Outline

- Motivation
- Definition and concepts
- Process and product properties
- Principles
Motivation
Software and the economy

- The economies of ALL developed nations are dependent on software.
- More and more systems are software controlled
- Expenditure on software represents a significant fraction of GNP in all developed countries.

*GNP: Gross National Product*
ICT Market – world - 2009

* ICT: information and communication technology
Sales

![Graph showing sales over time for PCs, Tablets, and Smartphones.](image-url)
Definitions and concepts
Software

- Software is a **collection of** computer programs, procedures, rules, associated documentation and data.
  - software development is more than merely the development of programs
  - software incorporates documents describing various views for various stakeholders (e.g. users, developers)
- **For a given problem, software is approximately 10 times more expensive to produce than a simple program**
  - Average: 10 to 50 LoC per person day
  - About 7 LoC in critical systems

* LoC: Lines Of Code
Types of Application Software:

- **Word Processing Software**: Allows users to create, edit a document. Example: MS Word, Word Pad etc.

- **Spreadsheet Software**: Allows users to create document and perform calculation. Example: Excel, Lotus1-2-3 etc.

- **Database Software**: Allows users to store and retrieve vast amount of data. Example: MS Access, MySQL, Oracle etc.

- **Presentation Graphic Software**: Allows users to create visual presentation. Example: MS Power Point

- **Multimedia Software**: Allows users to create image, audio, video etc. Example: Real Player, Media Player etc.
Ownership Rights and Delivery Methods

- **Commercial Software**: Installation in number of computers is specified by the software vendor/producer. User only buys the license to use it. User does not buy the software. He/she may not be allowed to install a software more than one machine. A demo version of software may exist for free but demo version does not include all the key components of the software.

- **Freeware**: Software that are given away for free by the vendor/producer. Example: Real Player trial version, MP3 Player trial version etc.
Software - criticality

- safety critical
  - aerospace, military, medical, ..
- mission critical
  - banking, logistics, industrial, production, ..
- other
  - games, ..
Complexity: Parts and interactions among parts.
- bicycle: 20 - 100
- car: 30,000
- airplane: 100,000
- cell phone, printer driver: 1M Lines of code
- cellular network, operating system: several Millions

Software systems are probably the most complex human artifacts.
Software complexity

- As of 2012, the Linux 3.2 release had 14,998,651 lines of code.[1]
- Windows 7 about 50 millions lines of code [2]
- An Android operating system in a smart phone consists of 12 million lines of code [3]
- The F-35 Joint Strike Fighter requires about 5.7 million lines of code to operate its onboard systems. [4]
- Boeing’s new 787 Dreamliner requires about 6.5 million lines of software code to operate its avionics and onboard support systems.[4]
- A bought a premium-class automobile recently, ”it probably contains close to 100 million lines of software code,” [4]


Diffusion

- **Local** (1945 – 1980): scientific community, military, banks, large private organizations
- **global** (1985 – today): ‘free’ hardware, huge diffusion of computing, impact on every day’s life
Misconceptions

- **Software is free**
  - a medium sized project with 50,000 LOC costs between $400,000 to $1,600,000 in personnel

- **Software is soft**
  - Yes, softer than hardware but changing it is difficult and costly

- **Software is produced**
  - Software is not mass produced
  - Software is developed

- **Software ages**
  - Software does not break as it ages
Software engineering

- Multi person construction of multi version software
- Not ‘solo programming’
Solo programming

- **Size**: small
  - One person can do it
- **Developer** is the user
  - No communication problems
- **Lifespan**: short
- **Cost**: limited (free)
- **Properties**: functional
Software engineering

- **Size**: large
  - Teams, documentation, communication and coordination problems
  - Modules and structure
- **User** is not the developer
  - 3rd party requirements, communication problems
- **Lifespan**: long (no ageing)
- **Cost**: development + operation/maintenance
- **Properties**: functional and not functional
Functional vs. non functional

- **Functional** characteristics of software
  - “Add two integer numbers”

- **Non functional** properties
  - Precision
    - relative error < $10^{-9}$
    - absolute error < $10^{-8}$
  - Reliability
  - Performance, efficiency
    - Sum must be performed < 0.01 millisecond
    - Sum must use < 10 kbytes ram memory
Non functional properties sometimes harder to express
Harder to design into software
They are emerging properties
Depend on the whole system, i.e.
- reliability, performance, ..
Process and product
Process and product

- Process: activities, people, tools
- Products: documents, data, code
- The quality of the product depends on the quality of the process
Process & product properties

- Process properties
  - cost
  - effort
  - punctuality

- Product properties
  - Functionality
  - Correctness
  - Reliability
  - Performance
Principles
Principles

- Separation of concerns
- Abstraction
- Modularity
Separation of concerns

- Given a large, difficult problem, try to split it into many (independent) parts, consider a part at a time
  - In war: divide and conquer
  - In SE: software process, concentrate on what the system should do, then on how, then do it.
Abstraction

Given a difficult problem/system, extract a simpler view of it, avoiding unneeded details.

```c
Main()
{  
    Int X = 0;
    X = pow(X, 2);
    Cout << X;
}
Int pow( int , int){
    ........
}
```
Modularity

- Divide a complex system in modules, with high cohesion and low coupling
Summary

- Software development is an important part of the economy.
- Software is not only computer programs.
- Software engineering considers techniques and methods to develop large, long-lived software, with many users.
- Software is characterized by its function, its correctness, reliability.
- Key guiding principles are separation of concerns, abstraction, modularity.
Software Engineering
Lecture 2: The whole picture

ABBAS HAIDER
M.Sc. In Computer Engineering
from POLITCNICO in Torino, Italy
Work in:
Control and Systems engineering department
University of technology
From the bottom up

- We need the final thing
  - Executable code
- But we do not write the executable
  - Source code
But the source code is large

- Several **physical** units
  - Files and directories

- Several **logical** units
  - Functions
  - classes
  - Packages
  - Subsystems

So, what units? How do we define and organize them?
But, exactly, what the software should do?

- Add numbers, count cars, forecast weather, control mobile phone, support administration of company
The development activities

- **Requirement engineering**
  - What the software should do

- **Architecture and design**
  - What units and how organized

- **Implementation**
  - Write source code, (executable code)
  - Integrate units
The development activities (2)

- Logically, each activity depends on the previous ones
  - To design, one must know the requirements
  - To implement, one must know the design and the requirements

- First approach is to do these activities in sequence
  - See waterfall model later

- In practice feedbacks and recycles must be provided

- Requirements and design are written down in documents
Development activities

- Requirements engineering
- Design document
- Design document
- Implement unit
- Implement unit
- Integrate units
- Unit
- Unit
- System
Ok, we did it

- Does it work?
- Is it doing what it should do?

Or

- Did we understand the requirements correctly?
- Did we implement the requirements correctly?
The V & V activities

- **V & V = verification and validation**

- **Control that the requirements are correct**
  - Externally: did we understand what the customer/user wants?
  - Internally: is the document consistent?

- **Control that the design is correct**
  - Externally: is the design capable of supporting the requirements?
  - Internally: is the design consistent?

- **Control that the code is correct**
  - Externally: is the code capable of supporting the requirements and the design?
  - Internally: is the code consistent (syntactic checks)?
Development + VV activities
Well, seems a lot of work

- Who does what, when?
- With what resources?
- How much will it cost, when will we finish?
- Where are the documents and units?
- Who can modify what?
- Are we doing it state of the art?
The management activities

- **Project management**
  - Assign work and monitor progress
  - Estimate and control budget

- **Configuration management**
  - Identify, store documents and units
  - Keep track of relationships and history

- **Quality assurance**
  - Define quality goals
  - Define how work will be done
  - Control results
The whole picture
SE in one slide

- Activities
  - Development, VV, management

- Documents (and code)
  - To share and control information, decisions

- Techniques
  - To support activities

- Languages
  - To write documents (UML), code
Software Engineering
Lecture 3: Requirement Engineering

ABBAS HAIDER
M.Sc. In Computer Engineering
from POLITCNICO in Torino, Italy
Work in:
Control and Systems engineering department
University of technology
The whole picture
Requirements engineering

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
What is a requirement?

- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
Activities in req. engineering

- Elicitation
- Analysis
- Formalization
- V&V
Stakeholders in req. engineering

- Stakeholder: Role or person with an interest (stake) in the system to be built

  - User
    * Uses the system
    * Can include different user profiles

  - Customer
    * Pays for the system

  - Administrator

  - Developer
Stakeholders - example

Account management system in a bank

- **User**
  - Clerk at counter (profile 1)
  - Bank customer at home (profile 2)

- **Customer**
  - CEO of bank and/or CTO (Chief technology officer) of bank

- **IT administrator**
  - Manages all applications in the bank

- **DB administrator**
  - Manages DBMSs on which applications are based

- **Security manager**
  - Responsible for security issues
Types of requirements

- User, system/developer
- Domain
- Functional, non functional
Requirement techniques

- Context diagram
- Requirement numbering and type
- Glossary
- Use case diagram (UML*)
- Scenario, sequence diagram (UML)
- System diagram (UML)
- Class diagram (UML)

* Unified Modeling Language
Requirement doc structure

- Overall description
- Stakeholders
- Context diagram and interfaces
- Requirements
  - Functional
  - Non functional
  - Domain
- Use case diagram
- Scenarios
- Class diagram
- Glossary
IEEE requirements standard

- IEEE Std 830 1984
- Defines a **generic structure** for a requirements document that must be instantiated for each specific system.
  - Introduction.
  - General description.
  - Specific requirements.
  - Appendices.
  - Index.
Context diagram

- Defines what is **inside** the system to be developed, what is **outside**
  - Other systems/subsystems/applications
  - Human users

- Defines **interfaces** between inside and outside
Interfaces

- **With cashier**
  - Physical level: Screen, keyboard
  - Logical: GUI (to be described)

- **With product**
  - Physical level: laser beam (bar code reader)
  - Logical level: bar code

- **With credit card system**
  - Physical level: internet connection
  - Logical: web services
Requirement types

- Functional
  - Description of services / behaviors provided by the system
- Non functional
  - Constraints on the services
<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Handle sale transaction</td>
</tr>
<tr>
<td>F2</td>
<td>Start sale transaction</td>
</tr>
<tr>
<td>F3</td>
<td>End sale transaction</td>
</tr>
<tr>
<td>F4</td>
<td>Log in</td>
</tr>
<tr>
<td>F5</td>
<td>Log out</td>
</tr>
<tr>
<td>F6</td>
<td>Read bar code</td>
</tr>
</tbody>
</table>
Non Functional

- Efficiency
- Usability
- Reliability
- …
Non-functional reqs (more info)

- **Product requirements**
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

- **Organizational requirements**
  - Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.

