מודלים לפיתוח מערכות תוכנה
Software Systems Modeling

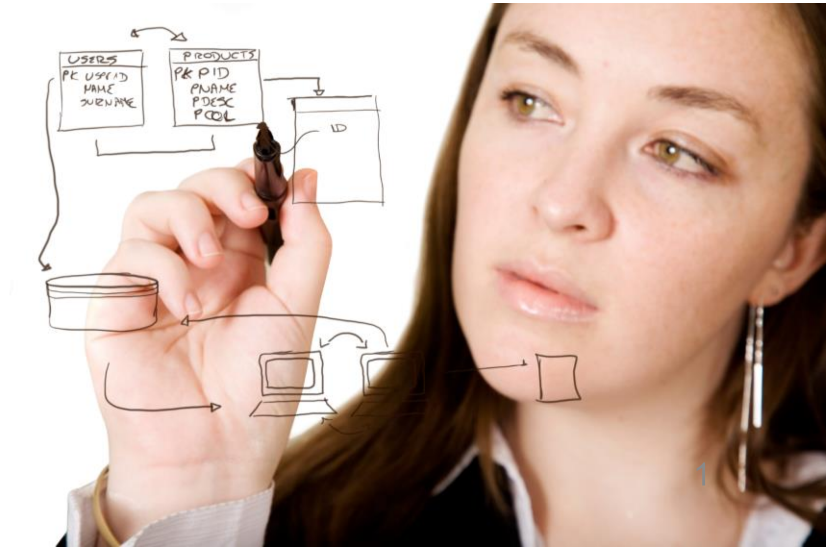
קורס 12003

סמסטר ב' תשע"ז

1. מבוא

ד"ר ראובן יגל

robi@post.jce.ac.il



השבוע

- על הקורס
  - לוגיסטיקה – סילבוס
- מבוא לתיכון ומודלים
- בקרת תצורה
  - תרגיל 1 – git
  - פירוט בפעם הבאה
- לקראת המשך, מבוא ל- UML

המרצה

אתם?

- רקע וניסיון
- במודלים?
- מה מעניין אתכם במודלים לתוכנה?

הקורס

הקורס

- עדיין בבניה!
- מבוסס מקורות שונים
- כולל למידה עצמית והתנסות
- מוזמנים להיות שותפים!
- מאגר הקורס וסילבוס

– https://github.com/jce-il/sw-modeling-2017b

תכנית (נתון לשינויים)

- Intro
- Version Control / Git – Practical & Modeling
- UML
  - OCL
- Modeling Tools
- SW Design & Architecture
- SW Process Modeling
- SW V&V (Testing) Modeling
- Project – modeling in OSS

מקורות להיום

- Design & Motivation
  - Edinburgh: Software Design Methods and Processes , A. Ireland
- Modeling
  - RIT Class, Wei Le
- Practiacl Git
  - Gitimmersion.com, Jim Weirich

מבוא למודלים

- Are Models Useful?
- From Coursera: Model Thinking https://www.coursera.org/course/modelthinking
  - ~~One to many and many to one 1:55m supertanker~~
  - Intelligent Citizen
- Designing vs Modeling?
- Edinburgh, RIT

- Chris Ganger, Coding is not the new literacy
- “Modeling is the new literacy”

Outline

- Motivations and challenges
- Process
- Strategies
- Quality
- ~~Roadmap~~

The Nature of Software

- Software lies at the nerve centre of most engineered artifacts and business processes, *i.e. from consumer electronics to financial modelling, and from automotives to medical applications*
- A single defect in millions of lines of code can result in a system failure (safety critical systems are and exception)
- Typically 50% of project costs are allocated to software design, of which 50% are spent on testing

The Nature of Software

- Software is among the most **complex** of engineered artifacts
- Software is flexible, so is expected to **conform** to standards imposed by other components, *e.g. hardware, external agents etc*
- Flexibility also increases the rate at which software is **changed** during its lifetime
- The **invisibility** of software makes it harder to contextualize compared to other engineering sectors, *e.g. construction industry*

Brooks 1987

The Economic Motive

"… the national annual cost estimates of an inadequate infrastructure for software testing are estimated to be $59.5 billion."

Federal Study, US Dept of Commerce, May 2002

"Worse - and spreading the effect of software flaws far beyond the original customer – several devastating computer viruses have taken advantage of bugs and defects in common operating systems ..."

