NExt ApplicationS of Quantum Computing

D6.9: QRBS software specifications

Document Properties

Contract Number	951821
Contractual Deadline	31-oct-2022
Dissemination Level	Public
Nature	Report
Editors	Vicente Moret-Bonillo, UDC Andrés Gómez Tato, Cesga
Authors	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC Diego Álvarez-Estévez, UDC
Reviewers	Vedran Dunjko, Leiden Univ. Mohamed Hibti, EDF
Date	21-oct-2022
Keywords	software specifications, static design, dynamic design, testing
Status	Reviewed
Release	1.1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 951821

History of Changes

Release	Date	Author, Organisation	Description of Changes
1.0	11/10/2022	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC	First version.
1.1	21/10/2022	Vicente Moret-Bonillo, UDC Samuel Magaz-Romero, UDC Eduardo Mosqueira-Rey, UDC	Post internal review.

Table of Contents

1. Executive Summary

This report is the third deliverable of Task 6.2 – Quantum Rule-Based Systems (QRBS) for breast cancer detection of the NEASQC project. The document presents the work carried out so far, and is complementary to the other deliverables of this task.

The report begins with an introduction into the software specification process, presenting the necessary definitions for the reader to comprehend the work carried out. Following those concepts come the several specifications obtained, including both static and dynamic design, as well as the implementations specifications for this use case.

Following that, focus is on the testing phase, in order to prove that the specification obtained is coherent regards both itself and the previous work.

To close the report, the conclusions obtained during the development of the work carried out are presented and some ideas for future work to be included in upcoming deliverables are shown.

2. Context

2.1. Project

In the context of this project, this document describes the specification of the software regarding the development of the framework for Quantum Rule-Based Systems (QRBSs). This specification has been approached from a software engineering perspective, specifically following the strategy of the Unified Process of software development (Kruchten, 2004).

The software specification is a widely accepted and established procedure in the field of software engineering, as it ties together the analysis of requirements with the implementation of the software itself. The design obtained during the process of specification should be based on abstraction, coupling and cohesion, decomposition and modularization, encapsulation of information and completeness (Bourque & Fairley, 2014).

This approach is based on software engineering, but some parts of the specification are related with quantum computing and used to interact with quantum computers and their simulators.

With this specification we provide the description and models on which the implementation of the software will be based, in order to follow the software engineering process and to be able to address the problems that may arise on future work.

2.2. Work package

In the context of the Work Package 6 – "Symbolic AI and graph algorithmics", this document illustrates one of the steps (the software specification) that must be followed in order to achieve the final result of our task, the development of the framework to represent Quantum Rule-Based Systems.

In our previous deliverable, "D6.5 Quantum Rule-Based System" (Moret-Bonillo et al., 2021), we analyzed the requirements, in the form of needs, features, use cases and test cases, that the final framework for Quantum Rule-Based Systems must contain. That analysis allowed us to establish the functionalities that must be provided for our use case. Now, those definitions support the work carried out on this document, since they will serve as the basis on which we base the specification of the different components that make up the software.

The software specification, according to the knowledge model for breast cancer diagnosis, screening and treatment of Invasive Ductal Carcinoma (IDC) (see appendix A) is a core step in the process of software development since it provides the design on which the implementation of the software itself will be based. This design specifies the characteristics and functionalities of each component of the system, as well as how they interact with each other. Therefore, it allows us to understand the architecture and structure of the system as a whole and the different flows that take place with each possible procedure (Pressman, 2005).

Future deliverables are complementary to this one, since they delve into the specifics of the implementation of the system designed here and its application for breast cancer detection.

3. Software specification

In this chapter, we introduce the reader into the basic ideas of software specification in order to comprehend the following work, conformed by the design obtained through this process and some details regarding its implementation in Python.

3.1. Definitions

In this section we describe the concepts of software specification on which the work is based. As we specified on our previous deliverable, we will follow the strategy of the agile version of the Unified Process, presented in (Ambler, 2002). The results of this methodology will be represented in UML diagrams. UML (that stands for Unified Modeling Language) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML is a standardized general-purpose modeling language in the field of object-oriented software engineering but also expressive enough to model other software engineering approaches and even non-software systems.

UML is organized in diagrams, which are drawn to visualize a system from different perspectives, so a diagram is a projection into a system. Diagrams are organized in two main groups:

- **Static diagrams**: which represents the program's structure based on packages and interrelated objects. The main static diagram is the class diagram, which shows the structure of the designed system at the level of classes and interfaces, shows their features, constraints and relationships (associations, generalizations, dependencies, etc).
- **Dynamic diagrams**: which represents the program's behavior based on communications between objects. The main dynamic diagram is the sequence diagram, which describes the temporal sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.

In addition, we must address the focus from which we will approach this process. Since we are developing a software library, we are approaching this process from the perspective of the development view. This view models the organization of the actual software modules (components, subsystems), often organized in layers.

To further develop the work on the previous deliverable (Moret-Bonillo et al., 2021), where we defined the use cases that must be covered, we use the concept of use case realization, an artificial artifact in the sense that it is effectively a collection of one or more models that describes the implementation of a single use case. We do this by exploring the different elements that conform the system via static diagrams and the flows of logic through the use cases—the scenarios defined for each use case—via dynamic diagrams.

With that we cover the necessary aspects for the development view to be complete, so it can satisfy the needs of the implementation process, having covered the work carried out so far.

3.2. Static design

In the first place, we begin with the static design of the system, since we need to comprehend the elements that compose it. This design is illustrated in Figure 1 via a class diagram, as we previously mentioned.