- **External requirements**
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.
Conflicts between different nonfunctional requirements are common in complex systems.

Spacecraft system

- To minimize weight, the number of separate chips in the system should be minimized.
- To minimize power consumption, lower power chips should be used.

- However, using low power chips may mean that more chips have to be used. Which is the most critical requirement?
Derived from the application domain and describe system characteristics and features that reflect the domain.

Domain requirements can be new functional requirements, constraints on existing requirements or define specific computations.

If domain requirements are not satisfied, the system may be unworkable.
Sequence of steps that describe a typical interaction with the system

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start sales transaction</td>
</tr>
<tr>
<td>2</td>
<td>Read bar code</td>
</tr>
<tr>
<td>3</td>
<td>Retrieve name and price given barcode</td>
</tr>
<tr>
<td>4</td>
<td>Repeat 2 and 3 for all products</td>
</tr>
<tr>
<td>5</td>
<td>Compute total</td>
</tr>
<tr>
<td>6</td>
<td>Manage payment cash</td>
</tr>
<tr>
<td>7</td>
<td>Print receipt</td>
</tr>
</tbody>
</table>
Use case

- Set of scenarios with common user goal
V&V of requirements

- Natural language, UML
  - Inspection, reading
  - By user, by developer
- UML
- Some syntactic check by tools
- Prototyping
- Formal language
- Model checking
Goal of requirement engineering is describing what the system should do in a requirement document.

Many stakeholders are involved in the process.

Techniques to make the document more precise are:
- Context diagram and interfaces
- Identifying requirements and classifying them (functional, non-functional, domain)

Scenarios

Use cases
Prepare next week for Quiz
Software Engineering
Lecture 4: V & V

HAIDER MUHI ABBAS
Control and Systems engineering department
University of technology
The whole picture

- Requirements engineering
  - Requirement document
  - VV requirements
  - Requirement document

- Design
  - Design document
  - VV design
  - Design document

- Implement unit
  - Unit
  - VV unit
  - Unit

- Implement unit
  - Unit
  - VV unit
  - Unit

- Integrate units
  - System
  - VV system
  - System

- Project management
  - Configuration management
  - Quality management
Validation
- is it the right software system?
- effectiveness
- external (vs user)
- reliability

Verification
- is the software system right?
- efficiency
- internal (vertical transformation of documents is done correctly)
- correctness
V&V

Verification

Requirement document

Design document

Unit

System

Validation

Stakeholders
V & V vs. cost of fixing defect

- Requirement document
  - Design document
    - Unit
      - Unit
        - System
  - Cost of fixing defect
Failure, fault, defect

❖ Failure
- An execution event where the software behaves in an unexpected way

❖ Fault
- The feature of software that causes a failure
- May be due to:
  > An error in software
  > Incomplete/incorrect requirements

❖ Defect
- Failure or fault
Defect is characterized by

- Insertion activity (phase)
- Removal activity (phase)
Basic goal of VV

- Minimize number of defects inserted
  - Cannot be zero due to inherent complexity of software
- Maximize number of defects discovered and removed
- Minimize time span between insertion and discovery and removal
The longer the delay insert-remove, the higher the cost of removing defect
V&V techniques

- Static
  - inspections
  - source code analysis
- Dynamic
  - testing
Inspections
Benefits

- Early defect detection improves product quality and reduces avoidable rework (down to 10-20%).
Inspection

- Consists in
  - reading documents/code
  - By a group of people (3+, group dynamics)
  - With goal of finding defects (no correction)

- Can find many defects
  - Test concentrates one defect at a time
V&V: Testing

- Static
  - inspections
  - source code analysis
- Dynamic
  - testing
Testing

- Dynamic technique, requires execution of **executable** system or executable unit
  - system test
  - unit test
Defect testing and debugging are different activities.

Testing tries to find failures.

Debugging searches for and removes the fault.
Debugging

1. Test case result (failure)
2. Design doc
3. Locate fault
4. Design fault repair
5. Repair (code)
6. Test suite
7. Re-test
Test case

- Certain stimulus applied to executable (system or unit), composed of:
  - name
  - input (or sequence of)
  - expected output
- With defined constraints/context
  ex. version and type of OS, DBMS, GUI

..
Test suite

- Set of (related) test cases
Test case log

- Test case +
  - Time and date of application
  - Actual output
  - Result (pass / no pass)
Example

- Function Plus(x,y)
- Test case:
  - T1(1,1; 2)
  - T2(3,5; 8)
- Test suite
  - TS1{T1, T2}
- Test log
  - T1, 16-3-2009 9:31, result 2, success
  - T2, 16-3-2009 9:32, result 9, fail
Test activities

- Write test cases
  - Test case, test suite
- Run test case (test suite)
- Record results
  - Test case log
The ideal condition would be to have an automatic oracle and an automatic comparator
- The former is very difficult to have
- The latter is available only in some cases

A human oracle is subject to errors

The oracle is based on the program specifications (which can be wrong)
Oracle

- Necessary condition to perform testing:
  - Know the expected behavior of a program for a given test case (oracle)
- Human oracle
  - Based on req. specification or judgment
- Automatic oracle
  - Generated from (formal) req. specification
  - Same software developed by others
  - Previous version of the program (regression)
Theory and constraints
Exhaustive test

- function: \( Y = A + B \)
- A and B integers, 32 bit
- Total number of test cases:
  \[ 2^{32} \times 2^{32} = 2^{64} \gg 10^{20} \]
- 1 ms/test \( \Rightarrow \) 3000 billion years
Exhaustive test is **not possible**

So **goal of test** is finding defects, not demonstrating that systems is defect free

Goal of test (and VV in general) is assuring a good enough level of confidence
Howden theorem

- For an arbitrary program P it is impossible to find an algorithm that is able to generate a finite ideal test (that is selected by a valid and reliable criterion)
A developer is **unsuitable** to test his/her own code
Test classification

- Per phase/granularity level
  - Unit, integration, system
- Per approach
  - Black box (functional)
  - White box (structural)
  - Reliability assessment/prediction
  - Risk based (safety security)
Test per granularity level/phase

- Unit tests
  - Individual modules
- Integration tests
  - Modules when working together
- System tests
  - The system as a whole (usable system)
<table>
<thead>
<tr>
<th></th>
<th>Unit</th>
<th>Integration</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional / black box</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Structural / white box</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Risks</td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>
Software Engineering
Lecture 5: Unit test

HAIDER MUHI ABBAS
Control and Systems engineering department
University of technology
Test per granularity level/phase

- **Unit tests**
  - Individual modules
- **Integration tests**
  - Modules when working together
- **System tests**
  - The system as a whole (usable system)
Unit test

Requirements engineering → Requirement document → VV requirements → Requirement document

Design

Design document → Design document

Implement unit

Implement unit → VV unit

Integrate units

Unit → Unit

System

VV system

Project management
Configuration management
Quality assurance
Unit test

- Black box (functional)
  - Random
  - Equivalence classes partitioning
  - Boundary conditions
- White Box (structural)
  - Coverage of structural elements
Function double squareRoot(double x);
- Extract randomly x
  \( \rightarrow T1 \ (3.0 \ ; \ \sqrt{3}) \)
  \( \rightarrow T2 \ (1000.8 \ ; \ \sqrt{1000.8}) \)
  \( \rightarrow T3 \ (-1223.7; \text{error}) \)

Function double invert(double x);
- Extract randomly x
  \( \rightarrow T1 \ (1.0 \ ; \ 1.0) \)
  \( \rightarrow T2 \ (-2.0 \ ; \ -0.5) \)
Random

- **Pros**
  - Indipendent of requirements

- **Cons**
  - Requires many test cases (easy to define)
  - the inputs, requires Oracle to compute
  - the expected output
Equivalence classes partitioning

- Divide input space in partitions
  - that have similar behavior from point of view of (requirements for) unit
- Take one / two test cases per partition
- Boundary conditions
- Boundary between partitions
- Take test cases on the boundary
Equivalence classes
A class corresponds to set of valid or invalid inputs for a condition on the input variables.

