M.Sc Computer Science

Object-Oriented Design and Programming

Session 1

Oded Lachish

Room: (FT session) Malet 404 13:30-17:00, (PT Session) Malet 407 18:00-21:00
Visiting Hours: Tuesday 17:00 to 19:00
Email: oded@dcs.bbk.ac.uk
Web Page: http://www.dcs.bbk.ac.uk/~oded/OODP12/OODP2012.html

Resources

Textbooks:

- "Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design and Iterative Development", Craig Larman, 3rd edition, Prentice-Hall 2002
- "Design Patterns. Elements of Reusable Object-Oriented Software." E. Gamma, R. Helm, R. Johnson, J. Vlissides.

Lecture Notes:

- http://www.dcs.bbk.ac.uk/~oded

OO Design and Programming

Main objectives:

- An introduction to software engineering.

What will we see:

- Object-oriented analysis and design (OODP) methodologies.
- UML for illustrating analysis and design models.
- Design Patterns
- Refactoring

Congratulations

- New Job
- You are now responsible for developing a new software product *from scratch*

Great,

what next?

I know C++, Java, UML?

How do you develop software?

Me?

I write the code,

all of it

compile

AND IT WORKS

EVERY SINGLE TIME

(that is why I teach)

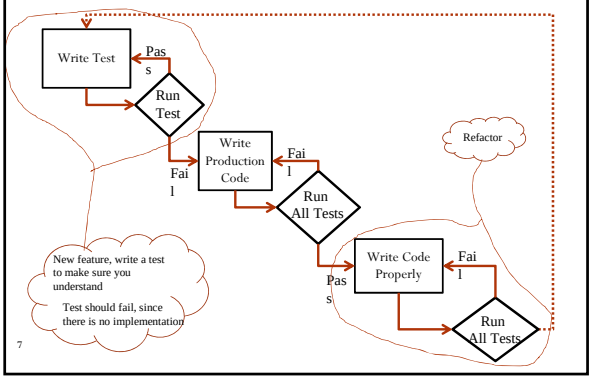
You?

Test Driven Development (TDD)

Philosophy

- Untested code is "not code"
- Short development cycle
- Testing is understanding

TDD Cycle



TDD benefits

- Less Debugging (debugging is tedious and expensive)
- Small steps – easier to backtrack (less debugging)
- Small pieces drive developers towards **modularity**
- Bugs caught earlier
- Testing is understanding

TDD vulnerabilities

- Developer writing tests for his own code
- Checking private variables may require dedicated “hacks” that may remain in the code.
- More code
- Longer time to write
- Over reliance on unit testing

TDD implementation

Wish List

- Easy to upgrade code (refactor)
- Dedicated testing tools
- Friendly version control

So I “Know” TDD Can I Start

Examples of software products

- *Real time:* air traffic control
- *Embedded systems:* digital camera, GPS
- *Data processing:* telephone billing, pensions
- *Information systems:* web sites, digital libraries
- *Sensors:* weather data
- *System software:* operating systems, compilers
- *Communications:* routers, mobile telephones
- *Offices:* word processing, video-conferences
- *Scientific:* simulations, weather forecasting
- *Graphical:* film making, design
- etc.

Putting software production in perspective

- Biggest program that you have written?
- Biggest program that you have worked on?
- Biggest project team that you have been part of?
- Longest project that you have worked on?
- Most people who have used your work?
- Longest that your project has been in production?
- The coursework for OODP should be about 0.01 person·years.
- A big project may be 100 to 1,000+ person·years.
- Software production requires team work.
- Large software projects are built on older ones.
- Software is expensive.

What makes good software?

- Relevance
- Usability
- Robustness
- Fault-tolerance
- Maintainability
- Efficiency
- …

The emphasis on these metrics varies across projects.

So?

- The number of possible tasks seems unbounded
- The scale of some tasks is immense
- There can be many different quality constraints

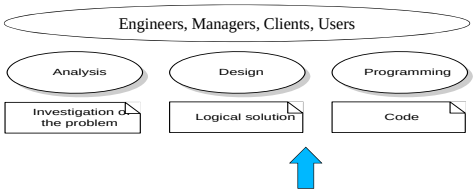
A systematic approach is required!