CNET Networks Inc, Aug 2002

The Economic Motive

- US Internal Revenue Service - a failed $4-billion modernization effort in 1997, followed by an equally troubled $8-billion update.
- FBI - $170-million virtual case-file management system was terminated in 2005.
- Moody's Corp: financial research & analysis credit-worthiness ratings:

"Moody's awarded incorrect triple-A ratings to billions of dollars worth of a type of complex debt product due to a bug in its computer models"

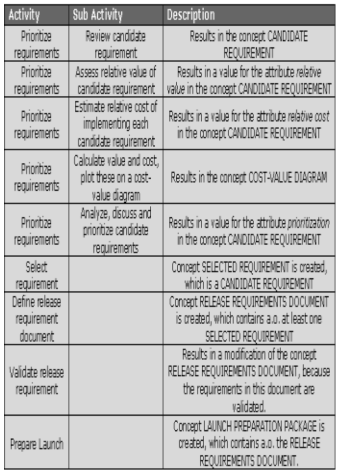
Financial Times 2008

The Security Motive

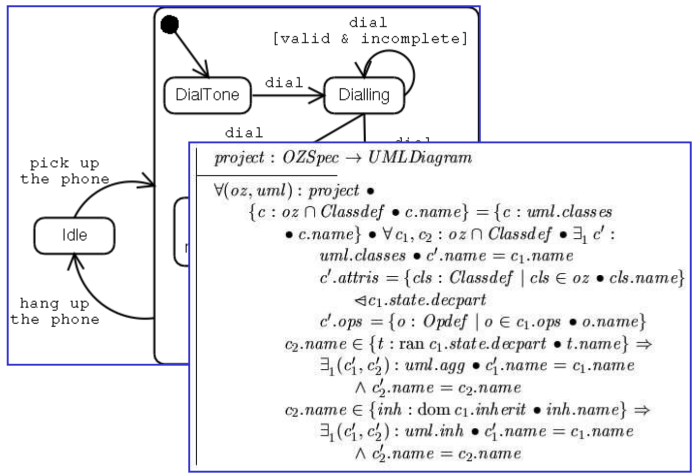
- The recent UK defence spending review identified Cyber Security as a priority area, giving rise to the Office of Cyber Security
- US Government formed a new branch of the military: the US Cyber Command (May 2010)
- Malware that can bring down real-world infrastructure is a clear and present danger, *e.g.*

  Stuxnet malware, which targets Siemens pump controller software, was responsible for damaging centrifuges within Iran's nuclear programme in 2010

The Role of Design



Activity	Sub Activity	Description
Prioritize requirements	Review candidate requirement	Results in the concept CANDIDATE REQUIREMENT
Prioritize requirements	Assess relative value of candidate requirement	Results in a value for the attribute *relative value* in the concept CANDIDATE REQUIREMENT
Prioritize requirements	Estimate relative cost of implementing each candidate requirement	Results in a value for the attribute *relative cost* in the concept CANDIDATE REQUIREMENT
Prioritize requirements	Calculate value and cost, plot these on a cost-value diagram	Results in the concept COST-VALUE DIAGRAM
Prioritize requirements	Analyze, discuss and prioritize candidate requirements	Results in a value for the attribute *prioritization* in the concept CANDIDATE REQUIREMENT
Select requirement		Concept SELECTED REQUIREMENT is created, which is a CANDIDATE REQUIREMENT
Define release requirement document		Concept RELEASE REQUIREMENTS DOCUMENT is created, which contains a.o. at least one SELECTED REQUIREMENT
Validate release requirement		Results in a modification of the concept RELEASE REQUIREMENTS DOCUMENT, because the requirements in this document are validated.
Prepare Launch		Concept LAUNCH PREPARATION PACKAGE is created, which contains a.o. the RELEASE REQUIREMENTS DOCUMENT.



Requirements

Design Models

Code

Design should play a pivotal role:

- *Clarify and refine requirements*
- *Early defect detection and elimination*

The Nature of Design

- A creative process involving:
  - Multiple perspectives (models)
  - Multiple layers of abstraction (models)
- An evolutionary process involving:
  - Incremental developments
  - Backtracking over designs
  - Requirements reformulation, elaboration and volatility

Process

- Architectural design: deciding on the subsystems and their relationships
- Subsystem design: provide an abstract specification for each subsystem
- Interface design: define the interface for each subsystem
- Component design: decomposition of subsystems into components
- Data structure design: data structuring decisions
- Algorithm design: algorithmic decisions