If a test in a class has not success the other tests in the same class may have the same behavior.
Common conditions:
- Single value: age = 33
- Interval: age between 0 and 200
- Boolean: married = true or false
- Discrete set: marital status = single, married, divorced
# Conditions and classes

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single value</td>
<td>Valid value, invalid values &lt; value</td>
<td>Age = 33</td>
</tr>
<tr>
<td></td>
<td>Invalid values &gt; value</td>
<td>Age &lt; 33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Age &gt; 33</td>
</tr>
<tr>
<td>Interval</td>
<td>Inside interval, Outside one side</td>
<td>Age &gt; 0 and age &lt; 200</td>
</tr>
<tr>
<td></td>
<td>Outside, other side</td>
<td>Age &gt; 200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Age &lt; 0</td>
</tr>
<tr>
<td>Boolean</td>
<td>True</td>
<td>Married = true</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>Married = false</td>
</tr>
<tr>
<td>Discrete set</td>
<td>Each value in set</td>
<td>Status = single</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Status = married</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Status = divorced</td>
</tr>
<tr>
<td></td>
<td>One value outside set</td>
<td>Status = jksfhj</td>
</tr>
</tbody>
</table>
Selection of test cases

- Every equivalence class must be covered by a test case at least
- A test case for each invalid input class
- Each test case for valid input classes must cover as many (remaining) valid classes
- as possible
Equivalence classes

- Function double squareRoot(double x);
- Partitions
  - Positive numbers $T_1$ (1; $\sqrt{1}$)
  - Negative numbers $T_2$ (-1; error)
  - Boundary: zero, infinite
  - Zero and close
    - $T_3$ (0; $\sqrt{0}$) $T_4$(0.01; $\sqrt{0.01}$) $T_5$(-0.01; error)
    - ‘Infinite’ and close
      - $T_6$ (maxdouble; $\sqrt{\text{maxdouble}}$) $T_7$
      - (maxdouble+0.01; err)
      - $T_8$ (mindouble; $\sqrt{\text{mindouble}}$) $T_7$ (mindouble-0.01; err)
Equivalence classes

- int convert(String s)
- converts a sequence of chars (max 6) in an integer
- number. Negative numbers start with a ‘-‘

<table>
<thead>
<tr>
<th>Criterion to define the class</th>
<th>Equivalence classes and test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>String represents a well formed integer</td>
<td>Yes T1(&quot;123&quot;; 123)</td>
</tr>
<tr>
<td>Sign of number</td>
<td>Positive T1(&quot;123&quot;; 123)</td>
</tr>
<tr>
<td>Number of characters</td>
<td>&lt;=6 T1(&quot;123&quot;; 123)</td>
</tr>
<tr>
<td>WF integer</td>
<td>sign</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>yes</td>
<td>Pos</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Neg</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>Pos</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Neg</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Boundary - combinatorial

<table>
<thead>
<tr>
<th>WF integer</th>
<th>sign</th>
<th>N char</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>Pos</td>
<td>&lt;= 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“0” “999999”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“1000000” “9999999”</td>
</tr>
<tr>
<td>Neg</td>
<td>&lt;= 6</td>
<td>“-0” “-99999”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“-999999”</td>
</tr>
<tr>
<td>no</td>
<td>Pos</td>
<td>&lt;= 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“ ”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“ ” (7 blanks)</td>
</tr>
<tr>
<td>Neg</td>
<td>&lt;= 6</td>
<td>“-”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt; 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“ - - - - - - - ”</td>
</tr>
</tbody>
</table>
Equiv classes and state

- When a module has state
- the state has to be considered to define
- the partitions
- State may be difficult to read/create
- Requires a sequence of calls
Equiv classes and state

- double Ave3(int i)
- Computes average of last three numbers passed, excluding the negative ones
- Criteria
  - state: n elements received
  - int i: positive, negative
### Equiv classes and state

<table>
<thead>
<tr>
<th>N elements</th>
<th>I</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Pos</td>
<td>T1(10; 10)</td>
</tr>
<tr>
<td></td>
<td>Neg</td>
<td>T2(-10; ?)</td>
</tr>
<tr>
<td>2</td>
<td>Pos</td>
<td>T3(10,20 ; 15)</td>
</tr>
<tr>
<td></td>
<td>Neg</td>
<td>T4(-10,-20; ?)</td>
</tr>
<tr>
<td>3</td>
<td>Pos</td>
<td>T5(10,2,6; 6)</td>
</tr>
<tr>
<td></td>
<td>Neg</td>
<td>T6(-10,-2,-6; ?)</td>
</tr>
<tr>
<td>&gt;3</td>
<td>Pos</td>
<td>T7(1,2,3,4; 2.5)</td>
</tr>
<tr>
<td></td>
<td>Neg</td>
<td>T8(-1,-2,-3,-4; ?)</td>
</tr>
</tbody>
</table>
Test of OO classes

- Have state
- Many functions to be tested
- Identify criteria and classes
- Apply them to each function
public class EventsQueue{
    // receives and stores events, with time tag
    public void reset(); // cancels all events
    public void push(int timeTag) throws InvalidTag
        // discards events with negative or zero time tag
        // and with time tag already existing
    public int pop() throws EmptyQueue
        // returns and cancels event with lower time tag
        // raises exception if queue empty
}
Unit test black box - summary

- Functional test of units (functions, classes) generates test cases starting from the specification of the unit
- Key techniques are
  - Random
  - Equivalence classes partitioning
  - Boundary conditions
System documentation of a heating control system

This document has been produced, within the context of software engineering courses at the University of Kaiserlslautern by Marco Heide, Wolfgang Wagenbichler, Andreas Schank, Angela Schmidt and Axel Seck, then reworked by Christiane Differding and Matthias Gutheil. It has been translated in english and adapted by Mario Negro Ponzi of Politecnico di Torino. It has subsequently undergone extensive rework by Vincenzo D’Elia, Alfredo Pironti and Massimiliano Schillaci of Politecnico di Torino.

V 3.1 march 2009

Abstract

A heating control system monitors and control temperature in a building composed of several rooms. It is a good example of a software system. The related documentation is divided in three parts: a problem-solution documentation, software-system documentation and software-component documentation. This document refers to the first part.

Table of contents

Part 1 Problem solution documentation ........................................................................ 2
  1.1 Problem description ........................................................................................... 2
  1.1.1 Problem ....................................................................................................... 2
  1.1.2 Expected functionalities .............................................................................. 2
  1.1.3 Environment characteristics ........................................................................ 3
  1.2 Requirements / specifications ............................................................................. 3
  1.2.1 User requirements / specifications .............................................................. 3
  1.2.2 User requirements analysis ......................................................................... 5
  1.2.3 Developer’s Requirements/Specifications .................................................. 9
  1.2.4 Traceability matrix: user requirements <-> class diagrams ....................... 38
Part 1  Problem solution documentation

1.1  Problem description
The following problem description is an implementation and adjustment of a heating control system. A house is composed by many subsystems (e.g. rooms or devices as heaters), that are connected with doors and tubes. It is also connected with the outside world with doors and windows. A house is also made of a texture of subsystems that strictly interact with each other. A software system should be developed that controls and manages this complex system. It should monitor weather conditions and optimize energy consumption.

Below the idea of this system is described. A user interface for getting and setting room parameters to the system is an integral part of the system.

1.1.1  Problem
Up until today the temperature of every room is regulated by a thermostat in the heater. A target temperature for each room can be set. An automatic management of the temperature is based only on heaters. A window also is useful to control temperature in a single room. But it should be opened or closed by someone. Water in heaters has a fixed temperature set by a boiler.

Problems in temperature management are:
1. Heater’s temperature cannot be differently set basing on heating demand or time. There is no regulation linked with people presence
2. Thermal regulation is limited to heating. The use of windows to cool down is based on user intervention
3. There is no optimization of boiler temperature. The boiler’s temperature is set to the maximum value regardless whether the temperature is too low or too high. That produces an energy loss.

The above control system is not satisfying both because of energy loss and because it is not comfortable.

1.1.2  Expected functionalities
The user wants to be able to individually set temperature for each room using a terminal. So he should be able to provide described parameters.

First it should be possible to set a temperature that could be maintained when there is someone in the room. In doing so the system should not react immediately when someone comes in. It should be provided a time span that indicates how long a person should stay in the room before the system takes control. The user should also be able to set the time span in which the desired temperature should be reached. The provided time span can be maintained heating up heaters for a small period of time over the desired temperature. Also the user should be able to set a minimum temperature that is acceptable when there is no one in the room. In this case, too, a time span should be provided, after everyone has left the room, before the system takes control.

There should also be default values available for every parameter.

Windows, too, should be controlled by the system. If a window has been completely opened by a user, the system should maintain only the minimal temperature. If the window is open and it starts raining the system should put it in tilt position. More, the windows should be usable by the system to cool down when the outside temperature is cooler, by tilting it.
1.1.3 Environment characteristics
Below are described the characteristics of the environment inside which the system will operate.

1.1.3.1 House
The house is composed by the following rooms:
- living room (40sqm)
- eating room (15sqm)
- sleeping room (15sqm)
- children room (20sqm)
- visitors room (15sqm)
- work room (10sqm)
- kitchen (15sqm)
- bathroom (5sqm)
- visitors bathroom (5sqm)
- hall (10sqm)

The house also has:
- an outside temperature sensor
- a rain switch (closed if it rains)

1.1.3.2 Room
Each room is so configured (as a result of the whole system concept):
- 1 window with two switches and two actuators. A switch gets opened when the window is completely open (open when the window is open / NC = normally closed). The other when is tilted (open when the window is tilted / NC = normally closed). When the window is completely open the tilt switch is closed. An actuator swings the window while the other tilts it.
- 1 heater (hot water) with a thermostat
- 1 presence switch (closed when someone is in the room)
- 1 room temperature sensor

1.1.3.3 Centralized heating room
The control of the heating system is an already deployed autonomous subsystem. The maximum temperature is fixed at 70 degrees.
The heating pump is controlled with impulses and it is so robust that there is no loss in the way between boiler and heaters.