Software Engineering

What is software engineering?

Software engineering is not just coding

- Knowing an OO language, like Java or C++, is necessary, but not sufficient in order to create object software.
- In between a nice idea and a working software, there is much more than programming.



Design Patterns
Verification

What Next?

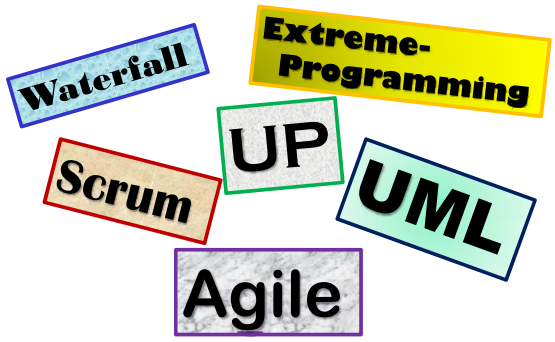
Is there a magic book that will ensure success?

Regretfully probably no.

A large variety of entities some
with massive resources have

failed miserably!

So ***BUZZWORDS***



Start from the beginning (very partial history)

- In the beginning there was **CHAOS**
  - Software is written on the fly

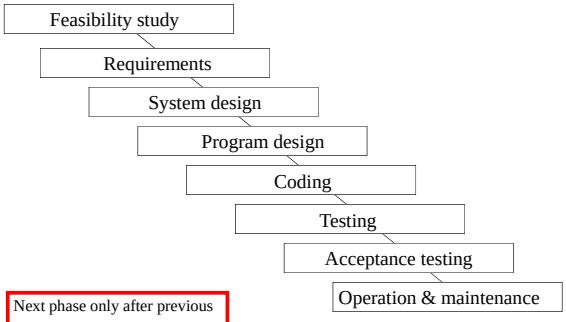
Systems Grow

- Bugs → verification takes forever
- Maintenance virtually impossible

Engineering Methodologies

- Plan before building
- Predictable, Efficient

The (classical) waterfall model



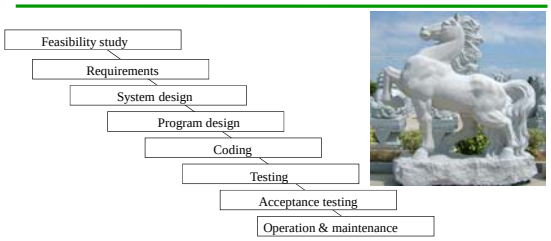
Next phase only after previous is 100% percent done!

Why waterfall model?

- Emphasis on documentation
  - knowledge not lost,
  - less dependence on team members
- Predict problems and deal with them as early as possible
- Simple structured approach

- (Like in other engineering areas such as construction)

The (classical) waterfall model



What are the problems with the waterfall model?

So?

Accept that

- Requirements may change
- System design problems may arise during program design
- Program design problems may arise when coding
- …

Need a design methodology that can deal with these problems.

Agile Software Development

A group of software development methodologies

Focus: **THE BIG PICTURE**

"Everything is personal"

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tool
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Principles behind the Agile Manifesto

Working software is the primary measure of progress.
Agile processes promote sustainable development.

The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

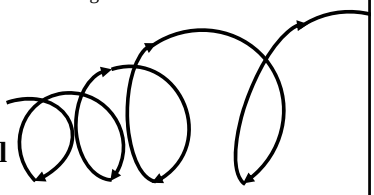
At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

The big conceptual changes

Software development methodologies that are

1. Iterative

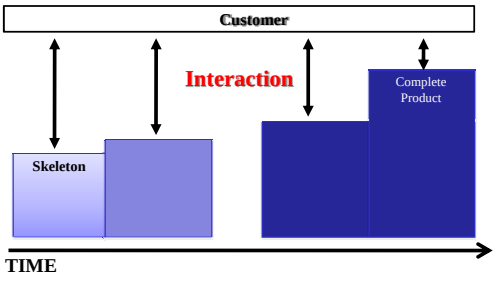
2. Incremental



3. Interactive

(Treat software development as a learning process)

