The machine delivers the amount of money to the user	Y	M
The machine resets the money count	N	M
The operator sets the current number of bottles in the machine	Y	W
A milk sensor signals the milk in the machine is finishing	Y	W
The machine decreases the counter of the current number of bottles	N	M
The machine goes out of service	Y	M

UML

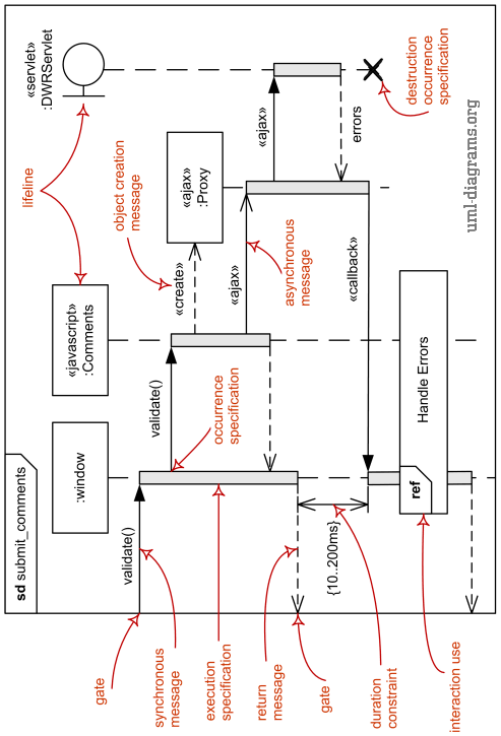
Use case diagram



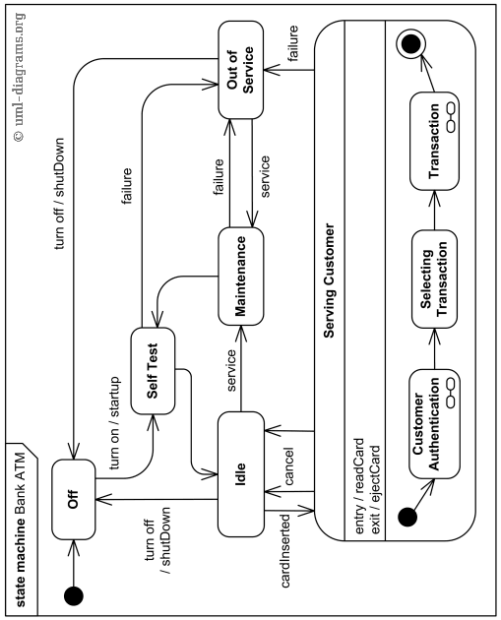
Class diagram



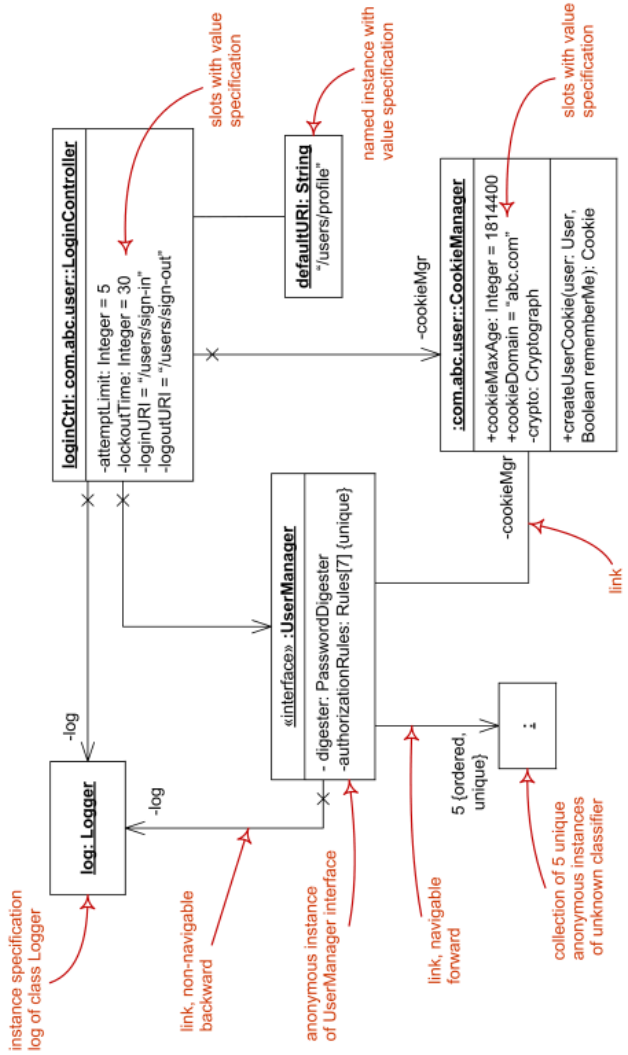
Sequence diagram



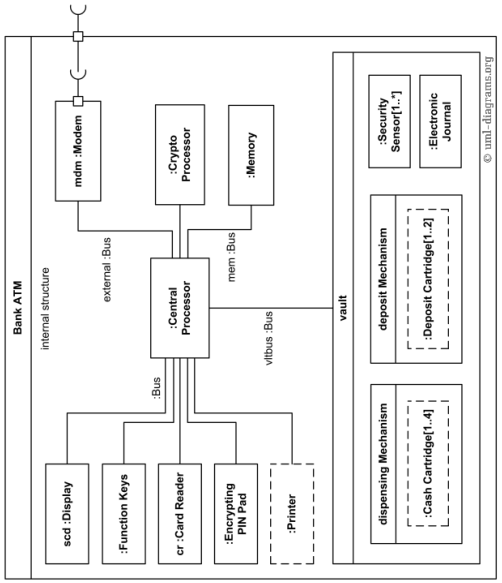
State diagram



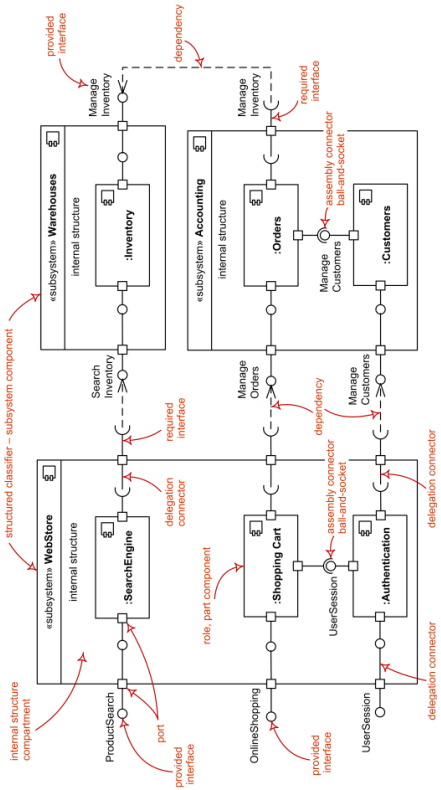
Object diagram



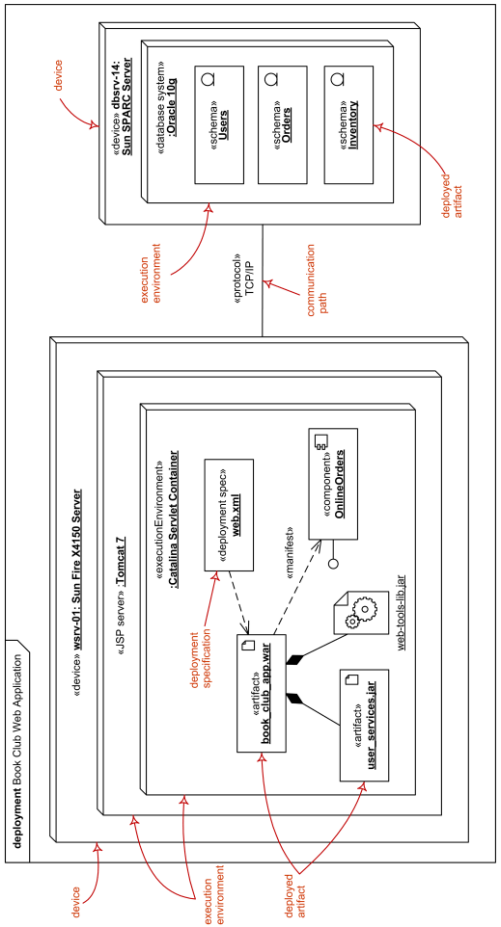
Composite Structure diagram



Component diagram



Deployment diagram



Design

Architectural styles

Layered style

The system is organized through abstraction levels. One level depends on some levels and provides functionalities to other levels. Example: onion-ring structure, OSI and TCP/IP model.

Client-server

One of the most used architectural styles for distributed applications. Server is invoked to provide some services, it is passive. Client initiates communication and has an active role. Client-server can be tiered.

Event-based systems

Also called publish-subscribe, communication in implicit through the generation and subscription of events. There is no explicit target in an event; the event is delivered to every component subscribed to that kind of event. Usually asynchronous, it is a reactive paradigm (computation begins when receiving a message) and there is loose coupling.

Service-oriented architecture

Also called SOA, composed of three actors:

- Service requestors: they request a service.
- Service providers: they provide a service.
- Intermediaries (brokers, registries): they give information (meta-data) about services. They facilitate the communication between service requestors and providers.

Three main phases:

- service description publication;
- service discovery;
- service binding.

Microservices

Based on the use of stateless components, which are easy to replicate and scale very well by simply adding more instances. No data is shared between different microservices, each microservice has its database.

Cloud patterns

Based on the use of stateless components. Resource and component instances can be added and removed regularly, based on demand. All the state is kept on persistent storage; if a stateless component fails, no data is lost and the computation can be made by another component (stateless components are indistinguishable). **Elastic component**
Used when there are multiple elastic compute nodes. There is a component whose job is to monitor the load and to add or remove instances of the components

Elastic load balancer Similar to elastic components, resources to be used determined by the requests coming to the load balancer (in the elastic component scenario only data about utilization of the compute node is used).