1.2 Requirements / specifications
Below system requirements are depicted. They will be divided in User and Developer requirements. As notation UML will be used. The UML diagrams to describe requirements will be ordered in User-Requirements and Developer-Requirements. The ordering follows the comprehensibility degree of the model from the point of view of the user. To user requirements also belong Use Cases and Scenarios. The rest of the model is in the Developer requirements described in chapter 1.3.2. There will also be a modeling of dynamic views.

1.2.1 User requirements / specifications
Below will be described functional, not functional and inverse user Requirements. Then user’s conceptual choices will be described. In chapter 1.2.2 Use Case diagrams and Scenarios will be depicted that are based on User Requirements.
1.2.1.1 Functional User Requirements

Functional User Requirements are originated by problem description (chapter 1.2) and from opinions that are collected in interviews with the user.

<table>
<thead>
<tr>
<th>UR-F</th>
<th>Requirement’s description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp-UR-F 1</td>
<td>The user shall be able to set StandardPresenceTemp in °C</td>
</tr>
<tr>
<td>Temp-UR-F 2</td>
<td>The user shall be able to set StandardAwayTemp in °C</td>
</tr>
<tr>
<td>Temp-UR-F 3</td>
<td>The user shall be able to set a time span StandardHeatingTime in minutes</td>
</tr>
<tr>
<td>Temp-UR-F 4</td>
<td>The user shall be able to set a time span StandardAwayTime in minutes</td>
</tr>
<tr>
<td>Temp-UR-F 5</td>
<td>The user shall be able to set a time span StandardInsideTime in minutes</td>
</tr>
<tr>
<td>Temp-UR-F 6</td>
<td>The temperature of Temp-UR-F 1 shall be maintained when there is someone in the room and the StandardInsideTime from Temp-UR-F 5 has elapsed</td>
</tr>
<tr>
<td>Temp-UR-F 7</td>
<td>The temperature of Temp-UR-F 2 should be maintained in the room when the room is empty and StandardAwayTime from Temp-UR-F 4 has elapsed</td>
</tr>
<tr>
<td>Temp-UR-F 8</td>
<td>The time span from Temp-UR-F 3 indicates after how many minutes the StandardPresenceTemp from Temp-UR-F 1 after the entrance of a person in the room shall be reached.</td>
</tr>
<tr>
<td>Temp-UR-F 9</td>
<td>The time span from Temp-UR-F 4 indicates for how many minutes the StandardPresenceTemp from Temp-UR-F 1 shall be maintained after the last person has left the room</td>
</tr>
<tr>
<td>Temp-UR-F 10</td>
<td>The time span from Temp-UR-F 5 indicates for how long shall someone stay in the room for the system to get in control</td>
</tr>
<tr>
<td>Temp-UR-F 11</td>
<td>The StandardHeatingTime from Temp-UR-F 3 must be at least 5 minutes longer than the StandardInsideTime from Temp-UR-F 5</td>
</tr>
<tr>
<td>Temp-UR-F 12</td>
<td>For each quantity from Temp-UR-F 1, Temp-UR-F 2, Temp-UR-F 3, Temp-UR-F 4, Temp-UR-F 5 default values shall exist.</td>
</tr>
<tr>
<td>Temp-UR-F 13</td>
<td>If a window gets opened the system maintains the room temperature at StandardAwayTemp, whether or not there is someone in the room.</td>
</tr>
<tr>
<td>Temp-UR-F 14</td>
<td>If it starts raining with the window open, the system will set the window in a rain secure position.</td>
</tr>
<tr>
<td>Temp-UR-F 15</td>
<td>When the temperature of a room with someone inside exceeds that from Temp-UR-F 1 and the outer temperature is lower than the room temperature, the window in this room gets tilted</td>
</tr>
<tr>
<td>Temp-UR-F 16</td>
<td>The control of the temperature is admitted with an error of +/-1 °C</td>
</tr>
<tr>
<td>Temp-UR-F 17</td>
<td>The system is planned for the room described on page 4</td>
</tr>
<tr>
<td>Temp-UR-F 18</td>
<td>The temperature of the boiler shall be set to the maximum required by any heater, not to waste energy.</td>
</tr>
</tbody>
</table>

Table 1

1.2.1.2 Non-functional user requirements

The system should comply also with the following requirements.

<table>
<thead>
<tr>
<th>UR-NF</th>
<th>Requirement’s description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UR-NF 1</td>
<td>The system should be easily changeable.</td>
</tr>
</tbody>
</table>

Table 2

1.2.1.3 Inverse user requirements

Following situations should be avoided.

<table>
<thead>
<tr>
<th>UR-Inv</th>
<th>Requirement’s description</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>UR-Inv 1</th>
<th>When there is nobody in the room and the room is hotter than target temperature, the window does not get tilted to cool.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UR-Inv 2</td>
<td>If the user manually opens the window, the system shall not tilt it to the rain secure position.</td>
</tr>
</tbody>
</table>

Table 3

1.2.1.4 Design decisions
Following decisions have been taken.

<table>
<thead>
<tr>
<th>DD 1</th>
<th>The system should be object oriented.</th>
</tr>
</thead>
</table>

Table 4

1.2.2 User requirements analysis
In the following UML use case diagrams and Scenarios derived from user requirements are reported, as noted in the introduction of chapter 1.2.1.

1.2.2.1 Use case diagram

Use cases typically depict interactions between external actors and the system from the point of view of the user. The interaction among actors and the system starts with initial events triggered by the actor on the system and ends when the logical reaction of the system has finished.

**Use Case: presence**
The activating actor of this Use Case is the user. The presence of a person in the room is notified to the system by the presence switch.
This Use Case will be shown in the following scenarios:
- **Scenario 2**
- Scenario 3
- Scenario 4

**Use Case: operateWindow**
The activating actor of this Use Case is the user that opens or closes the window. The system detects the changed position through the open window switch and the tilted window switch.
This Use Case will be shown in the following scenario:
- Scenario 5

**Use Case: heatingControl**
This Use Case describes the recalculation of the temperature of the heater (to heat, maintain or cool). At the same time the new temperature of the boiler is notified.
This Use Case will be shown in the following scenarios:
- Scenario 7
- Scenario 9
- Scenario 10
- Scenario 11

**Use Case: weather**
The activating actor of this Use Case is the environment. The rain sensor is monitored by the system to discover whether it’s raining on the house.
This Use Case will be shown in the following Scenario:
- Scenario 6

**Use Case: setValues**
The user sets various values. The activating actor is the user.
This Use Case will be shown in the following scenarios:
- Scenario 1
- Scenario 8

### 1.2.2.2 Scenarios for Use Cases
The following scenarios describe the system in common situations and realize the Use Cases. So they must show every functional and inverse user requirement.

**Other notes (to the following table):**
- (-) after a User Requirement indicates that a specific precondition (e.g. a time span elapsing) has not been met
- (+) after a User Requirement indicates that a specific precondition has been met
- Without any other specific note, in the following scenarios those following default values should be considered:
  o StandardPresenceTemp: 20 °C
  o StandardAwayTemp: 15 °C
  o StandardHeatingTime: 10 minutes
  o StandardInsideTime: 5 minutes
  o StandardAwayTime: 15 minutes

<table>
<thead>
<tr>
<th>#</th>
<th>Step</th>
<th>Scenario’s description</th>
<th>User requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scenario for Use Case setValues. User changed StandardHeatingTime, StandardInsideTime and StandardPresenceTemp.</td>
<td>The room will be heated up.</td>
<td></td>
</tr>
</tbody>
</table>
**Scenario for Use Case presence**

The user leaves the room before the system activates

1. **Presence switch**: closed  
   **Outside Temperature**: 21 ºC  
   **Window**: closed  
   **Rain switch**: open  
   **Target temperature**: 19 ºC (StandardPresenceTemp)  
   **Room temperature**: 18 ºC

2. The user sets StandardInsideTime to 5 minutes

3. The user sets StandardHeatingTime to 9 minutes

4. The system ignores new time set, because it is just 4 minutes over StandardInsideTime and throws and error message

5. The user sets StandardHeatingTime to 10 minutes

6. The user sets StandardAwayTemp to 16 ºC

7. The user sets StandardAwayTime to 10 minutes

8. The user sets StandardPresenceTemp to 21 ºC

9. The target temperature is set at 21 ºC

-> Use Case Heating control

**Scenario for Use Case presence**

The user leaves the room for a long time, affecting system behavior

1. **Presence switch**: closed  
   **Outside temperature**: 16 ºC  
   **Window**: closed  
   **Rain sensor**: open  
   **Target temperature**: 20 ºC (StandardPresenceTemp)  
   **Room temperature**: 19 ºC

2. The user leaves the room, which becomes empty

3. After StandardAwayTime the target temperature is set to StandardAwayTemp.

4. The window does not get opened because there is no one inside the room.

-> Use Case Heating control

**Scenario for Use Case presence**

There is only one user in the room. This user leaves for a short time the room so that it does not have any effect on the system behavior

1. **Presence switch**: closed  
   **Outside temperature**: 19 ºC  
   **Window**: closed  
   **Rain sensor**: open  
   **Target temperature**: 20 ºC (StandardPresenceTemp)  
   **Room temperature**: 19 ºC

2. The user leaves the room, which becomes empty
3. The user comes back in the room before StandardAwayTime has elapsed.  
4. The temperature does not change  

