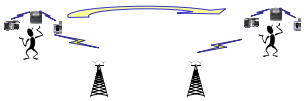


UML for Global Computing
Lecture 3: Property-Driven Development

Martin Wirsing
LMU München
in cooperation with
Hubert Baumeister

Contents

- Lecture 1: Introducing UML for Mobility
- Lecture 2: Refining Mobility Designs
- **Lecture 3: Property-driven Development of Mobile Systems**
  - **Development process**
  - **Case study: A Multi User Dungeon Game**
    - **Simple game**
    - **Game with logical distribution**

Property-Driven Design

- Joint development of test / specification / model
- Executable models
  - Immediate feedback
  - Automatic tests
- Refinement
  - Adding details
  - Refactoring
- Tools
  - Specification: JML / OCL
  - Tests: Fit and JUnit
  - Modeling language: UML and Java

Property-Driven Development: Process I

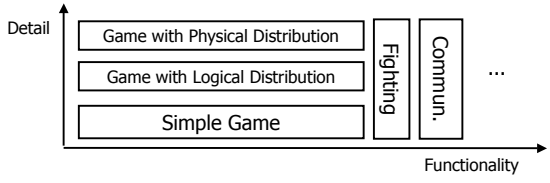
- Step 1: Requirements Capture
  - Develop User Stories (functional- and non functional requirements)
  - Define development strategy

Example: MUD Game

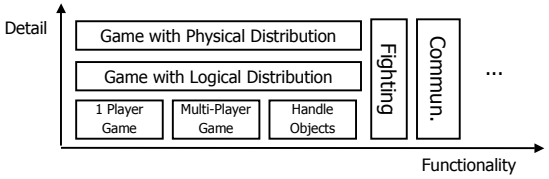
Requirements

- Multi User Dungeon Game — Functional Requirements
  - Player walks through rooms
  - Meets other human- /non human players
    - Talks
    - Fights
    - Trades
- Played via mobile phones — Non-functional Requirements
  - Distribution
  - Client server vs. Peer to Peer

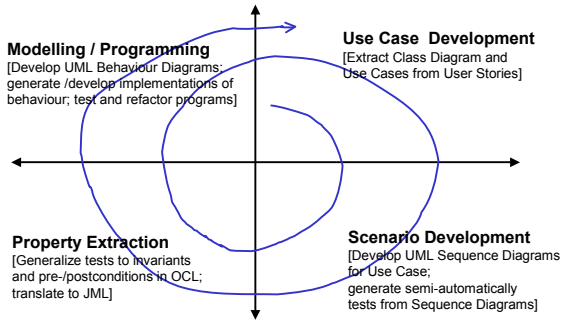
Development Strategy for the MUD Game



Development Strategy for the MUD Game



Property-Driven Develpmt: Design and Implementation



Property-Driven Development: Process II

- Step 2: Design and Implementation

  For each User Story iterate
  - Step 2.1: Use Case Development
    - Extract Use Cases from User Stories
  - Step 2.2 Scenario Development
    - Develop UML Sequence Diagrams for Use Case
    - Generate semi-automatically tests from Sequence Diagrams (generate test templates for FIT or Junit and complete templates with test data and expected results)
  - Step 2.3: Property Extraction
    - Generalize tests to invariants and pre-/postconditions in OCL/MTLA
    - Translate to JML
  - Step 2.4: Modelling / Programming
    - Develop UML Behaviour Diagrams (Sequence-, Activity Diagrams, and State Charts)
    - Generate /develop implementations of behaviors
    - Test and refactor programs

Property-Driven Design: Remarks

- Class Diagram evolves in parallel with the steps
- Different tools / notations for different levels of detail
  - High level
    - UML Diagrams
      - Sequence Diagrams, Activity Diagrams, State Charts, etc.
    - FIT acceptance tests
    - OCL / MTLA logic specifications
  - Low level
    - Java (or for mobile agents: Jade)
    - JUnit unit tests
    - JML logic specifications

Difference Use Case / User Story

- User Story
  - Tells an interesting "story" about the system that is relevant to the customer
  - Functional- / non-functional requirements
  - Includes Use Cases; however, usually less formal than use cases
- Use Case
  - Functionality of the System
  - Defined by
    - Pre- / Postconditions
    - Primary- / Secondary Scenarios.