Iterative, Incremental, Interactive



Agile Software Development Processes

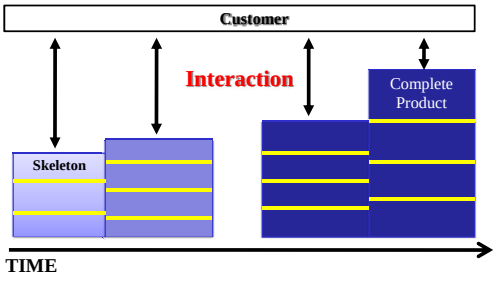
- Extreme Programming
  - Test-driven development
  - Continuous integration
- Scrum
  - “war room” – common project room
  - Daily meetings

What do we want?

A software development FRAMEWORK that deals with

- Analysis and design
- Object-oriented technology
- Well-defined
- Clear documentation and sharing of information during software development
- Re-usability
- Verification
- Testing

Both vertical and horizontal partition



UP – Unified Process

Unified Process (UP)

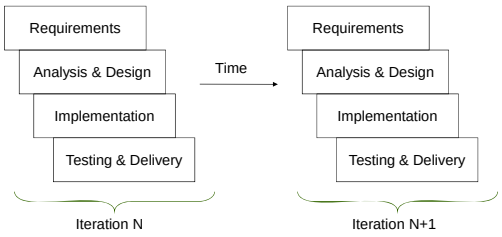
- The Unified Process is a process framework particularly suited for building OO systems.
- Rational Unified Process (RUP) is a detailed refinement of UP
- Why use the Unified Process to develop software?

Why UP?

- Iterative and evolutionary (incremental) development contrasted with a sequential or “waterfall” lifecycle.
- Early programming and testing of a partial system, in repeating cycles.
- It assumes development starts before all the requirements are defined in detail.
- Feedback is used to clarify and adapt the system to evolving specifications

The UP iterative development

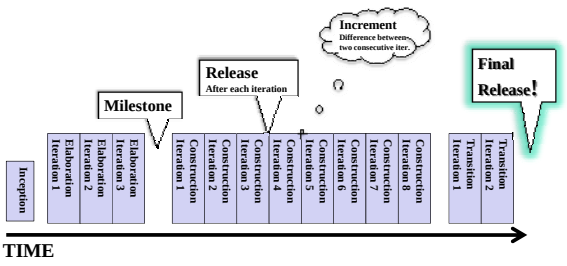
- Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1
- Iterations are fixed in length - Outcome is a tested, integrated and executable partial system.



UP phases

- Inception
  - Feasibility phase and **approximate** vision
- Elaboration
  - Core architecture implementation, high risk resolution
- Construction
  - Implementation of remaining elements
- Transition
  - Beta tests, deployment

UP phases



UP artifacts and disciplines

- The UP describes work activities, which result in *work products* called *artifacts*. Examples of artifacts:
  - Code
  - Web graphics
  - Database schema
  - Use cases (text documents explaining business scenarios)
  - Diagrams, models and so on
- A *discipline* (or workflow) is a set of work activities. Examples:
  - Requirements
  - Business modeling
  - Design

Inception

Inception

(Usually, lasts one or two weeks (e.g. one iteration))

- Define the vision and business scope of the project
  - Define high-level goals and constraints
  - Investigate core requirements
- Study feasibility
  - Identify critical (business, technical, schedule, …) risks
  - Obtain an order-of-magnitude estimate of the cost
  - Determine who will do it (buy and/or build)
- Decide if it is worthwhile to invest in deeper exploration
  - Plan the first iteration of elaboration
  - Estimate the number of iterations needed for elaboration