Scenario for use Case operateWindow  
The window is open and a user is present. Target temperature is set to StandardAwayTemp

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1.   | Presence switch: closed  
Outside temperature: 19 ºC  
Window: closed  
Rain sensor: open  
Target temperature: 20 ºC (StandardPresenceTemp)  
Room temperature: 20 ºC |
| 2.   | The user opens the window |
| 3.   | Target temperature is set to StandardAwayTemp |

-> Use Case Heating Control

Scenario for Use Case weather  
It rains with the window open

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1.   | Presence switch: closed  
Outside temperature: 17 ºC  
Window: completely open  
Rain sensor: open  
Target temperature: 15 ºC (StandardAwayTemp)  
Room temperature: 18 ºC |
| 2.   | It starts raining |
| 3.   | The system closes the window |
| 4.   | Target temperature is set to StandardPresenceTemp |

-> Use Case Heating Control

Scenario for Use Case heatingControl  
The window gets controlled by the system to cool

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1.   | Presence switch: closed  
Outside temperature: 17 ºC  
Window: closed  
Rain sensor: open  
Target temperature: 20 ºC (StandardPresenceTemp)  
Room temperature: 21 ºC |
| 2.   | The room temperature raise to 22 ºC and so is more that 1 ºC over target temperature. |

-> Use Case Heating Control

Scenario for Use Case setValues  
The user sets all values to default

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The user sets StandardPresenceTemp to default</td>
</tr>
<tr>
<td>2.</td>
<td>The user sets StandardAwayTemp to default</td>
</tr>
<tr>
<td>3.</td>
<td>The user sets StandardInsideTime to default</td>
</tr>
<tr>
<td>4.</td>
<td>The user sets StandardAwayTime to default</td>
</tr>
<tr>
<td>5.</td>
<td>The user sets StandardHeatingTime to default</td>
</tr>
</tbody>
</table>

Scenario for Use Case heatingControl  
Target Temperature of a room is reached
1. **Presence switch**: closed  
**Outside temperature**: 14 ºC  
**Window**: closed  
**Rain sensor**: open  
**Target temperature**: 20 ºC (StandardPresenceTemp)  
**Room temperature**: 18 ºC

2. Room temperature reach 19 ºC

3. Stop temperature has been reached, since room temperature is just 1 ºC below target temperature.  

4. Required new boiler temperature is computed

---

**Scenario for Use Case Heating Control**  
*The room must be heated up*

**Scenario 10**

1. **Presence switch**: closed  
**Outside temperature**: 23 ºC  
**Window**: closed  
**Rain sensor**: open  
**Target temperature**: 20 ºC (StandardPresenceTemp)  
**Room temperature**: 19 ºC

2. Room temperature drops to 18 ºC

3. Heating temperature gets calculated because room temperature has dropped below 1 ºC under target temperature.

4. Heater temperature is set to the necessary value

5. Required new boiler temperature is computed

---

**Scenario for Use Case Heating Control**  
*Room temperature is above target temperature*

**Scenario 11**

1. **Presence switch**: closed  
**Outside temperature**: 23 ºC  
**Window**: closed  
**Rain sensor**: open  
**Target temperature**: 20 ºC (StandardPresenceTemp)  
**Room temperature**: 21 ºC

2. Room temperature raises to 22 ºC

3. As the temperature of the room is more than 1 ºC over target temperature, heater temperature is set to 0 ºC

4. Required new boiler temperature is computed

5. The window is not opened to cool down room temperature.

---

1.2.3 **Developer’s Requirements/Specifications**

Here UML diagrams that describe requirements follow. These include a class diagram with Data-Dictionary, Sequence Diagrams and State Diagrams.

1.2.3.1 **UML class diagrams**

In class diagrams the static structure of the system is described, composed of the classes and their relationships. It is further detailed in the Data-Dictionary.

1.2.3.1.1 **Package structure**

This diagram shows the partitioning of the system into subsystems (Packages). Different packages all have their own classes. In a package methods and attributes of classes aren’t defined.
Description of each package

Package: gui
This package includes classes that build up functionalities of the user interface.

Package: stubs
Includes interfaces and classes that model the hardware at an abstract level

Package: mystubs
Is composed of classes that represent real hardware. Currently contains only simplified models.

Package: activity
This package includes singleton classes that set up parameters and actions for the rest of the system.

Package: hcs
This package contains the main class in the heating control system, and three other subpackages. These are shown as separate packages for proper visualization.

Package: hcs.model
Includes the classes that contribute in controlling temperature, and also provides classes that logically model the various parts of the system.
Package: hcs.interfaces
Includes abstract classes that capture the common behavior of parts of the heating control system.

Package: hcs.settings
Is composed of classes that handle the global settings of the system.

1.2.3.1.2 Temperature control
This diagram shows the main classes that participate in controlling temperature. For clarity the classes that only contribute to the hardware wrapper are omitted, as well as the classes that handle the global system settings.

Class diagram

1.2.3.1.3 Hardware wrapper
This diagram shows all the classes that represent the logical and abstract view of the hardware
1.2.3.1.4 Event model
This diagram shows how the generation and propagation of events is handled.
The `Clock` class generates an event every second; the `update()` method of every class that inherits from the `ClockObserver` is called as a result of that event.

### 1.2.3.1.5 User interface
This diagram shows a simplified view of the user interface subsystem.

**Class diagram**
1.2.3.1.6 Initialization
This diagram shows the classes involved in the initialization of the entire system.
### 1.2.3.2 Data-dictionary

Into the data-dictionary are described all the classes of the UML-analysis together with their attributes, methods and relationships.

<table>
<thead>
<tr>
<th>Object to be described</th>
<th>Attribute, Method, Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AwayManagementStrategy</td>
<td>Controls the room management strategy when the user is away or the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>window is manually opened</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTemperatureToFollow() Returns the target temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>roomIsOK() Manages the room when the temperature is correct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>roomIsTooCool() Manages the room when the temperature is below the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
</tr>
<tr>
<td></td>
<td>roomIsTooWarm() Manages the room when the temperature is above the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
</tr>
<tr>
<td>Boiler</td>
<td>Models the logical behavior of the central boiler.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>currentTemperature The current measured water temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>targetTemperature The desired water temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCurrentTemperature() Returns the current measured water temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setTargetTemperature() Sets the desired water temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTargetTemperature() Returns the desired water temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>update() Computes the next logical state of the boiler</td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td>Measures wall-clock time and distributes events at fixed intervals.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>instance The current class instance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getInstance() Returns the current class instance, or generates one if</td>
<td></td>
</tr>
<tr>
<td></td>
<td>it does not exist</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tic() Executes one clock tick, notifying all observers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getPriority() Returns the priority of a given observer</td>
<td></td>
</tr>
<tr>
<td>ClockObserver</td>
<td>This interface describes all classes that must autonomously update</td>
<td></td>
</tr>
<tr>
<td></td>
<td>their status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>update() Computes the next state of the object</td>
<td></td>
</tr>
<tr>
<td>CommandMenuBar</td>
<td>The application menu bar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getMainFrame() Navigates to the containing MainFrame</td>
<td></td>
</tr>
<tr>
<td>CRoom</td>
<td>This class is a simplified model of the physical room. It also includes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a simple thermal model.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>temperature The room temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>heater The current heater temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>windowOpen The open/closed state of the window</td>
<td></td>
</tr>
<tr>
<td></td>
<td>windowTilted The tilted/not tilted state of the window</td>
<td></td>
</tr>
<tr>
<td></td>
<td>notPresence The presence state of the user</td>
<td></td>
</tr>
<tr>
<td></td>
<td>closedWindowSwitchOpen() Returns the state of the open window switch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTemperature() Returns the measured temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>presenceSwitchOpen() Returns the state of the presence switch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setHeaterTarget() Sets the target heater temperature</td>
<td></td>
</tr>
<tr>
<td></td>
<td>swingWindow() Opens the window if it is closed, or vice versa.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tiltWindow() Tils the window if it is closed, or vice versa.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tiltedWindowSwitchOpen() Returns the state of the tilted window switch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>update() Computes the next state of the room</td>
<td></td>
</tr>
<tr>
<td>DefaultHouseSettings</td>
<td>Represents the defaults the user can use for values.</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>standardPresenceTemp</td>
<td>Default temperature when the user is in the room</td>
<td></td>
</tr>
<tr>
<td>standardAwayTemp</td>
<td>Default temperature when the room is empty</td>
<td></td>
</tr>
<tr>
<td>standardHeatingTime</td>
<td>Default time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp</td>
<td></td>
</tr>
<tr>
<td>standardAwayTime</td>
<td>Default time in minutes that should pass, after the user has left the room, before the room is managed as empty</td>
<td></td>
</tr>
<tr>
<td>standardInsideTime</td>
<td>Default time in minutes that should pass, after the user has entered the room, before the room is managed as non empty</td>
<td></td>
</tr>
<tr>
<td>roomNames</td>
<td>The names of all the rooms</td>
<td></td>
</tr>
<tr>
<td>sqm</td>
<td>The areas of all the rooms in square meters</td>
<td></td>
</tr>
<tr>
<td>getRoomNames()</td>
<td>Returns the names of all the rooms</td>
<td></td>
</tr>
<tr>
<td>getSqm()</td>
<td>Returns the areas of all the rooms in square meters</td>
<td></td>
</tr>
<tr>
<td>StandardPresenceTemp()</td>
<td>Returns the default temperature when the user is in the room</td>
<td></td>
</tr>
<tr>
<td>StandardAwayTemp()</td>
<td>Returns the default temperature when the room is empty</td>
<td></td>
</tr>
<tr>
<td>StandardHeatingTime()</td>
<td>Returns the default time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp</td>
<td></td>
</tr>
<tr>
<td>StandardAwayTime()</td>
<td>Returns the default time in minutes that should pass, after the user has left the room, before the room is managed as empty</td>
<td></td>
</tr>
<tr>
<td>StandardInsideTime()</td>
<td>Returns the default time in minutes that should pass, after the user has entered the room, before the room is managed as non empty</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Env</th>
<th>This class provides a simplified model of the physical environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>The outside temperature</td>
</tr>
<tr>
<td>rain</td>
<td>It is raining</td>
</tr>
<tr>
<td>seconds</td>
<td>The time in seconds elapsed since application start</td>
</tr>
<tr>
<td>getTemperature()</td>
<td>Returns the measured temperature</td>
</tr>
<tr>
<td>isRainSwitchClosed()</td>
<td>Returns the state of the rain switch</td>
</tr>
<tr>
<td>update()</td>
<td>Computes the next state of the environment</td>
</tr>
<tr>
<td>getSecondsSinceStart()</td>
<td>Returns the time in seconds elapsed since application start</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment</th>
<th>Represents the logical model of the environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td>The measured outside temperature</td>
</tr>
<tr>
<td>isRaining</td>
<td>The logical indication of rain, as given by the switch</td>
</tr>
<tr>
<td>seconds</td>
<td>The measured time elapsed since application start, in seconds</td>
</tr>
<tr>
<td>getTemperature()</td>
<td>Returns the measured outside temperature</td>
</tr>
<tr>
<td>isRainSwitchClosed()</td>
<td>Returns the state of the rain switch</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>update()</code></td>
<td>Computes the next logical state of the environment</td>
</tr>
<tr>
<td><code>getSecondsSinceStart()</code></td>
<td>Returns the measured time elapsed since application start, in seconds</td>
</tr>
</tbody>
</table>