Simple Game

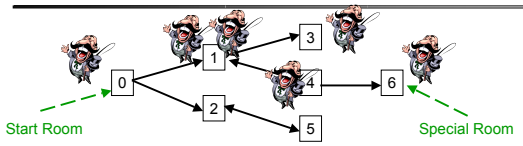
Example: Simple MUD-Game

- The Simple MUD Game consists of the User Stories:

1 Player Game	Multi-Player Game	Handle Objects

- First iteration: Develop 1 Player Game from User Story
- Second iteration: Develop Multi-Player Game from User Story
  - Extract Class diagram and Use Cases
  - Develop UML Sequence Diagrams and Acceptance Tests (FIT)
  - Find Properties: Invariants and Pre-/Post Conditions in OCL and translate to JML
  - Develop Implementation (in Java)
  - Run Acceptance Tests alone and with JML assertions

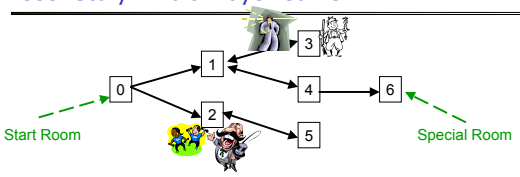
User Story: 1 Player Game



Game Rules

- The game has several Levels; each Level contains several Rooms which may be connected.
- The game starts in the Start Room of the lowest Level and ends in the highest Level.
- From the Special Room the player advances to the next Level or the game is over.
- The Player moves through the rooms until he finds the Special Room. He can see the rooms which are directly connected with its current room.
- The player moves always to a room which is directly connected to its current room.

User Story: Multi Player Game



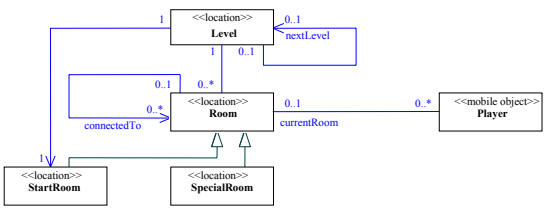
Additional Game Rules

- Several players in the game
- A player can see the other players in its room

Design of Class Diagram and Use Cases

- Extract classes (without operations) from User Story.
  - Room, Level, Player, Start Room, Special Room
- Define the conceptual mobility structure of the game:
  - Rooms and Levels are locations;
  - players are mobile objects
- Extract Use Cases from User Story
  - Look to connected Rooms
  - Move
    - Move to next Room
    - Advance to next level

Class Diagram for Multi Player Game

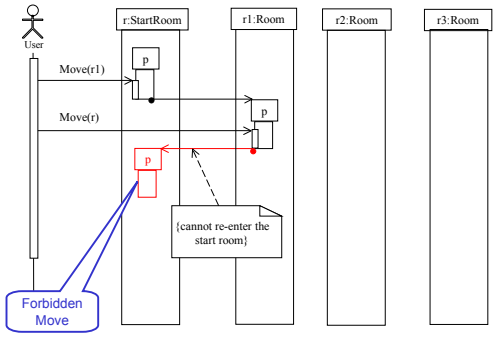


Use Cases for the Multi-Player Game

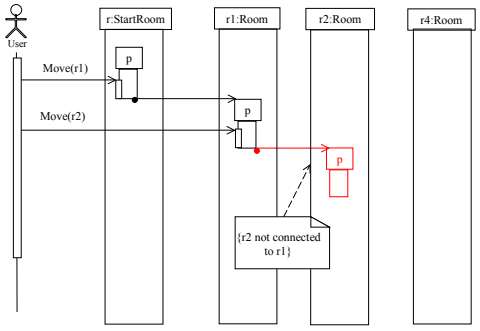
Development of Scenarios and Acceptance Tests

- Develop Sequence Diagrams for each Use Case
  - Standard behaviour (moves in black color are possible)
  - Forbidden behaviour (moves in red color are forbidden!)
- Derive Acceptance Test from Sequence Diagram
  - Create Test Template
  - Add parameter data and expected results
- Extend Class Diagram by the operations of the Acceptance Test?