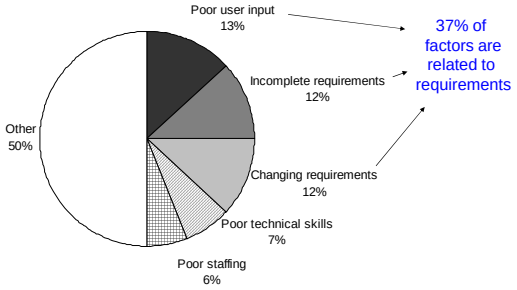
Artifacts that may be started during inception:

Artifact	Description
Vision and Business Case	**High level goals and constraints**
Use-Case Model	**Functional requirements**
Supplementary Specifications	**Non functional requirement**
Glossary	**Key domain terminology, and data dictionary**
Risk List and Risk Management Plan	Risks and their mitigations
Prototypes and Proof-of-Concepts	Validating technical ideas
Iteration Plan	Description of first iteration
Phase Plan & Software Development Plan	Guess for elaboration duration. Tools etc.
Development Case	Description of the Customized UP

Requirements

Importance of requirements specification

- Factors on challenged software projects



- The UP embraces changes in requirements and suggests their skilful specification through use-case writing.

Requirements

- Requirements are capabilities and conditions to which the system – and more broadly, the project - must conform. Either derived directly from user needs, or stated in a contract, standard, spec, formally imposed document.
- The FURPS+ model [Grady92]:
  - **F**unctional
  - **U**sability
  - **R**eliability
  - **P**erformance
  - **S**upportability
  - **+**: {implementation, interface, operations, packaging, legal}

quality requirements

non-functional requirements

+

- **Implementation**
  - Required standards, implementation languages, resource limits.
- **Interface**
  - Specifies external items with which the system must interact
- **Physical**
  - Constraints the hardware used to house the system, shape, size or weight for example.
- **Legal**
  - Licensing etc

Examples

- The project will be localised (support multiple human languages)
- The persistence will be handled by a relational database.
- The database will be Oracle 9i.
- The system will run 24x7
- All presentation logic will be written in Visual Basic.

Vision, Supplementary Specification and Glossary

Vision, Supp. Spec. and Glossary

- The *Vision* includes the big ideas regarding why the project was proposed, what the problems are, who the stakeholders are, what they need, and what the proposed solution looks like.
- The *Supplementary Specification* captures non-functional requirements, e.g. documentation, packaging, supportability, licensing etc.
- The *Glossary* is like a data dictionary; it defines terms used in other artifacts (use cases, SS, vision etc.).

  Suggestion: start the glossary early!

A (partial) Vision example

Revision History: (version, date, description, author)
Introduction: We envision a fault-tolerant point-of-sale appl. With the flexibility to support varying business rules … and integration with multiple third-party supporting systems.
Positioning:
Business Opportunity: (what existing systems can't do)
Problem Statement: (problems caused by lack of features)
Product Position Statement: (who the system is for, outstanding features, how is it different from competition)
Stakeholder Descriptions:
Stakeholder (non-user) summary: **Stakeholder goals:**
User summary (and their goals): **User-level goals:**
Product Overview: …
Product perspective: **Product benefits:**
Cost and Pricing: **Licensing and Installation:**
Summary of System Features:
Other Requirements and Constraints:

A (partial) SS example

Revision History: (version, date, description, author)
Introduction: This document is the repository of all requirements not captured in the use cases.
Functionality:
Logging and Error Handling: Log all errors to pers. storage
Pluggable Business Rules: Customize functionality…
Security: All usage requires user authentication
Usability: …
Reliability: …
Performance: …
Supportability: (Adaptability: and Configurability:) …
Implementation Constraints: Java technology solution
Purchased Components: Tax calculator.
Interfaces: (Hardware: and Software:)
Domain (Business) Rules: (id, rule, changeability, source)
Legal Issues:
Information in Domains of Interest: (pricing, sales tax, item ids)

Use-Cases

Description of Functionality

Problem

- The Clients are not engineers
- Throwing UML diagrams at them, is at best confusing
- Good communications with them is **critical**!

HOW?