Strategies

- Two broad strategies for tackling software design
- Function-oriented design:
  - Software is structured around a centralized system state
  - System state is shared between a collection of ***functions*** (subroutines)
- Object-oriented design:
  - Software is structured around a collection of ***objects***, where each object is responsible for it own state
  - Object organized into a class hierarchy, exploiting inheritance

Quality

- Cohesion:
  - A measure of how well the parts of a component fit together, *i.e. how functionally related the parts are*
  - For example, strong cohesion exists when all parts of a component contribute different aspects of related functions
  - Strong cohesion promotes understanding and reasoning, and thus provides dividends with respect to maintenance and reuse via **separation of concerns**
  - Cohesion provides a measure as to how self-contained an object class is – however, inheritance reduces cohesion

Quality

- Coupling:
  - A measure of how strongly components are interconnected
  - Tightly coupled components share data (common coupling) or exchange control information (control coupling)
  - Loose coupling is achieved by not having shared data, or at least restricting access, *e.g. data communicated by parameters*
  - Loose coupling promotes **separation of concerns**
  - Object-oriented design promotes loose coupling, however, inheritance increases coupling, *i.e. a class is coupled with its super-class*

Quality

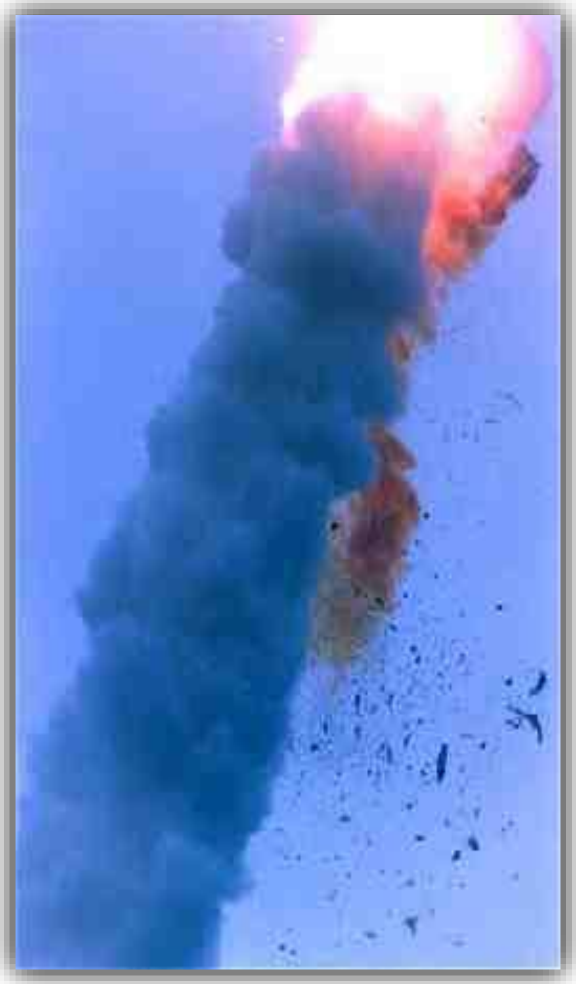
- Understandability:
  - Understandability of a design is very important for maintenance and change
  - Cohesion, coupling and complexity impact on the understandability of a design
- Adaptability:
  - Understandability, strong cohesion and loose coupling enhance the adaptability of a design
  - Traceability is also an important ingredient, *i.e. traceability between design representations as well as between requirements, design and code*

The Cost of Failure

- Consider converting 64-bits of data into 16-bits:

Arithmetic Overflow Error

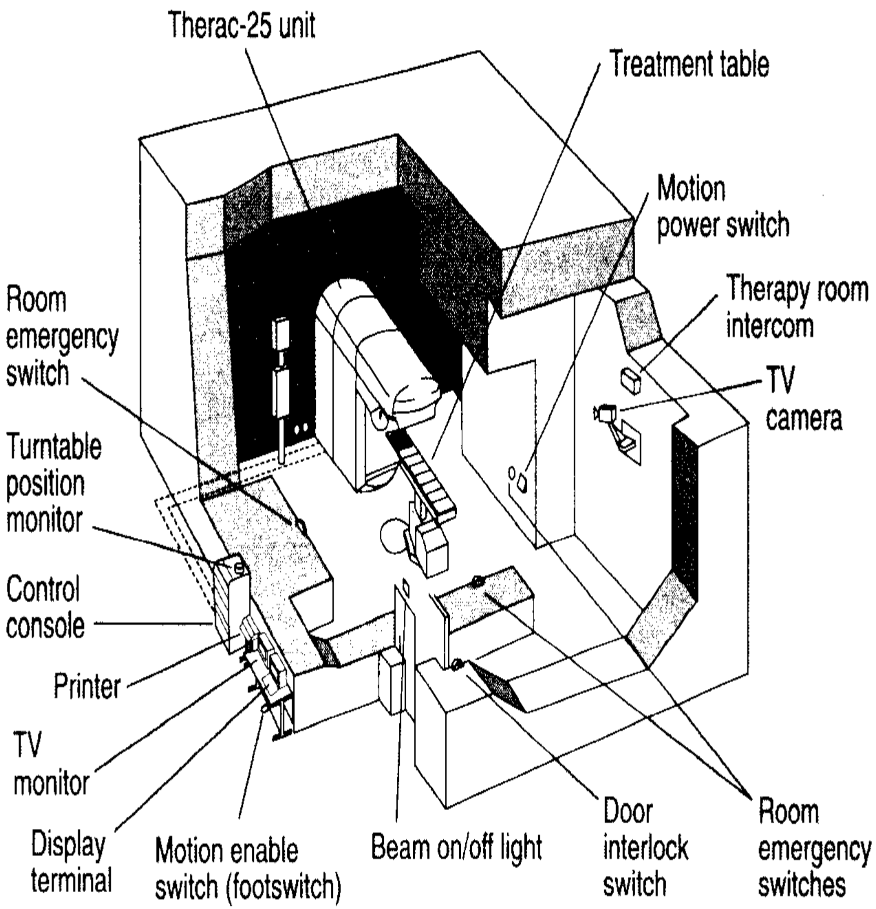
The Cost of Failure



Ariane 5

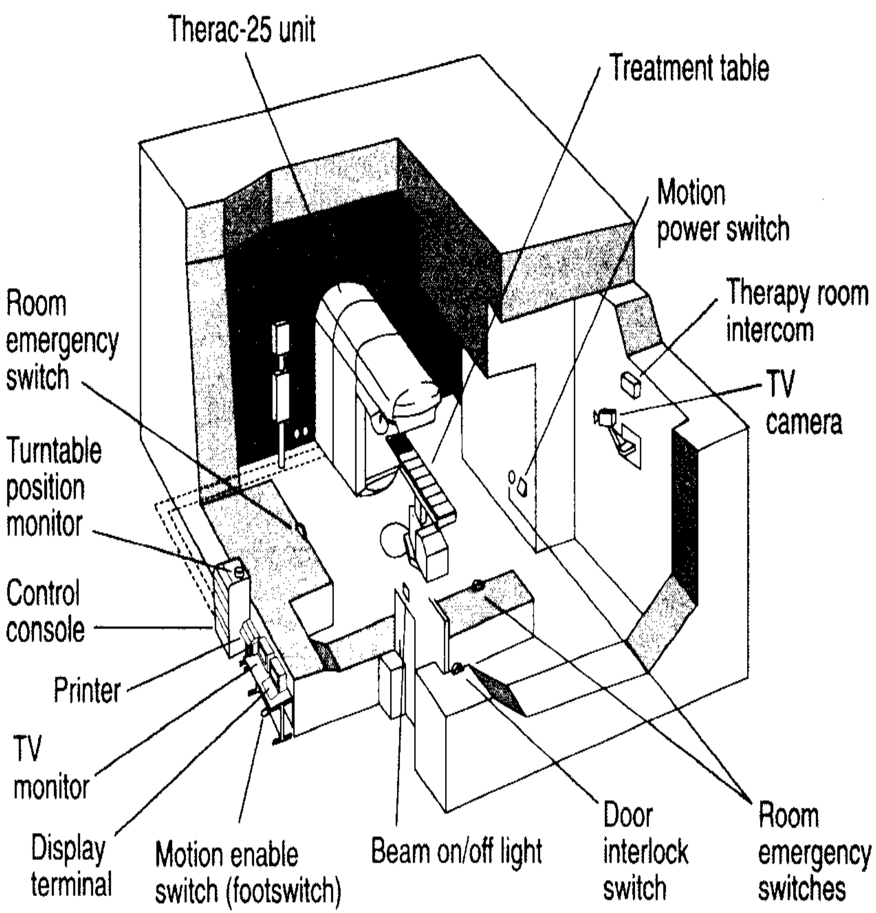
- Developed by European Space Agency
- Unmanned rocket with a cargo of scientific satellites ($500 million)
- In 1996, just 39 seconds into its maiden flight an overflow error occurred resulting the Ariane 5 control software initiating a *self-destruction operation!*