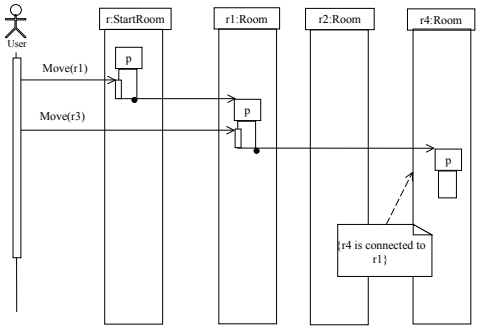
SDM: Scenario for moveToRoom Use Case



SDM: Scenario for move Use Case

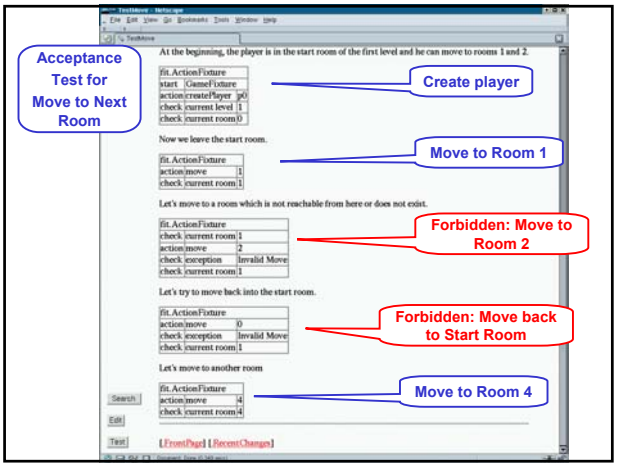


SDM: Scenario for move Use Case



Acceptance Tests with FITNesse

- FitNesse is a collaborative testing and documentation tool developed by R. Martin et al.
- It is a Wiki implemented in Java
- Test development:
  - Write acceptance tests in Wiki where every test is shown in a table
  - Implement a test fixture in Java for the operations in the test tables

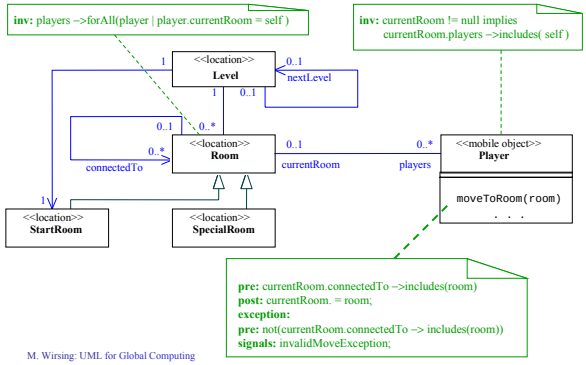


Derive Properties

- Extend the class diagram by the operations of discovered in the acceptance test (moveToRoom; . . .)
- Derive OCL pre- and post conditions for the operations
- Derive OCL invariants

- Translate OCL to JML
  which automatically creates assertions in the implementation

Class Diagram for Multi-Player Game



Translate OCL Pre/Post Conditons to JML

```
/*@
      public normal_behavior
        requires currentRoom().connectedTo().contains(room);
        ensures currentRoom() == room;
      also
      public exceptional_behavior
        requires !currentRoom().connectedTo().contains(room);
        assignable \nothing;
        signals (InvalidMoveException);
@*/

public void moveToRoom(Room room) throws InvalidMoveException …
```


Translate OCL Invariants to JML

```
public instance invariant
            (\forall Player player;
               players().contains(player);
               player.currentRoom() == this);

public instance invariant
    currentRoom() != null ==>
    currentRoom().players().contains(this);
```