Design principles

- Divide and conquer.
- Keep the level of abstraction as high as possible.
- Increase cohesion where possible.
- Reduce coupling where possible.
- Design for reusability.
- Reuse existing designs and code.
- Design for flexibility.
- Anticipate obsolescence.
- Design for portability.
- Design for testability.
- Design defensively.

Design process

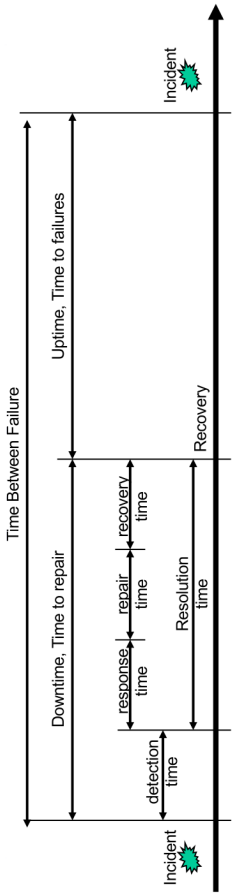
Top down approach

First design the very high level structure of the system. Then gradually work down to detailed decisions about low-level constructs. Finally arrive at detailed decisions such as the format of particular data items or the individual algorithms that will be used.

Bottom-up approach

Make decisions about reusable low-level utilities. Then decide how these will be put together to create high-level constructs.

Analysis of architectures



Time of occurence: time at which the user becomes aware of the fault.
Detection time: the service provider is informed.
Response time: time required by the service provider to respond to the user.
Repair time: time required to restore the service or the component that caused the fault.
Recovery time: time required to restore the system.

Mean Time To Repair (MTTR): average time between the occurrence of a fault and recovery.
Mean Time To Failures (MTTF): average time between the recovery of an incident and the occurence of the next one.
Mean Time Between Failures (MTBF): average time between the occurrence of two consecutive faults.

$MTBF = MTTR + MTTF.$

Availability
Probability that a component is working properly at a given time.

$$A = \frac{MTTF}{MTTF + MTTR}$$

Availability in series: $A = \prod A_i$
Availability in parallel: $A = 1 - \prod(1 - A_i)$

Reliability
Probability that a component has always been working properly during a time interval $(0, t)$.

$$R = e^{-\lambda t} \qquad \lambda = \frac{1}{MTTF}$$

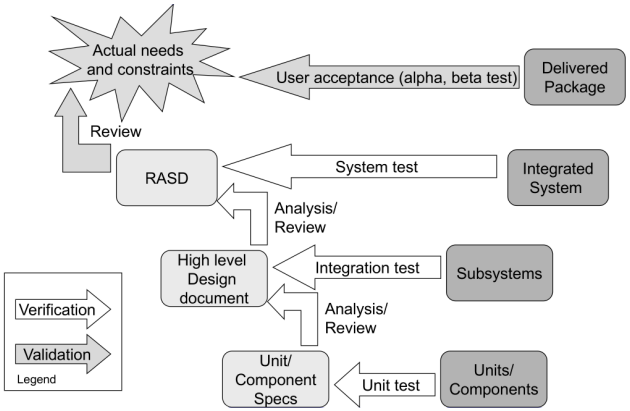
Testing and Verification

- Verification: is the program right (correct)?
- Validation: is the right program (what the client wants)?

Everything must be verified (specifications, design, test data etc.) along the entire development process.
Failures are usually result of faults introduced by human error.
Human Error → System fault → System failure
Verification and validation start as soon we decide to develop a product, design must permit testing of the system.
Difficulties:

- Impossible to develop error free software
- Properties may bu subjective
- Properties may be implicit or not clearly stated
- Goals not reasonable

Verification model



Analysis

Software is not executed, so analysis can be made even at early stages when software cannot be executable (yet). The two main approaches are (manual) inspection and automated static analysis.

Review, Walkthrough, Inspection

Formal evaluation in which software (or plans, design, documentation, everything) is examined to find faults, violations of standards etc.
Objective is finding errors in the product, not fixing them or evaulting the producer.
Walkthrough Producer presents product and the reviewers comment on the correctness and consistency.

- Informal review
- Reviewers are experts in the domain
- Subject is correctness of product from POV of experts
- Leader of discussion/session controller is the producer

Inspection

- Formal review
- Reviewers are trained, professional inspectors
- Subject is correctness of product according to given checklist
- Leader of discussion/session controller is official moderator of review team

Various roles:

- Moderator: plans meeting, chooses participants, controls all the process
- Readers and testers (inspectors): read code and look for flaws
- Author: passive, only answer to questions
- Scribe (recorder)

Session are at most 2 hours, max 150 lines per hour. Defects are only logged, not fixed.
Modern code review Code visible to everyone, review facilitated by various tools. Helps in exchanging and recording ideas.

Static Analysis

Based on identification of pairs of variables definitions and use. Typically used by compilers to check for possible errors and to make optimizations. Pessimistic approach, some problems may actually be false positives.

1. Derive control flow graph
2. Derive def and use sets for every node
3. Identify pairs of def-use

Can be used to check if variable is guaranteed to be initialized or if variable is never used.
Symbolic execution Values are expressed over symbols, executing statements computes new expressions. In case of branches, execution is performed only for a specified path.