The Cost of Failure



Therac-25: a computer-controlled radiation therapy machine, build by Atomic Energy of Canada Ltd (AECL) used in US and Canadian hospitals & clinics during the 1980's. The Therac-25 was the successor to the Therac-6 and Therac-20 models. Unlike its predecessors the Therac-25 relied more on software control mechanisms

The Cost of Failure



- Therac-25 delivers two kinds of electron beams: **low energy** and **high energy**.
- A raw high energy beam is dangerous to living tissue so magnets are used to spread the beam energy so as to produce a safe therapeutic concentration.

The Cost of Failure

- Among the parameters a Therac-25 operator was able to set are the beam energy levels & beam modes. The latter effects the setting of the magnets.
- Operators have two ways of setting the system parameters:
  - data entry procedure
  - screen based editing
- A problem arose when the values established via the data entry procedure are edited during the magnet set-up phase, *i.e. screen display did not reflect actual settings.*

The Cost of Failure

- This problem resulted in high-powered electron beams striking patients with 100 times (approx) the intended dose of radiation
- Several patients showed the symptoms of radiation poisoning, 3 patients died later from radiation poisoning
- Aside: Therac-25 (March 1983) excluded the possibility of software defects since ***extensive testing had been undertaken!***

Course Road-Map

- Architectural design
- Function-oriented design
- Object-oriented design
- Component-based design
- Verification and Validation
- Dynamic Analysis (CS)
- Unit Testing & JUnit (CS)
- Static Analysis (CS)

Summary

- Learning outcomes:
  - Motivations for software design
  - The nature of software design – *process, strategies and quality*
- Recommended reading:
  - D. Budgen, “Software Design”, Addison-Wesley 2003
  - I. Sommerville, “Software Engineering”, Addison-Wesley 2007
  - F.P. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering”, IEEE Computer, 1987

Overview

What is a model?

Why software modeling?

What to model?

How to obtain a model?

Why modeling?

- Modeling is a tool for design, verification and testing
- Modeling and simulation
- Not only software, but any systems
- address more challenging problems, such as parallel computing and distributed systems.

Why Software Modeling?

- Schedule and divide tasks
- Collaboration and communication (contract)
- Decomposing complexity for coding
- Checking for software (correctness, security)
- Refactoring code
- Reuse and automatic coding

……

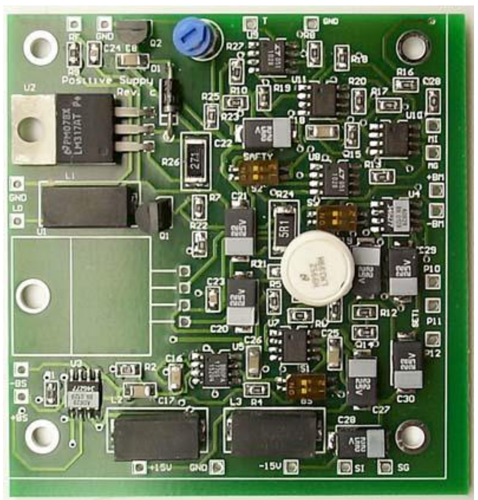
Other Questions

- Is Software modeling in real use?
- What about process development modeling?
- Modeling in Agile?

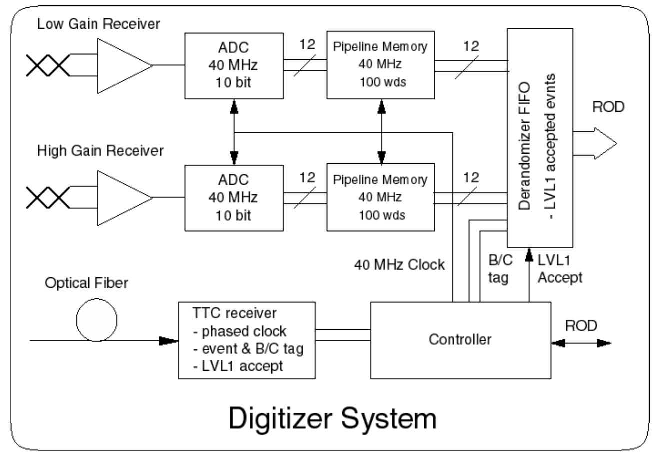
What is a model?