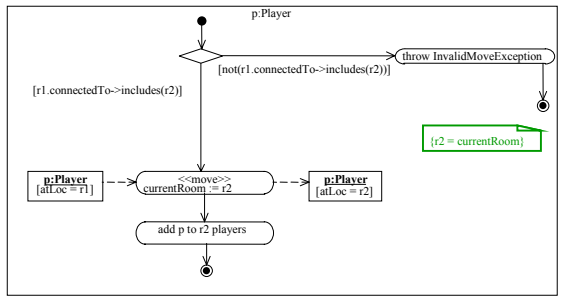
Invariante der Klasse Room

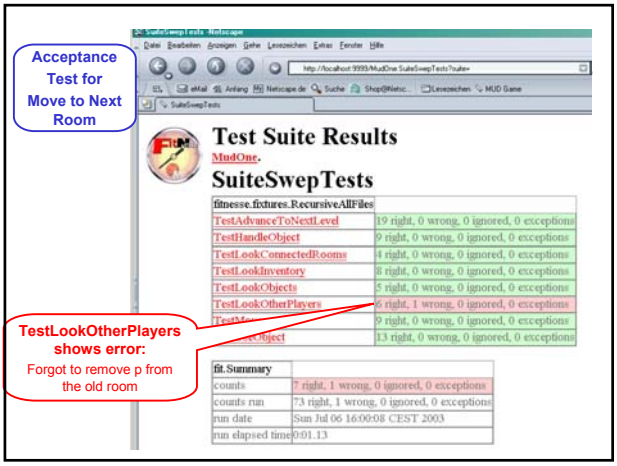
Invariante der Klasse Player

Step 2.4: Modelling / Programming

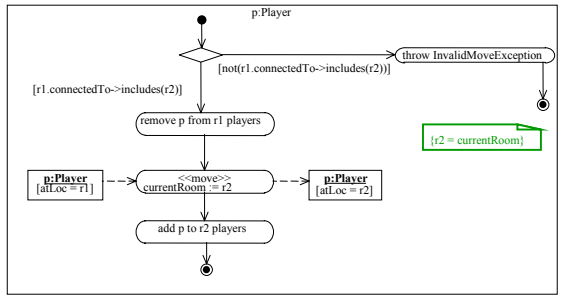
- Develop Activity Diagrams for moveToRoom()
  - Responsibility centered
  - Location centered
- Generate and test Java implementation
- Refactor, if necessary

Activity Diagram (RC) for `p.moveToRoom(r2)`

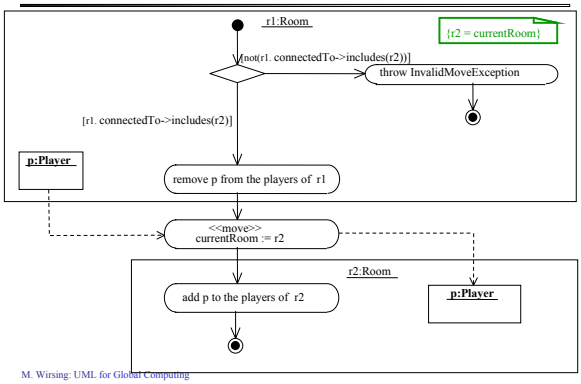


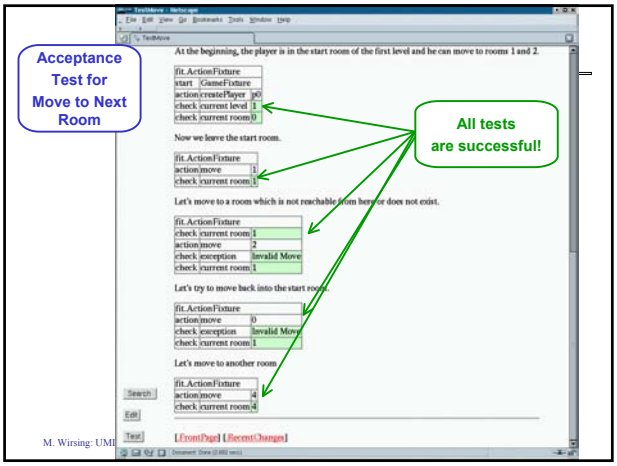


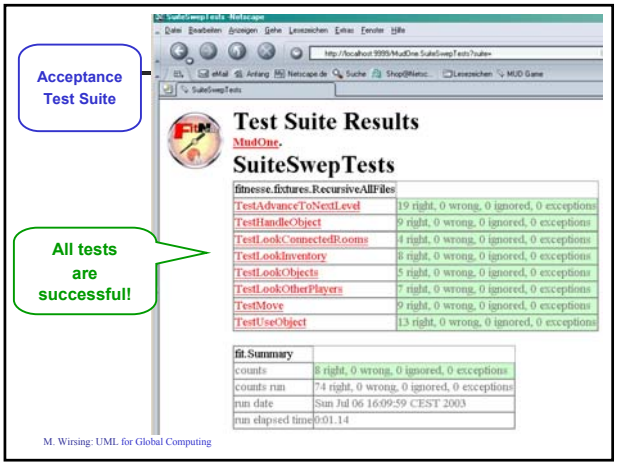
Corrected Diagram (RC) for `p.moveToRoom(r2)`