Testing

Test case is a set of inputs of the system, along with the expected outcome (given hypothesis on state of the system).
Test set is a set of test cases.
Test cases can be generate randomly or systematically.

Unit testing

Conducted directly by the developers on single units of code. Component may not work in isolation, so *drivers* and *stubs* are needed (this is called **scaffolding**). Unit testing should be done as soon as possible.

Integration testing

Aimed to test interfaces and modules interactions. Example of integration faults are:

- Inconsistent interpretation of parameters
- Violations of capacity or size limits
- Side effects on resources
- Omitted or misunderstood functionality
- Nonfunctional properties (e.g. performance issues)
- Dynamic mismatches

Integration plan is an important part of the test plan, which is part of the project plan.

Big Bang testing

All integration testing done at the end, no scaffolding required. Very bad, it has an high cost of repair (bugs found early cost less to be fixed).

Incremental testing

Integration testing is done while component are released, even at early stages.

Testing strategies

- Top-down (requires stubs)
- Bottom-up (requires drivers)
- Thread (develop one function at a time)
- Critical modules (start by riskier modules, to verify feasibility)

System testing
Testing of the whole system, both functional and non-functional.

- Performance testing (identify bottlenecks and benchmarking)
- Load testing (expose bugs such as memory leaks, identify upper limits of components)
- Stress testing (see how the system behaves in case of failures or sudden change of load and resources)

Testing techniques
White box Used usually for unit testing, based on knowledge of structure of the system.

Black box Used to test integration and the whole system. Used to check if expected behavior is the behavior of the software (*Model-based testing*).

Capture and reply First test is manual (so it is costly) but it is recorded. Next time the test is an automatic replay of the first test, for which we know the outcome (used for example for auto-regression testing on GUI).

Black box testing a state machine
If the system acts like a state machine (there are states and transactions between them) there are 2 criterion of coverage:

- Coverage of States: the test cases must collectively reach all the states
- Coverage of Transactions: the test cases must collectively reach all the transactions

Project Management
A project is a temporary organization that has a specific and unique goal and usually a budget (costs, materials, resources).

Project Management is used to manage (plan, monitor, control) the scope, time and cost in order to make the project successful.

Project management processes
1. Initiating
2. Planning
3. Executing
4. Monitoring and controlling
5. Closing

1. Initiating
- Define the project
- Define initial scope
- Estimate cost and resources
- Define the stakeholders

2. Planning
- Scope management plan: defining, validating and controlling scope
- Schedule management plan: how schedule is developed, managed, executed and controlled
- Cost management plan
- Quality management plan: quality standards, quality assurance and control
- Change management plan
- Communication management plan
- Risk management plan: identify risks and plan responses

Risk Management
Risk is an uncertain event that if occurs can impact the achievement of objectives.

- Risk cause: source (or driver) of the risk
- Risk event: uncertain event that might follow the cause
- Risk effect: how the objectives might be affected by the risk event

Steps for risk management:

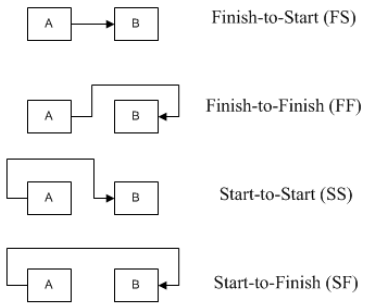
1. Define roles and responsibilities
2. Identify possible risks
3. Give probability (high, low, medium) to each risk
4. Develop a risk response plan
5. Define a budget for unknown risks

Type of risks:

- Project risks (threaten project plan)
- Technical risks (threaten quality and timeliness of product)
- Business risks (e.g. market risk, strategic risk, sales risk, management risk, budget risk)

Schedule planning
Tasks are activities which must be completed to achive the project goal. Milestones are points in the schedule where progress can be assessed. Deliverables are work products delivered to the customer (e.g documents).

1. Break down project in tasks
2. Define dependencies between tasks
3. Define lag time between dependencies (even negative)



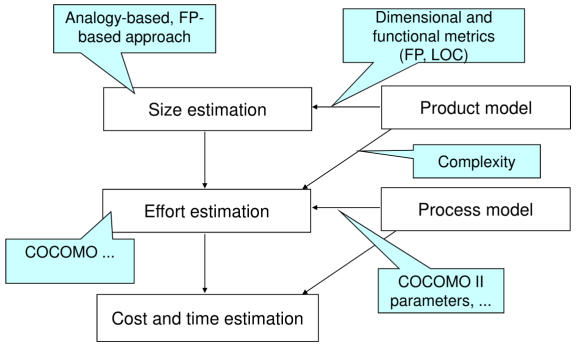
The critical path is a sequence of tasks that runs from the start to the end of the project. Changes to task on the critical path changes the project finish date. A task is critical if it cannot float earlier or later.

Contraints can be:

- Flexible (as soon as possible)
- Partial flexible (start no earlier than / finish no later than)
- Inflexible (must occur on specific time interval)

Cost and Effort estimation
Two types of techniques: experience-based techniques (based on past projects) and algorithmic cost modelling (formulaic approach).

Estimation changes based on number of COTS and components, programming language used, distribution of the team etc. Exact size can only be known when project is finished.