**EnvironmentMonitor**

- **tempLabel**
  - The screen label for the measured temperature
- **rainLabel**
  - The screen label for the rain switch state
- **boilerLabel**
  - The screen label for the current measured boiler temperature
- **boilerTargetLabel**
  - The screen label for the desired boiler temperature
- **update()**
  - Updates the environment display

**FileMenu**

- **ExitOnClose**
  - A private action listener that makes the application exit when the Close option is selected. Semantically similar to a method
- **ShowLogger**
  - A private action listener that makes the application log appear on the screen when the Show Logger option is selected. Semantically similar to a method

**GuiUtilities**

- **fitMenuItem()**
  - Adds a new item to an existing menu
- **fitNewLabel()**
  - Adds a new component to an existing interface window

**HcsGuiMenu**

- **getCommandMenuBar()**
  - Navigates to the containing CommandMenuBar

**HelpMenu**

- **aboutMessage**
  - The message to be displayed when the About option is selected
- **ShowAbout**
  - A private action listener that makes the about message appear on the screen when the About option is selected. Semantically similar to a method

**HouseController**

- **computeBoilerTemperature()**
  - Computes the desired water temperature in the boiler
- **getEnvironment()**
  - Navigates to the logical model of the environment
- **getClock()**
  - Navigates to the Clock
- **iterator()**
  - Returns an iterator to the contained Rooms
- **getNumberOfRooms()**
  - Returns the number of rooms
- **getHouseSettings()**
  - Navigates to the current global settings
- **update()**
  - Computes the next logical state of the system
- **addBoilerObserver()**
  - Adds an observer to the Boiler
<table>
<thead>
<tr>
<th>Method/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteBoilerObserver()</td>
<td>Removes an object from the list of Boiler observers</td>
</tr>
<tr>
<td>HouseSettings</td>
<td>A factory that returns the global system settings (i.e. not the settings of any single room).  <strong>settings</strong> The name of the XML file containing the global system settings  <strong>getHouseSettings()</strong> Returns an instance of either XMLSettings or, if the settings file could not be read, of DefaultHouseSettings</td>
</tr>
<tr>
<td>IHouseSettings</td>
<td>The interface that defines the global system settings.  <strong>getRoomNames()</strong> Returns the names of all the rooms  <strong>getSqm()</strong> Returns the areas of all the rooms in square meters  <strong>StandardPresenceTemp()</strong> Returns the desired temperature when the user is in the room  <strong>StandardAwayTemp()</strong> Returns the desired temperature when the room is empty  <strong>StandardHeatingTime()</strong> Returns the desired time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp  <strong>StandardAwayTime()</strong> Returns the desired time in minutes that should pass, after the user has left the room, before the room is managed as empty  <strong>StandardInsideTime()</strong> Returns the desired time in minutes that should pass, after the user has entered the room, before the room is managed as non empty</td>
</tr>
<tr>
<td>InvalidTimeException</td>
<td>This exception is raised when the user specifies a StandardHeatingTime less than 5 minutes longer than StandardInsideTime.  <strong>requestedTime</strong> The StandardHeatingTime requested by the user  <strong>standardInsideTime</strong> The current StandardInsideTime  <strong>serialVersionUID</strong> The unique numerical identifier of the class version  <strong>getRequestedTime()</strong> Returns the StandardHeatingTime requested by the user  <strong>getStandardInsideTime()</strong> Returns the current StandardInsideTime</td>
</tr>
<tr>
<td>IPhysicalBoiler</td>
<td>The interface that defines the behavior of a physical boiler.  <strong>getTargetTemperature()</strong> Returns the desired water temperature  <strong>setTargetTemperature()</strong> Sets the desired water temperature  <strong>getCurrentTemperature()</strong> Returns the current measured water temperature</td>
</tr>
<tr>
<td>IPhysicalEnvironment</td>
<td>The interface that defines the system behavior of the physical environment.  <strong>isRainingSwitchClosed()</strong> Returns the state of the rain switch  <strong>getTemperature()</strong> Returns the measured temperature  <strong>getSecondsSinceStart()</strong> Returns the time in seconds elapsed since application start</td>
</tr>
<tr>
<td>IPhysicalRoom</td>
<td>The interface that defines the behavior of a physical room.  <strong>presenceSwitchOpen()</strong> Returns the state of the presence switch  <strong>tiltedWindowSwitchOpen()</strong> Returns the state of the tilted window switch  <strong>closedWindowSwitchOpen()</strong> Returns the state of the open window switch  <strong>swingWindow()</strong> Opens the window if it is closed, or vice versa. Does not care of the tilt state</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>tiltWindow()</code></td>
<td>Tilts the window if it is closed, or vice versa. Does not care of the open state.</td>
</tr>
<tr>
<td><code>getTemperature()</code></td>
<td>Returns the measured temperature.</td>
</tr>
<tr>
<td><code>setHeaterTarget()</code></td>
<td>Sets the target heater temperature.</td>
</tr>
<tr>
<td><code>getHeaterTarget()</code></td>
<td>Returns the target heater temperature.</td>
</tr>
<tr>
<td><code>getHeaterCurrent()</code></td>
<td>Returns the current heater temperature.</td>
</tr>
</tbody>
</table>

**LogViewer**

- **TextArea**
  - Provides the user a view of the system log.
  - The text area used to hold log information.
- **setVisible()**
  - Makes the log information visible on screen.
- **update()**
  - Updates the log information.
- **serializeBoiler()**
  - Transforms current Boiler information in string format.
- **serializeSettings()**
  - Transforms current RoomSettings information in string format.
- **serializeRoom()**
  - Transforms current Room information in string format.
- **serializeEnvironment()**
  - Transforms current Environment information in string format.

**Main**

- The starting point for code execution.
- `main()`
- `getMainFrame()`
  - Navigates to the application main interface component.

**MainFrame**

- This class integrates the user interface components.
- `getRoomMonitors()`
  - Navigates to the list of enclosed Room Monitors.
- `getEnvironmentMonitor()`
  - Navigates to the enclosed Environment Monitor.
- `getHouseController()`
  - Navigates to the HouseController.

**PhysBoiler**

- This class is a simplified model of a physical boiler.
- `currentTemperature`
  - The current water temperature.
- `targetTemperature`
  - The desired water temperature.
- `getCurrentTemperature()`
  - Returns the desired water temperature.
- `setTargetTemperature()`
  - Sets the desired water temperature.
- `update()`
  - Computes the next boiler state.
- `getTargetTemperature()`
  - Returns the current measured water temperature.

**PhysicalFactory**

- Factory class that returns instances of physical models.
- `getPhysicalRoom()`
  - Returns an existing instance of a CRoom.
- `getPhysicalEnvironment()`
  - Returns the existing instance of Env.
- `getPhysicalBoiler()`
  - Returns the existing instance of PhysBoiler.

**PresenceManagementStrategy**

- Controls the room management strategy when the user is inside.
- `applyStrategy()`
  - Manages the room supposing that the user is inside.
- `getTemperatureToFollow()`
  - Returns the target temperature.
- `roomIsOK()`
  - Manages the room when the temperature is correct.
- `roomIsTooCool()`
  - Manages the room when the temperature is below the target.
- `roomIsTooWarm()`
  - Manages the room when the temperature is above the target.