Activity Diagram (LC): `p.moveToRoom(r2)`







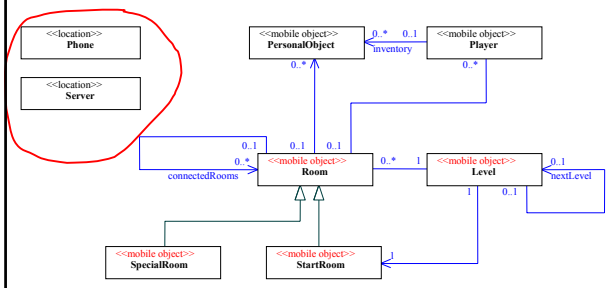
Game with Logical Distribution

MUD Game with Logical Distribution

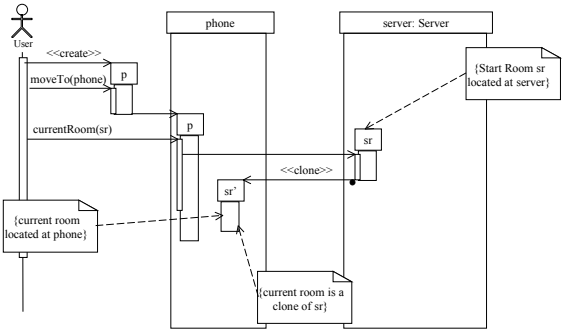
- User Story: Play the MUD Game with mobile phones:
  - Add server and mobile phones as locations
  - Players are located on phones
  - For playing the game, the current room is moved to the phone with the player

    → Rooms and Levels are mobile objects
- Property-driven development as before:
  - Refactor and extend Class Diagram
  - Develop Sequence Diagrams and tests
  - Derive OCL Properties
  - Develop UML Activity Diagram and generate and test implementation
  - Refactor, if necessary

Refactor and extend Class Diagram



Develop SDM: set current room



Unit Test with JUnit

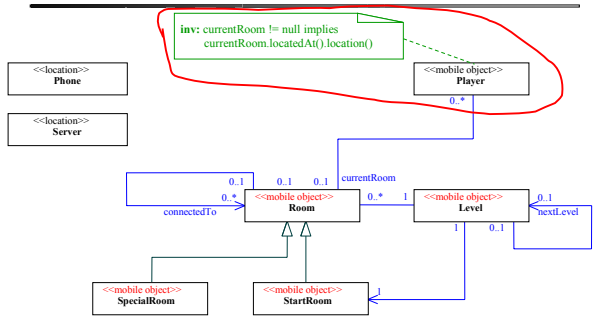
- JUnit is an „Open Source Framework“ for automatising Unit-Tests for Java.
- Developed by Kent Beck and Erich Gamma
- Test Case Design:
  - Generate test template from Sequence Diagram
  - Add assertions for the checking the current state

JUnit Test for „set Current Room“

```
public void testGameLocation() throws Exception {
   Server server = new Server();
   Phone phone = new Phone();

   Player player = new Player("Hubert");
   player.moveTo(phone);

   Level game = server.game();

   assertTrue(game.startRoom().locatedAt(server));

   player.currentRoom(game.startRoom());

   assertTrue(player.currentRoom().locatedAt(phone));
   assertTrue(player.currentRoom().cloneOf(game.startRoom()));
   assertTrue(game.startRoom().locatedAt(server));
}
```

Constructs clone of Start Room at phone

Properties: Additional Invariant for Player

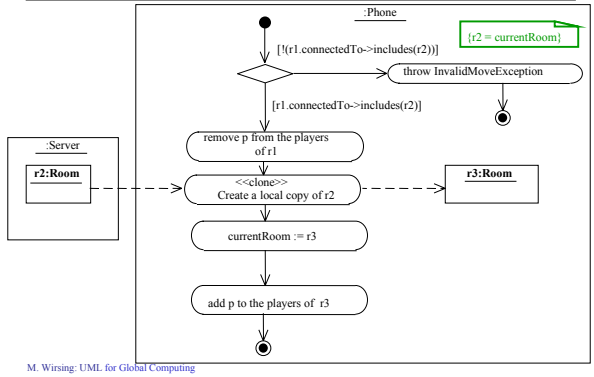


Translation to JML

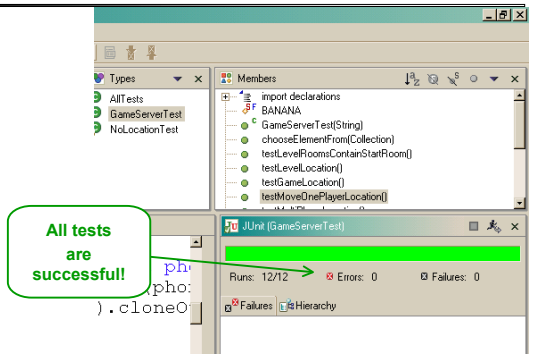
Additional Invariant of the Player

```
public instance invariant
      currentRoom() != null ==>
         currentRoom().locatedAt(location());
```