Function Points
Characterize software dimension basing on functionalities.

Function types:

- Internal Logical File (ILF): homogeneous set of data used and managed by the application.
- External Logical File (ELF): homogeneous set of data used by the application but maintained by others.
- External Input: elementary operation to elaborate date coming from external environment.
- External Output: elementary operation that generates data for the external environment (usually includes elaboration of LF).
- External Inquiry: elementary operation that involves input and output (without significant elaboration of LF).

Every function type is given a weight based on the complexity (simple, medium, complex). The sum is the Unadjusted Function Points (UFP). Function points can be used to estimate $LOC = AVC \cdot UFP$, where AVC is language dependent.

COCOMO II
Two cases: *Post-Architecture* (extension of existing product) or *Early Design*.

$$PM = A \cdot Size^E \prod EM_i$$

Where:

- PM is Person-Month
- $A = 2.94 \frac{PM}{KSLOC}$
- $Size$ is the estimated size of the project in $KSLOC$ (Kilo-Source Lines of Code)
- EM is *Effort Multiplier* (derived from *Cost Drivers*)
- E is aggregation of five *Scale Factors*

Scale Factors

- Precedentedness: high if product is similar to previously developed projects.

- Dev. Flexibility: high if there are no specific constraints conform to pre-established requirements and external interface specs.
- Risk resolution: high if we have a good risk management plan, clear budget and schedule.
- Team cohesion: high if stakeholders are able to work in a team.
- Process maturity: refers to a well known method for assessing maturity of software organization (CMM, CMMI).

Scale Factors		Very Low	Low	Nominal	High	Very High	Extra High
PREC	SF_j:	thoroughly unprecedented 6.20	largely unprecedented 4.96	somewhat unprecedented 3.72	generally familiar 2.48	largely familiar 1.24	thoroughly familiar 0.00
FLEX	SF_j:	rigorous 5.07	occasional relaxation 4.05	some relaxation 3.04	general conformity 2.03	some conformity 1.01	general goals 0.00
RESL	SF_j:	little (20%) 7.07	some (40%) 5.65	often (60%) 4.24	generally (75%) 2.83	mostly (90%) 1.41	full (100%) 0.00
TEAM	SF_j:	very difficult interactions 5.48	some difficult interactions 4.38	basically cooperative interactions 3.29	largely cooperative 2.19	highly cooperative 1.10	seamless interactions 0.00
PMAT	SF_j:	SW-CMM Level 1 Lower 7.80	SW-CMM Level 1 Upper 6.24	SW-CMM Level 2 4.68	SW-CMM Level 3 3.12	SW-CMM Level 4 1.56	SW-CMM Level 5 0.00

The estimated Equivalent Process Maturity Level (EPML) or

$$E = B + 0.01 \sum_{j=1}^{5} SF_j \qquad B = 0.91$$

Cost Drivers (Post-Architecture)

- Product Factors
  - Required Software Reliability (RELY)
  - Data Base Size (DATA)
  - Product Complexity (CPLX)
  - Developed for Reusability (RUSE)
  - Documentation Match to Life-Cycle Needs (DOCU)
- Platform Factors
  - Execution Time Constraint (TIME)
  - Main Storage Constraint (STOR)
  - Platform Volatility (PVOL)
- Personnel Factors
  - Analyst Capability (ACAP)
  - Programmer Capability (PCAP)
  - Personnel Continuity (PCON)
  - Applications Experience (APEX)
  - Platform Experience (PLEX)
  - Language and Tool Experience (LTEX)
- Project Factors
  - Use of Software Tools (TOOL)
  - Multisite Development (SITE)
- General Factor
  - Required Development Schedule (SCED)

Cost Drivers (Early-design)

- PERS (ACAP, PCAP, PCON)
- RCPX (RELY, DATA, CPLX, DOCU)
- RUSE (RUSE)
- PDIF (TIME, STOR, PVOL)
- PREX (APEX, PLEX, LTEX)
- FCIL (TOOL, SITE)
- SCED (SCED)

3. Executing

- Launch the project : kick off meeting
- Acquire and manage project team (internal and
- external resources)
- Acquire the required equipment and materials and external services
- Execute the plans (communication, change, quality)
- Perform the work identified in the WBS
- Perform controlling and monitoring activities

4. Monitoring and Controlling

Monitoring consists of collecting data about where the project stands, since projects never stick to the initial plans due to changes, problems, etc. Controlling is where you implement corrections to get your project back on track. Controlling increases risks factors. If schedule is important you can use techniques like fast-tracking and crashing If money is a priority you can reduce costs by reducing resources or overhead costs. If the schedule, money and resources are not negotiable reduce scope eliminating the tasks associated with it.

Earned Value Analysis

- Budget at completion (BAC): total budget for the project
- Planned value (PV): budgeted cost of work planned
- Earned value (EV): budgeted cost of work performed
- Actual cost (AC): actual cost for the completed work

Behind schedule if $EV < PV$. Over budget if $EV < AC$.