**PriorityObservable**

- Abstract class that defines the behavior of a class that is observable by other classes.
- `orderedQueue`
  - The queue of all observers, ordered by their priority.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPriority()</td>
<td>Returns the priority of a given observer</td>
</tr>
<tr>
<td>addObserver()</td>
<td>Adds an observer to the ordered queue</td>
</tr>
<tr>
<td>getAllObservers()</td>
<td>Returns a list of all observers, in priority order</td>
</tr>
<tr>
<td>deleteObserver()</td>
<td>Removes an observer from the ordered queue</td>
</tr>
<tr>
<td>notifyObservers()</td>
<td>Calls the update() method of all the observers in priority order</td>
</tr>
<tr>
<td>deleteObservers()</td>
<td>Removes all observers from the queue</td>
</tr>
<tr>
<td>countObservers()</td>
<td>Returns the current number of observers</td>
</tr>
</tbody>
</table>

**Room**

This class defines the logical model of a room. It also handles the relative switches and actuators.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The room name</td>
</tr>
<tr>
<td>sqm</td>
<td>The room area in square meters</td>
</tr>
<tr>
<td>temperature</td>
<td>The measured room temperature</td>
</tr>
<tr>
<td>windowState</td>
<td>The last remembered window state</td>
</tr>
<tr>
<td>presence</td>
<td>The last remembered user presence state</td>
</tr>
<tr>
<td>manualWindow</td>
<td>The window management state. True if the window should not be controlled by</td>
</tr>
<tr>
<td></td>
<td>the system, false otherwise</td>
</tr>
<tr>
<td>heaterTarget</td>
<td>The desired heater temperature</td>
</tr>
<tr>
<td>heaterCurrent</td>
<td>The current measured heater temperature</td>
</tr>
<tr>
<td>lastUpdateTime</td>
<td>The last time the logical state of the room has been recomputed</td>
</tr>
<tr>
<td>inTime</td>
<td>Elapsed time since the user last entered the room, if he is still in</td>
</tr>
<tr>
<td>outTime</td>
<td>Elapsed time since the user last left the room, if he is still out</td>
</tr>
<tr>
<td>decodeWindowState()</td>
<td>Returns a composite window state information by reading the switch states</td>
</tr>
<tr>
<td>getWindowStatus()</td>
<td>Returns the last remembered window state</td>
</tr>
<tr>
<td>setWindowStatus()</td>
<td>Operates the window</td>
</tr>
<tr>
<td>getTemperature()</td>
<td>Returns the measured temperature</td>
</tr>
<tr>
<td>poll()</td>
<td>Computes the next logical status of the room</td>
</tr>
<tr>
<td>getName()</td>
<td>Returns the room name</td>
</tr>
<tr>
<td>getSqm()</td>
<td>Returns the room area, in square meters</td>
</tr>
<tr>
<td>SomebodyHere()</td>
<td>Returns the last remembered user presence state</td>
</tr>
<tr>
<td>getHeaterTarget()</td>
<td>Returns the target heater temperature</td>
</tr>
<tr>
<td>getHeaterCurrent()</td>
<td>Returns the current heater temperature</td>
</tr>
<tr>
<td>setHeaterTarget()</td>
<td>Sets the target heater temperature</td>
</tr>
<tr>
<td>getSettings()</td>
<td>Navigates to the RoomSettings for this Room</td>
</tr>
<tr>
<td>getHouseController()</td>
<td>Navigates to the enclosing HouseController</td>
</tr>
</tbody>
</table>

**RoomManagementStrategy**

An abstract class that defines a generic management strategy.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applyStrategy()</td>
<td>Manages the room in the general case</td>
</tr>
<tr>
<td>getTemperatureToFollow()</td>
<td>Returns the target temperature</td>
</tr>
<tr>
<td>roomIsOK()</td>
<td>Manages the room when the temperature is correct</td>
</tr>
<tr>
<td>roomIsTooCool()</td>
<td>Manages the room when the temperature is below the target</td>
</tr>
<tr>
<td>roomIsTooWarm()</td>
<td>Manages the room when the temperature is above the target</td>
</tr>
</tbody>
</table>

**RoomMonitor**

Observes the state of a Room, and reports it to the user.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td>The temperature to display</td>
</tr>
<tr>
<td>windowStatus</td>
<td>The window status to display</td>
</tr>
<tr>
<td>somebodyHere</td>
<td>The user presence status</td>
</tr>
<tr>
<td>heater</td>
<td>The current heater temperature</td>
</tr>
<tr>
<td>heaterTarget</td>
<td>The desired heater temperature</td>
</tr>
<tr>
<td>selected</td>
<td>The selection state of the RoomMonitor. True if the user has selected the room</td>
</tr>
<tr>
<td>initLayout()</td>
<td>Initializes the display of the room status</td>
</tr>
<tr>
<td>update()</td>
<td>Updates the displayed room information</td>
</tr>
<tr>
<td>tempToString()</td>
<td>Converts the heater temperature to a string</td>
</tr>
<tr>
<td>isSelected()</td>
<td>Returns true if the RoomMonitor has been selected by the user</td>
</tr>
<tr>
<td>getFrame()</td>
<td>Navigates to the enclosing MainFrame</td>
</tr>
<tr>
<td>getRoom()</td>
<td>Navigates to the relative Room</td>
</tr>
</tbody>
</table>

**RoomSettings**

This class contains the settings for a Room.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardPresenceTemp</td>
<td>Desired temperature when the user is in the room</td>
</tr>
<tr>
<td>StandardAwayTemp</td>
<td>Desired temperature when the room is empty</td>
</tr>
<tr>
<td>StandardHeatingTime</td>
<td>Desired time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp</td>
</tr>
<tr>
<td>StandardAwayTime</td>
<td>Desired time in minutes that should pass, after the user has left the room, before the room is managed as empty</td>
</tr>
<tr>
<td>StandardInsideTime</td>
<td>Desired time in minutes that should pass, after the user has entered the room, before the room is managed as non empty</td>
</tr>
<tr>
<td>revertToDefault()</td>
<td>Resets all settings to their default values</td>
</tr>
<tr>
<td>getStandardPresenceTemp()</td>
<td>Returns the desired temperature when the user is in the room</td>
</tr>
<tr>
<td>setStandardPresenceTemp()</td>
<td>Sets the desired temperature when the user is in the room</td>
</tr>
<tr>
<td>getStandardAwayTemp()</td>
<td>Returns the desired temperature when the room is empty</td>
</tr>
<tr>
<td>setStandardAwayTemp()</td>
<td>Sets the desired temperature when the room is empty</td>
</tr>
<tr>
<td>getStandardHeatingTime()</td>
<td>Returns the desired time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp</td>
</tr>
<tr>
<td>setStandardHeatingTime()</td>
<td>Sets the desired time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp</td>
</tr>
<tr>
<td>getStandardAwayTime()</td>
<td>Returns the desired time in minutes that should pass, after the user has left the room, before the room is managed as empty</td>
</tr>
<tr>
<td>setStandardAwayTime()</td>
<td>Sets the desired time in minutes that should pass, after the user has left the room, before the room is managed as empty</td>
</tr>
<tr>
<td>getStandardInsideTime()</td>
<td>Returns the desired time in minutes that should pass, after the user has left the room, before the room is managed as non empty</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setStandardInsideTime()</td>
<td>Sets the desired time in minutes that should pass, after the user has entered the room, before the room is managed as non empty</td>
</tr>
<tr>
<td>toString()</td>
<td>Converts the setting values to a HTML string</td>
</tr>
<tr>
<td>getRoom()</td>
<td>Navigates to the related Room</td>
</tr>
</tbody>
</table>

**SetClockActivity**

This class is used to set up the Clock, start it and stop it.

- **t**
  - The actual timer

- **delay**
  - The delay between one timer tick and the next

- **tic**
  - A private action listener that calls the tic() method of Clock at every tick of the timer. Semantically similar to a method

- **start()**
  - Starts a new timer. If a running timer already exists, stops it first

- **stop()**
  - Stops the running timer

- **getDelay()**
  - Returns the programmed timer delay

- **setDelay()**
  - Programs the timer delay

**SetRoomParametersActivity**

This class is used to set room parameters and get the defaults.

- **setRoomParameter()**
  - Sets a single room parameter

- **getDefaultParameter()**
  - Returns the default for a single room parameter

**SetRoomParametersDialog**

This class provides the user interface to set room parameters.

- **labels**
  - The names of the parameters for a given room

- **map**
  - The correspondence map between parameters and display components

- **setRoomParameter()**
  - Sets a single room parameter

- **onCancel**
  - A private action listener that makes the parameter dialog disappear without consequences when the Cancel option is selected. Semantically similar to a method

- **onSet**
  - A private action listener that makes the parameter dialog disappear and sets all the options to new values when the Set option is selected. Semantically similar to a method

- **buildButton()**
  - Sets up the buttons for the dialog box

**StateObservable**

This abstract class defines the interface to a class whose state is observable.

- **addObserver()**
  - Adds an observer to the object

- **deleteObserver()**
  - Removes an object from the set of observers

**StateObserver**

This interface defines the behavior of a class that observes the state of another class.

- **update()**
  - Computes the next state of the object

**ToolsMenu**

Provides the user with the options in the Tools Menu.