The use-case model [Jacobson92]

- The *use-case model* – the set of all use cases - is an artifact within the *requirements* discipline of the Unified Process.
- *Use cases* are stories of using a system to meet goals.
- Use case example (in brief format):

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt and leaves with items.

Use-cases what for and why?

- What for?
  - To describe how the system will behave
- Why?
  - Simple, easy to communicate
  - Replace the old feature list, with something that relates to the "structure" of the problem

A good way of understanding and describing requirements (esp. functional ones).

Guidelines for writing use cases

- Try to answer the question: *How can using the system provide observable value to the users or fulfil their goals?*
- Don't think of system requirements in terms of a check list of features or functions.
- Emphasize the functional requirements (other requirement types are sometimes included, but have a secondary role).
- Text, not diagrams.
- *Black-box* use cases are highly recommended. We describe the responsibilities of a system, not the internal workings.

Process Sale: … The system records the sale …

NOT: The system generates an SQL INSERT stmt for the sale…

Use case formats

- **Brief:**

  one paragraph describing the main success scenario

- **Casual:**

  multiple paragraphs that cover various scenarios

Previously

- **Fully-dressed:**

  Coleman's Use Case Template

Next

Already have brief and casual use cases

- An *actor* is something with behaviour, such as a person, computer system, or organization; e.g. a cashier.
- A *scenario* – also called *use case instance* - is a specific sequence of actions and interactions between actors and the system under discussion. It is one path through the use case.
- A *use case* is a collection of related success and failure scenarios that describe actors using a system to support a goal.

Process Sale

Main Success Scenario: A customer arrives at a checkout with items to purchase …

Alternate Scenarios:

If the system rejects customer's credit card…

If the customer has less cash than the required payment…

Actors?

Actor – anything with a behaviour (including the system under discussion if it requires other systems)

- Primary actor – has goals fulfilled by SuD.
- Supporting Actor – service provider, information.
- Offstage actor – anyone with an interest in the SuD that is not one of the above, HMRC,...

Coleman's Use Case template

Fields	Description
Use Case	Use case identifier and reference number and modification history.
Description	Goal to be achieved by use case and sources for requirement.
Actors	List of actors (primary, secondary) involved in use case.
Assumptions	Conditions that must be true for use case to terminate successfully.
Steps	Interactions between actors and system that are necessary to achieve goal.
Variations (optional)	Any variations in the steps of a use case.
Non-Functional (optional)	List of non-functional requirements that the use case must meet.
Issues	List of issues that remain to be resolved

Steps

- A step completes when all its interactions have completed.
- Each step is completed before the next is started.

<sequence number><interaction>

1. Interaction a
2. interaction b
3. IF test THEN 3.1 interaction a
   3.2 interaction
   ELSE 3.3 interaction b
4. REPEAT 4.1 interaction a
   4.2 interaction b
   UNTIL no more

Steps

- Concurrency

1. IN PARALLEL interaction a||interaction b||interaction c
2. Interaction d

1. IN PARALLEL 1.1 interaction a||1.2 interaction b||1.3 interaction c
2. Interaction d

Variations

1. Buyer calls in with purchase request

1. Buyer may phone in , **or** fax in, **or** use web order form.

Non functional requirements

<keyword>:<requirement>

Performance, Reliability, Fault Tolerance, Frequency, Priority

Example 1

Use Case	**2.** Repairing _Cellular_Network **history** created 1/5/98 Derek Coleman, modified 5/5/98.
Description	Operator rectified a report by changing parameters of a cell. **sources** [Operating Manual 1993],[Jones 1998]
Assumptions	Changes to network are always successful when applied to a network.
Actors	Operator (primary) Cellular network Field maintenance engineer (secondary)

Steps	1. Operator notified of network problem
	2. Operator starts repair session
	3. REPEAT
	  1. Operator runs network diagnosis application
	  2. Operator identified cells to be changed and their new parameter values.
	  3. IN PARALLEL
	  1. Maintenance engineer tests network cells ||
	  2. Maintenance Engineer sends fault reports.
	UNTIL no more reports of problems.
	4. Operator closes repair session

Example 1 (cont)

Variations	**#1** System may detect fault and notifiy operator **or** Field maintenance engineer may report fault to Operator.
Non-Functional	**Performance Mean**: time to repair network fault must be less than 3 hours. **Fault tolerance:** A repair session must be able to tolerate fialure of Operator's console.
Issues	What are the modes of communication between field maintenance engineer and operator?