Schedule POV

Schedule Variance: $SV = EV - PV$
Schedule Performance Index = $SPI = EV/PV$

Cost POV

Cost Variance: $CV = EV - AC$
Cost Performance Index = $CPI = EV/AC$

Estimate at completion

Spending at the same rate: $EAC = BAC/CPI$
Continue to spend at original rate: $EAC = AC + (BAC - EV)$
Both CPI and SPI influence remaining work:
$EAC = [AC + (BAC - EV)]/(CPI \cdot SPI)$

Fast Tracking

Push tasks to occur faster than they would, by using negative lag time.

Crashing

Shorten the tasks on the critical path, usually increases costs.

5. Closing

- Ensure project acceptance
- Track project performance
- Lessons learned
- Close contracts
- Release resources

JEE

Framework used to facilitate the development of Enterprise Application (EA). Application based on JEE are portable everywhere there is Java EE. JEE applications are based on the use of *components* that communicate with each other. Components are deployed inside *containers*, which provide various services such as lifecycle management, lookup of other components and communication between components.

Multitiered architecture:

- Client tier
- Web tier
- Business tier
- Enterprise Information System (EIS) tier

Client Tier Containers

Applet container to manage the execution of applets (web browser + Java plugin).

Application client container to manage the execution of application client components.

Client Tier Components

Application client is a component that runs directly on the machine.

Applet is a component that is integrated in a web page and executed through a plugin.

Web clients is a client made of dynamic web pages generated by web components (only browser needed).

Web Tier Containers

Web container is the interface between web components and the web server. Manages components lifecycle, dispatches requests to application components, provides information about current request.

Web Tier Components

Servlets is a component that extends the capabilities of a base server by using the request-response paradigm. Commonly used for HTTP requests. Acts as a middleware.

JSP pages text-based documents that embed JSP (Java) elements, that are dynamically constructed (dynamic content).

JavaServer Pages based on servlets and JSP, provides user interface component framework fro web applications.

Business Tier Container

EJB container provides a run-time environment for enterprise beans within the application. Middleware between business logic within the beans and the rest of the application server. Maintains pools of beans to improve performance and is a midddleware between beans and the underlying EIS.

Business Tier Components

Enterprise Java Bean is a server-side component that encapsulates the business logic of an application. There are various types of beans:

- Session beans
  - Stateful session beans: one bean for each client, maintains state of the conversation.
  - Stateless session beans: does not maintain any kind of state, all instances are equivalent.
  - Singleton session beans: only one instance per application.
- Message-driven beans: act as a JMS (Java Messaging System) message listener. These beans execute upon receiving a new message, stateless by design, all instances are equivalent.

JNDI

Java Naming and Directory Interface enables components to locate other components and resources. Each resource is identified by a unique identifier (JNDI name).
Generic lookup:

```
DataSource ds = (DataSource) InitialContext
    .lookup("java:comp/DefaultDataSource");
```

Resource Injection

Another way to use JNDI names. Injects JNDI resources directly, resolved by resource name and it is not type safe.

```
public class MyServlet extends HttpServlet {
    @Resource(name="java:comp/DefaultDataSource")
    private javax.sql.DataSource dsc;
}
```

Dependency Injection

Java classes become managed objects. Higher decoupling. Injects regular classes, resolved by type, and it is type safe.

```
public class Billing {
    @inject CurrencyConverter cc;
}
```

Instance Pooling

JEE maintains a pool of ready to use beans, in order to have better performance. In case of stateless session beans or message-driven beans, every instance is the same (there is no state). In this case beans can only have two states: *ready* or *non-existent*.
Lifecycle:

1. Create bean.
2. Dependency Injection.
3. Invocation of `@PostConstruct` method (if exists). The bean is now ready.
4. When ready, the bean can accept various requests (or messages in case of message-driven beans).
5. At the end of lifecycle, invocation of `@PreDestroy` method (if exists).
6. Bean is garbage-collected.

In case of singleton beans, only one instance per application. If singleton is annotated with `@Startup`, it is instantiated upon application deployment.

Activation/Passivation

Used for stateful session beans. Each bean, if not used, is serialized and written to disk (Passivation). When a requests to that bean arrives, the bean is read from disk and recreated (Activation).
Lifecycle:

1. Create bean.
2. Dependency Injection.
3. Invocation of `@PostConstruct` method (if exists). The bean is now ready.
4. When ready, the bean can accept various requests.
5. For passivation, invocation to `@PrePassivate`.
6. For activation, invocation to `@PostActivate`.
7. At the end of lifecycle, invocation of `@Remove`.
8. Invocation of `@PreDestroy` method (if exists).
9. Bean is garbage-collected.

JPA

Java Persistence API is an interface to manage a DB (usually relational).

Entity

A table of the DB corresponds to a Java class. The attributes of the Java class are mapped to columns of the table.
Relationships are define through annotations.
Example:

```
@Entity
@Table(name="users")
public class User implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long Uid;

    @NotNull
    private String name;

    @Pattern(regexp="...")
    private String email;

    public User() {
        super();
    }

    // getter
    public Long getUid() {
        return Uid;
    }
}
```

EntityManager

Entities are managed by an `EntityManager`, associated to a *persistence context*.