Activity Diagram (LC): MoveTo Method



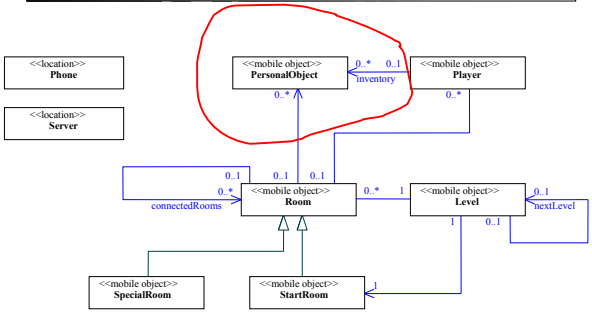
JUnit Test Run



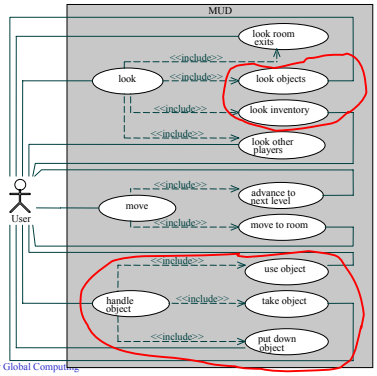
User Story „Handle Objects“

- User Story
  - Personal objects like bananas or apples can be in a room
  - A player can see the objects in the current room, he can use, take or put down an object

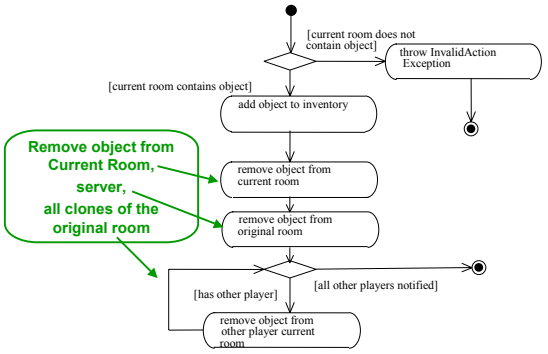
Class Diagram for “Handle Object”



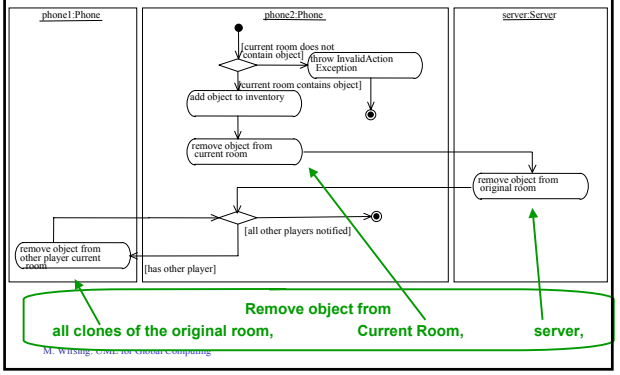
All Use Cases



Activity Diagram: TakeObjectFromRoom



Activity Diagram (LC): TakeObjectFromRoom



MUD Game with Physical Distribution

Implement MUD Game over

- Platform which supports mobility (e.g. Jade) or
- Use RMI and serialization of Java
  - Problem:
    RMI sends remote object with all its associated objects;
  - Remedy: Use explicit object names to separate objects from associated objects in order to send only the actual object

Summary

- Lecture 1: UML extension with mobility
  - First approach for explicit modeling of mobility in UML
  - Simple solution
  - Used already in industry and for teaching
    (DEGAS-Project, SWE-Praktikum)
- Lecture 2: Refinement
  - Simple syntactic refinement calculi for activity and sequence diagrams for mobility
  - MTLA as a formal basis for a UML notion of refinement:
    Refinement is implication (with possible hiding of variables or locations)!

Summary

- Lecture 3: Property-Driven Design
  - Executable models
    - Immediate feedback
    - Allows to experiment with the system
  - Joint development of formal properties and model
    - Tests
    - Formal specification
  - Tests + Refactoring = "Soft"ware

Future Work

- Test- / Verification Tool for State Charts
- Better Integration of Formal Methods
- Executable UML
  - Model-Driven Architecture (MDA)
    - Problems: User defined translations