Uses Relationship

- In UML, commonalties between use cases are expressed with the **uses relationship.**
- The relationship means that the sequence of behaviour described in a *used (or sub) use case is included in the sequence of the using* use case.
- Using a use case is thus analogous to the notion of calling a subroutine.
- Sub use cases are full use cases in their own right, and therefore can be expressed using the use case template.

Uses example

Using a sub- use case in a step is expressed by a keyword such as PERFORM (or USING). For example, if Tune_Cell were a use case it could be used by the following interaction:

3. Operator PERFORMS 2.1 Tune_Cell

In allocating reference numbers to use cases it maybe convenient to use a Dewey Decimal numbering scheme to convey the using hierarchy. Thus if use case W, with reference number n, uses three sub use cases X, Y and Z, then X may be numbered n.1, Y n.2 etc.

Extending a Use-Case

Use Case Extension	<extension identifier> **extends** <use case identifier>
Description/Change	Goal to be achieved by extension
Steps	Changes to use case steps
Variations *(optional)*	…
Non-functional *(optional)*	…
Issues	…

Use Case Extension	<Repair_may_fail> **extends** <2.Repairing_Cellular_Networks>
Description/Change	Deals with assumption that network changes can never fail.
Steps	#3.3 **if** changes to network fail **then** the network is rolled back to its previous state
Issues	How are failures detected? Are roll backs automatic or is Operator intervention required?

Example 2

Use Case	**1.Process_Sale**
Descirption	A user arrives at a POS sales point and tries to purchase an item.
Actors	Cashier (Primary)
Assumptions	Cashier is identified and authenticated

Steps	1. Customer arrives at POS checkout with goods to purchase.
	2. Cashier starts a new sale.
	3. REPEAT
	  1. Cashier enters item identifier.
	  2. System records sale line item and presents item description, price and running total.
	UNTIL EMPTY_BASKET
	4. System presents total with taxes calculated.
	5. Cashier tells Customer the total, and asks for payment.
	6. Customer pays and System handles payment.
	7. System updates Accounting and Inventory systems.
	8. System presents receipt.
	9. Customer leaves with receipt and goods (if any).

Extensions?

- Can you think of useful extensions (alternative flows) that are not covered by the use case?

Extensions

*α. At any time, System fails:

Ensure all transaction states are recoverable.

1. Cashier restarts the system, logs in, and requests rollback.

2. System rollback.

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important.

1. Cashier enters item category identifier and the quantity.

3-6a. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

What do we do
with all the use cases?

Actor-goal lists

Actor	Goal	Actor	Goal
Cashier	process sales handle returns cash in cash out …	System Administrator	add users modify users delete users …
Manager	start up shut down …	Sales Activity System	analyze sales …
…	…	…	…

What happened in inception?

- A short requirements workshop
- Most actors, goals and use cases named
- Most use cases written in brief format; 10-20% of cases written in fully dressed format
- Most influential and risky quality requirements identified
- First version of the Vision and SS documents
- User interface-oriented prototypes to clarify the vision of functional requirements
- Recommendations on what components to buy/build/reuse
- Plan for the first iteration of elaboration
- …

Summary

- The Use-Case Model is an artifact of the Requirements discipline that includes:
  - writing use cases and designing use case diagrams (during Inception and Elaboration)
  - designing system sequence diagrams (in Elaboration)
- Vision, SS and Glossary are other artifacts of the Requirements discipline, which start in Inception and are further refined in Elaboration.
- 10-20% of requirements are defined in detail by the end of the Inception phase.
- Almost all requirements are defined in detail by the end of Elaboration.