- Managing an Entity Instance's Lifecycle
- Finding Entities Using the Entity Manager
- Persisting and Removing Entity Instances
- Query
- Synchronizing Entity Data to the Database

Example:

```
@Stateless
public class MyProductManager implements ↩
     ProdutManager {
    @PersistenceContext
    private EntityManager em;

    @Override
    public Product findProduct(int Productid) {
        return em.find(Product.class, Productid);
    }
}
```

Named query

Example:

```
@Entity
@NamedQueries({
    @NamedQuery(
        name="Product.FIND_BY_NAME",
        query="SELECT p FROM Product PT WHERE PT.↩
            name=:name"
    )
})

// inside a business logic bean
@PersistenceContext
private EntityManager em;

@Override
public List<Product> findByName(String name) {
    Query query = em.createNamedQuery(
        "Product.FIND_BY_NAME",
        Product.class
    );
    query.setParameter("name", name);
    List<Product> p = query.getResultList();
    ...
}
```

Relationship

Unidirectional One-to-one

```
@OneToOne
@JoinColumn(name="columnName")
private OtherEntity foo;
```

Bidirectional One-to-one

In the first entity entity:

```
@OneToOne
@JoinColumn(name="columnName")
private OtherEntity foo;
```

In the other entity:

```
@OneToOne(mappedBy=foo)
private FirstEntity bar;
```

Many-to-one

The primary key of the other entity is used as a foreign key.

```
@ManyToOne
private OtherEntity foo;
```

One-to-many

```
@OneToMany
private Collection<OtherEntity> foos;
```

Bidirectional one-to-many/many-to-one
In the first entity:

```
@ManyToOne
private OtherEntity foo;
```

In the other entity:

```
@OneToMany(mappedBy=foo)
private Collection<FirstEntity> bars;
```

Unidirectional many-to-many
Let's suppose to have two tables A and B, A having primary key Aid, B having primary key Bid. The join table is called JT and has two columns fkA and fkB, respectively the foreign key to Aid and Bid.
Inside entity A we have:

```
@ManyToMany
@JoinTable(
    name="JT",
    joinColumns = {
        @JoinColumn(
            name="fkA",
            referencedColumnName="Aid"
        )
    },
    inverseJoinColumns ={
        @JoinColumn(
            name="fkB",
            referencedColumnName="Bid"
        )
    }
)
private Collection<B> foo;
```

Bidirectional many-to-many
In entity A is the same as before, in entity B:

```
@ManyToMany(mappedBy="foo")
private Collection<A> bar;
```

Signatures, Fields, Paragraphs

Signatures

General

```
sig qualified-name ... {
  field declarations
}
// or ...
sig qualified-name ... {
  field declarations
}{
  signature facts
}
```

Given `sig S { ... } { F }`, *F* is interpreted as if the model read `sig S { ... } fact { all this : S | F' }`, where *F'* is like *F* but each name *f* is expanded to `this.`*f* if *f* names a field of *S*. Write `@`*f* to suppress the expansion.

Top-level type signatures

```
sig qname { ... }
```

Subtype signatures

```
sig qname extends superclass { ... }
```

N.B. If *A* and *B* each extend *C*, then *A* and *B* are disjoint.

Subset signatures

```
sig qname in sup { ... }
sig qname in sup1 + sup2 + ... { ... }
```

N.B. Subset signatures are *not* necessarily pairwise disjoint, and may have multiple parents.

Multiple signatures

```
sig qname1, qname2, ... { ... }
```

≡ `sig qname1 { ...} sig qname2 { ... } ...`

Paragraphs

Facts

```
fact name { formulas }
// name is optional:
fact name { formulas }
```

Predicates, Run

Predicates are either true or false; they can take arguments.

```
pred name { formulas }
pred name [decl, decl2 ...]{ formulas }
```

Use `run` to request an instance satisfying the predicate:

```
run name
```

Optionally specify *scope* (defaults to 3):

```
run name for 2
run name for 2 but 1 sig1, 5 sig2
```

The function `disj` is **predefined**; true iff its arguments are mutually disjoint.

Assertions, Check

```
assert name { formulas }
```

Unlike predicates, assertions don't bind arguments.

Use `check` to look for counter-examples:

```
check name for 2 but 1 sig1, 5 sig2
```

Functions

```
fun name [decl, ...] : e1 { e2 }
```

The body expression *e2* is evaluated to produce the function value; the bounding expression *e1* describes the set from which the result is drawn.

The function `sum` is **predefined**.

Declarations, Formulas, Expressions

Declarations

Fields of signatures, function arguments, predicate arguments, comprehension variables, quantified variables all use same declaration syntax:

Simple declaration

```
name : bounding-expression
```

Constrains values to be a subset of the value of the bounding expression.