- **start**
  - The Start option of the menu

- **stop**
  - The Stop option of the menu

- **StopTimer**
  - A private action listener that makes the application stop its timer when the Stop option is selected. Semantically similar to a method
<table>
<thead>
<tr>
<th>Action Listener</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartTimer</td>
<td>A private action listener that makes the application start a new timer when the Start option is selected. Semantically similar to a method.</td>
</tr>
<tr>
<td>SetClock</td>
<td>A private action listener that makes the application display a dialog box when the Set Clock option is selected and then program a new timer delay when the Ok option of the box is selected. Semantically similar to a method.</td>
</tr>
<tr>
<td>SelectAllMonitors</td>
<td>A private action listener that selects all rooms when the Select All option is selected. Semantically similar to a method.</td>
</tr>
<tr>
<td>DeSelectAllMonitors</td>
<td>A private action listener that deselects all rooms when the Deselect All option is selected. Semantically similar to a method.</td>
</tr>
<tr>
<td>buildSelector()</td>
<td>Returns an action listener that handles single selections.</td>
</tr>
<tr>
<td>SetRoomParameter</td>
<td>A private action listener that makes the application exit when the Close option is selected.</td>
</tr>
<tr>
<td>getCommandMenuBar()</td>
<td>Navigates to the enclosing CommandMenuBar.</td>
</tr>
</tbody>
</table>

**XMLSettings**

- Represents the setting values derived from the XML file.
- `standardPresenceTemp` Desired temperature when the user is in the room
- `standardAwayTemp` Desired temperature when the room is empty
- `standardHeatingTime` Desired time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp
- `standardAwayTime` Desired time in minutes that should pass, after the user has left the room, before the room is managed as empty
- `standardInsideTime` Desired time in minutes that should pass, after the user has entered the room, before the room is managed as non empty
- `roomNames` The names of all the rooms
- `sqm` The areas of all the rooms in square meters
- `getRoomNames()` Returns the names of all the rooms
- `getSqm()` Returns the areas of all the rooms in square meters
- `StandardPresenceTemp()` Returns the desired temperature when the user is in the room
- `StandardAwayTemp()` Returns the desired temperature when the room is empty
- `StandardHeatingTime()` Returns the desired time in minutes, after an user has entered the room, within which the room should reach the StandardPresenceTemp
- `StandardAwayTime()` Returns the desired time in minutes that should pass, after the user has left the room, before the room is managed as empty
| StandardInsideTime() | Returns the desired time in minutes that should pass, after the user has entered the room, before the room is managed as non empty |

Table 5
### 1.2.3.3 UML Sequence Diagrams

Use Cases of this system are detailed below with sequence diagrams.

#### 1.2.3.3.1 Sequence diagram for Use Case Presence

Sequence diagram for scenario 3:

1. Clock
2. HouseController
3. Room
4. AwayManagementStrategy
5. PresenceManagementStrategy

Sequence for scenario 3:

1. update()
2. poll()
3. applyStrategy()
4. update()
5. poll()
6. applyStrategy()

If `outTime > StandardAwayTime`

- user absent

Sequence for scenario 2:

1. Clock
2. HouseController
3. Room
4. AwayManagementStrategy

Sequence for scenario 2:

1. update()
2. poll()
3. applyStrategy()
4. update()
5. poll()
6. applyStrategy()

If `inTime < StandardInsideTime`

- user present
- user absent
Sequence diagram for scenario 4:

Sequence diagram for scenario 5:

1.2.3.2 Sequence diagrams for Use Case WindowPosition

Sequence diagram for scenario 5:
1.2.3.3  Sequence Diagram for Use Case HeatingControl

Sequence diagram for scenario 10:

```
Clock -> HouseController: update()
HouseController -> Room: poll()
Room -> PresenceManagementStrategy: applyStrategy()
PresenceManagementStrategy -> Room: setHeaterTarget()
Room -> Room: setWindowStatus()
Room -> Boiler: computeBoilerTemperature()
Boiler -> Room: setTargetTemperature()
Room -> Room: update()
Room -> HouseController: poll()
HouseController -> PresenceManagementStrategy: applyStrategy()
PresenceManagementStrategy -> Room: setHeaterTarget()
Room -> Room: setWindowStatus()
Room -> Room: computeBoilerTemperature()
Room -> Boiler: setTargetTemperature()
```

```
max boiler temperature
```
Sequence diagram for scenario 11:

1. update()
2. poll()
3. applyStrategy()
4. setHeaterTarget()
5. setHeaterTarget()
6. setWindowStatus()
7. computeBoilerTemperature()
8. setTargetTemperature()
9. update()
10. poll()
11. applyStrategy()
12. setHeaterTarget()
13. setHeaterTarget()
14. setWindowStatus()
15. computeBoilerTemperature()
16. setTargetTemperature()
Sequence diagram for scenario 9:

1: update() -> Clock
2: poll() -> HouseController
3: applyStrategy() -> PresenceManagementStrategy
4: setHeaterTarget() -> Room
5: setHeaterTarget() -> CRoom
6: setWindowStatus() -> Room
7: computeBoilerTemperature() -> Room
8: setTargetTemperature() -> Boiler
9: update() -> Clock
10: poll() -> HouseController
11: applyStrategy() -> PresenceManagementStrategy
12: setHeaterTarget() -> Room
13: setHeaterTarget() -> CRoom
14: setWindowStatus() -> Room
15: computeBoilerTemperature() -> Room
16: setTargetTemperature() -> Boiler
1.2.3.4 Sequence Diagrams for Use Case RainMonitoring
Sequence diagram for scenario 6:
1.2.3.5  Sequence diagram for Use Case TemperatureTooHigh
Sequence diagram for scenario 7:

```
/ : Clock

1. update()
2. poll()
3. applyStrategy()
4. setWindowStatus()
```

1.2.3.6  Sequence diagrams for Use Case ValuesSetting
Sequence diagram for scenario 1:

```
/ : SetRoomParametersDialog
/ : SetRoomParametersActivity
/ : RoomSettings

1. onSet()
2. setRoomParameter()
3. setRoomParameter()
4. setStandardInsideTime()

5. onSet()
6. setRoomParameter()
7. setRoomParameter()
8. setStandardHeatingTime()
9. onSet()
10. setRoomParameter()
11. setRoomParameter()
12. setStandardHeatingTime()

13. onSet()
14. setRoomParameter()
15. setRoomParameter()
16. setStandardAwayTemp()

17. onSet()
18. setRoomParameter()
19. setRoomParameter()
20. setStandardAwayTime()

21. onSet()
22. setRoomParameter()
23. setRoomParameter()
24. setStandardPresenceTemp()
```
Sequence diagram for scenario 8:

1. onSet()
2. setRoomParameter()
3. setRoomParameter()
4. setStandardInsideTime()
5. onSet()
6. setRoomParameter()
7. setRoomParameter()
8. setStandardHeatingTime()
9. onSet()
10. setRoomParameter()
11. setRoomParameter()
12. setStandardAwayTemp()
13. onSet()
14. setRoomParameter()
15. setRoomParameter()
16. setStandardAwayTime()
17. onSet()
18. setRoomParameter()
19. setRoomParameter()
20. setStandardPresenceTemp()
1.2.3.4  UML state diagrams
State diagrams depict dynamic behavior of classes. The behavior of each class will be depicted below with one or more specific state diagrams. Methods calls are unaccounted in the state diagrams, as they serve only for setting or getting values.

1.2.3.4.1  Manual Window (class Room)
This state diagram refers to the manual or automatic operation of the window. In manual state the system does not attempt to operate the window, whereas in notManual state the system may open, close or tilt the window as needed.

![State Diagram for Manual Window](image)

Do-activity for State: notManual
windowState := decodeWindowState()
presence := CRoom.presenceSwitchOpen()

Do-activity for State: manual
windowState := decodeWindowState()
presence := CRoom.presenceSwitchOpen()

1.2.3.4.2  Management Strategy (class Room)
This diagram depicts the choice of the temperature management strategy to use.
Do-activity for State: PresenceStrategy
presence := Croom.presenceSwitchOpen()

Do-activity for State: AwayStrategy
presence := Croom.presenceSwitchOpen()

1.2.3.4.3 Management strategy (class PresenceManagementStrategy)
Here the management strategy used when the user is present is depicted.

Do-activity for State: roomIsTooCool
temperature := Room.getTemperature()

Do-activity for State: roomIsOK
temperature := Room.getTemperature()
Do-activity for State: roomIsTooWarm
temperature := Room.getTemperature()

1.2.3.4.4 Management strategy (class AwayManagementStrategy)
Here is described the management strategy used when the user is not present or has manually opened the window.

Do-activity for State: roomIsTooCool
temperature := Room.getTemperature()

Do-activity for State: roomIsOK
temperature := Room.getTemperature()

Do-activity for State: roomIsTooWarm
temperature := Room.getTemperature()
1.2.3.4.5  Window operation (class Room)

This diagram details the flow of computation that leads to system use of the window actuators.

```
Do-activity for State: roomIsTooCool
  temperature := Room.getTemperature()
  environmentTemp := Environment.getTemperature()

Do-activity for State: roomIsOK
  temperature := Room.getTemperature()
  environmentTemp := Environment.getTemperature()

Do-activity for State: roomIsTooWarm
  temperature := Room.getTemperature()
  environmentTemp := Environment.getTemperature()
```
Do-activity for State: winDesiredOpen
isRaining := Environment.isRainSwitchClosed()

Do-activity for State: winDesiredClosed
none

Do-activity for State: winDesiredTilted
none

Do-activity for State: operateWindow
setWindowStatus()

1.2.4 Traceability matrix: user requirements <-> class diagrams
The traceability matrix reports which classes user requirements have been mapped to. Inside it you’ll find the classes, but not instances.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp-UR-F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F6</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F7</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F8</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F9</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F10</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F13</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F14</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F15</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F16</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F17</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temp-UR-F18</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UR-Inv 1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UR-Inv 2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Software Engineering Quiz 1

HAIDER MUHI ABBAS
Control and Systems engineering department
University of technology
Q: what is the difference between safety critical and mission critical software.

Q: True or False(correct the wrong one):
1) Airplane is probably the most complex human artifacts.
2) Software is produced.
3) Solo programming is a software which consider only functional properties.
4) Reliability is possible to be domain