- Engineering model: abstraction

  A <u>reduced representation</u> of some system that highlights the properties of interest <u>from a given viewpoint</u>



system

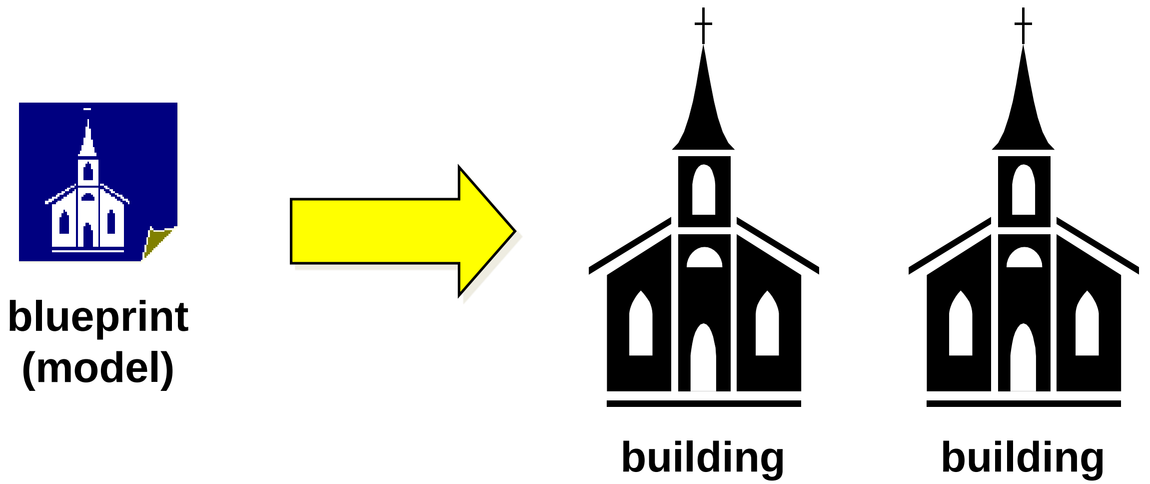


Functional Model

- We don't see everything at once
- We use a representation (notation) that is easily understood for the purpose on hand

Intuitive Understanding

- A *model* is a description of something
  - *"a pattern for something to be made"* (Merriam-Webster)



- model ≠ thing that is modeled
  - The Map is Not The Territory

Levels of Abstraction and Reasons

- Business model
- Requirement
- Design and Algorithm
- Architecture
- Code

- Tracibility

Modeling Maturity Level

- Level 0: No specification
- Level 1: Textual
- Level 2: Text with Diagrams
- Level 3: Models with Text
- Level 4: Precise Models
- Level 5: Models only

What to Model?
- Structures, Behaviors, Requirement

- Overall architecture of the system
- System dependencies
- Complexity
- Flow of information through a system
- Business requirements
- Database organization and structure
- Security features (attack models)
- Configuration and environment

....

How to Obtain Models?

- Manually construct
- Automatically transform from one model to another
- Automatically recover from the code

…….

Challenges

Create Software Models

- Modeling languages: https://en.wikipedia.org/wiki/Modeling_language
  - General purpose and domain-specific languages
  - Formalism
  - Level of abstraction

- Models for software running in different platforms
  - Model-driven architecture
  - Views: PIM (computation), CIM (environment), PSM

- Models for software consistently changing at runtime (agent)

- Modularity, separate concerns

Manage Software Models

- Find information from the models (query)
- Correctness of the models:
  - Model consistencies
  - Model checking models
- Transformations
  - Decomposition
  - Composition
  - Between models
- Evolutions of models

Use Software Models

- Generate code
- Monitor runtime software behavior (interacting with environments, adaptation)
- Testing (model-based testing criteria and test input generation)

UML Modeling - Overview

UML Modeling

- A language: syntax and semantics
- Capture ideas, relations, decisions, requirements in a well-defined notations

AgileData.org: … all developers should have a basic understanding of the industry-standard Unified Modeling Language (UML). A good starting point is to understand what I consider to be the core UML diagrams – use case diagrams, sequence diagrams, and class diagrams – although as I argued in An Introduction to Agile Modeling and Agile Documentation you must be willing to learn more models over time.

UML Diagrams

- Structural : relations of objects (class diagram, component diagram)
- Behavioral : sequence of actions (activity diagram, sequence diagram)

סיכום

- הקורס
- מבוא
  - מוטיבציה
  - מידול
- UML