Multiple declaration

```
name1, name2 : bounding-expression
// or
disj name1, name2 : bounding-expression
```

In field declarations, `disj` can also be on the right:

```
sig S {f : disj e}
```

Requires distinct *S* atoms to have distinct *f* values; ≡ `all a, b : S | a != b implies no a.f & b.f` ≡ `all disj a, b : S | disj [a.f, b.f]`

Multiplicities

Default multiplicity is one:

```
name1 : bounding-expression
// equivalent to:
name1 : one bounding-expression
```

Other multiplicities:

```
name2 : lone expr // at most one
name3 : some expr // one or more
name4 : set expr // zero or more
```

Relations

Bounding expression may denote a relation:

```
r : e1 -> e2
```

Multiplicities in declaring relations:

```
r : e1 -> one e2 // total function
r : e1 -> lone e2 // partial function
r : e1 one -> one e2 // 1:1 (bijection)
```

Formulas

Formulas (aka constraints) are boolean expressions. Primitive boolean operators include the comparison operators:

```
set1 in set2
set1 = set2
scalar = value
```

Expression quantifiers make booleans out of relational expressions.

```
some relation-name
no r1 & r2 // etc.
```

Quantified expressions are formulas:

```
some var : bounding-expr | expr
all var : bounding-expr | expr
one var : bounding-expr | expr
lone var : bounding-expr | expr
no var : bounding-expr | expr
```

True iff *expr* is true for some, all, exactly one, at most one, or no elements of the set denoted by *bounding-expr*

The logical operators (`not`, `and`, `or`, `implies`, `iff`) can form compound booleans; most of them apply *only* to boolean expressions.

```
boolean and boolean2
not boolean or boolean2
boolean implies boolean2 // etc.
```

Operators

Precedence

In precedence order.

a, *b*, *c* are *n*-ary relations ($n \neq 0$), *f* a functional relation, *r*, *r1*, *r2* are binary relations, *s* is a set (unary relation).

N.B. ≡ is standard mathematical syntax, not Alloy syntax.

Unary operators: `~r` (transpose / inverse), `^r` (positive transitive closure), `*r` (reflexive transitive closure)

Dot join: `a.b`

Box join: `b[a]` (also for function application, `f[t]`). N.B. dot binds tighter than box, so `a.b[c]` ≡ `(a.b)[c]`

Restriction: `s <: a` (domain restriction), `a :> s` (range restriction)

Arrow product: `a -> b` (Cartesian product)

Intersection: `a & b` (intersection*)

Override: `r1 ++ r2` (relational override)

Cardinality: `#a` (how many members in a?**)

Union, difference: `a + b` (union*), `a - b` (difference*)

Expression quantifiers, multiplicities: `no`, `some`, `lone`, `one`, `set`

Comparison negation: `not`, `!`

Comparison operators: `in`, `=`, `<`, `>`, `=`, `=<`, `=>`

Logical negation: `not`, `!`

Conjunction: `and`, `&&`

Implication: `implies`, `else`, `=>`

Bi-implication: `iff`, `<=>`

Disjunction: `or`, `||`

Let, quantification operators: `let`, `no`, `some`, `lone`, `one`, `sum`

* `a` and `b` must have matching arity

** Arithmetic overflow may occur.

Associativity:

Implication associates right: `p => q => r` ≡ `p => (q => r)`

`else` binds to the nearest possible `implies`: `p => q => r else s` ≡ `p => (q => r else s)`

All other binary operators associate left: `a.b.c` ≡ `(a.b).c`

Conditional expressions

```
boolean implies expression
boolean implies expr1 else expr2
```

Let expressions

```
let decl, decl2 ... | expression
let decl, decl2 ... { formulas }
```

Relational expressions

Constants: `none` (the empty set), `univ` (the universal set), `iden` (the identity function).

Compound expressions: *r1 op r2* where *op* is a **relational operator** (`->`, `.`, `[]`, `~`, `&`, `*`, `<:`, `:>`, `++`).

Integer expressions

Arithmetic operators (`plus`, `minus`, `mul`, `div`, `rem`) apply only to integer expressions. They name ternary relations, so *x* + 1 can be written as any of: `plus[x][1]`, `plus[x,1]`, `x.plus[1]`, or `1.(x.plus)`.

Miscellaneous

Module structure

```
// module declaration
module qualified/name

// imports
open other_module
open qual/name[Param] as Alias

// paragraphs (any order)
sig name ...
fact name { formulas }
pred name { formulas }
assert name { formulas }
fun name [Param] : bounding-expr {
  body-expression
}
run pred-name for scope
check assertion for scope
```

Lexical structure

Characters: any ASCII character except `\ ` $ % ?`

Alloy is **case-sensitive**

Tokenization: any whitespace or punctuation separates tokens, *except* that `=> >= =< -> <: :> ++ || // -- /* */` are single tokens (so: `!=` can be written `! =`)

Comments: from `//` to end of line; from `--` to end of line; `/*` to next `*/` (no nesting).

Identifiers (names): letters, numerals, underscore, quote marks (*no hyphens*)

Qualified names (qnames): sequence of slash-separated names, optionally beginning with `this` (e.g. `xyz`, `this/a/b/c`, `util/ordering`)

Numeric constant: [1-9][0-9]*

Reserved words: `abstract all and as assert but check disj else exactly extends fact for fun iden iff implies in Int let lone module no none not one open or pred run set sig some sum univ`

Namespaces: 1 module names and aliases; 2 signatures, fields, paragraphs (facts, predicates, assertions, functions), bound variables; 3 command names. Names in different namespaces do not conflict; variables are lexically scoped (inner bindings shadow outer). Otherwise, no two things can share a name.