In our case, classes are logically divided into two packages:

- **`knowledge_representation`**: this package is conformed by the classes that allow us to encode knowledge into the system. These being `LeftHandSide` (and the subclasses `Fact`, `AndOperator`, `OrOperator`, `NotOperator`), `Rule` and `KnowledgeIsland`.
- **`qrbs`**: this package is conformed by the classes that manage the encoded knowledge and extract utility from it. These being `QRBS` (and its respective `WorkingMemory` and `InferenceEngine`), `QRBSHandler` that works as a facade between the classical world and the quantum world, and `QPU` and its implementations, that represents the quantum backend in which we evaluate and execute quantum circuits.

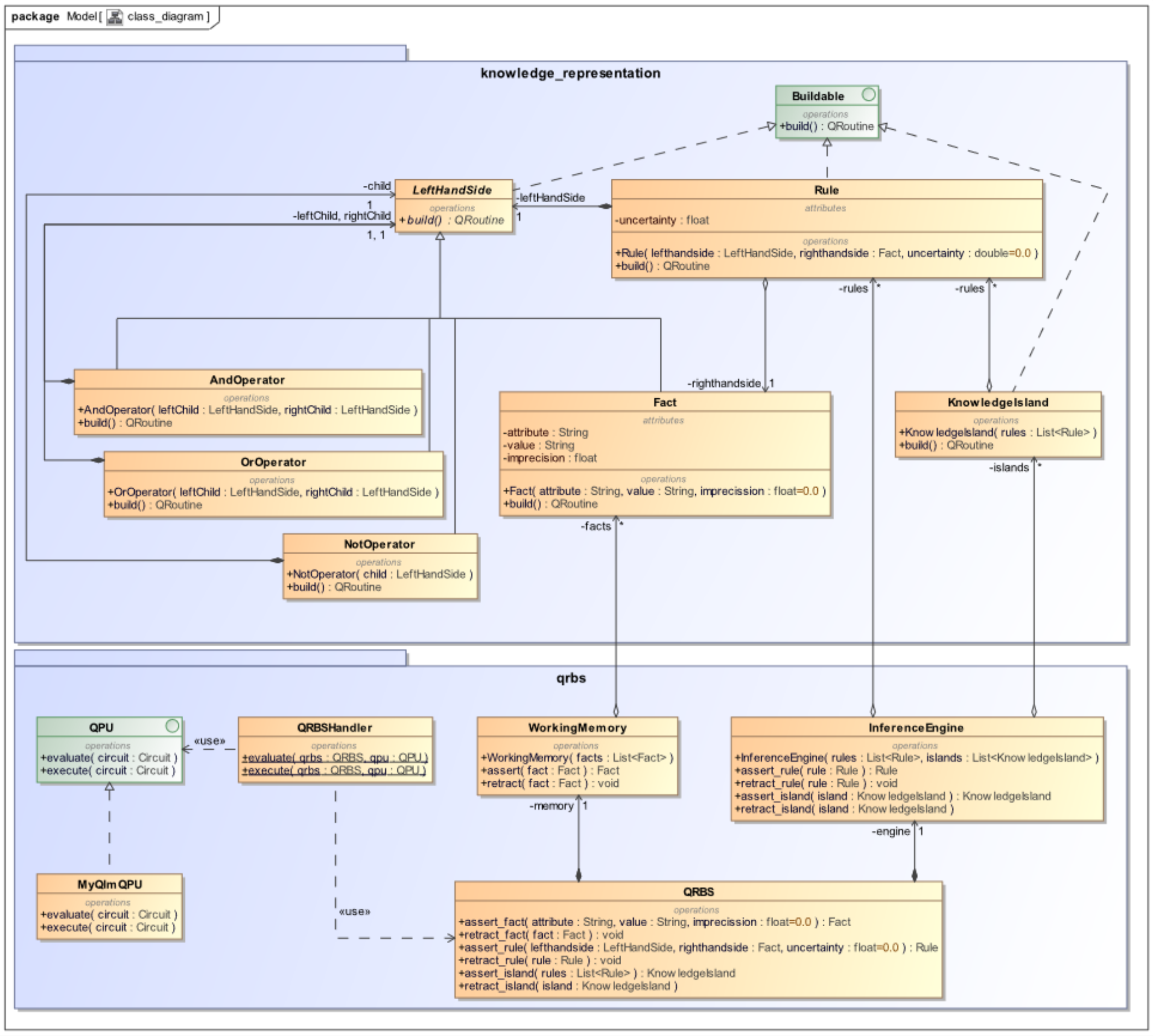

package Model[class_diagram]
knowledge_representation
Buildable
operations
+build() : QRoutine
LeftHandSide
operations
+build() : QRoutine
Rule
attributes
-uncertainty : float
operations
+Rule(lefthandside : LeftHandSide, righthandside : Fact, uncertainty : double=0.0)
+build() : QRoutine
-child
1
-leftChild, rightChild
1, 1
-leftHandSide
1
AndOperator
operations
+AndOperator(leftChild : LeftHandSide, rightChild : LeftHandSide)
+build() : QRoutine
OrOperator
operations
+OrOperator(leftChild : LeftHandSide, rightChild : LeftHandSide)
+build() : QRoutine
NotOperator
operations
+NotOperator(child : LeftHandSide)
+build() : QRoutine
-righthandside
1
Fact
attributes
-attribute : String
-value : String
-imprecision : float
operations
+Fact(attribute : String, value : String, imprecission : float=0.0)
+build() : QRoutine
-rules
*
-rules
*
KnowledgeIsland
operations
+KnowledgeIsland(rules : List<Rule>)
+build() : QRoutine
-islands
*
-facts
*
qrbs
QPU
operations
+evaluate(circuit : Circuit)
+execute(circuit : Circuit)
«use»
QRBSHandler
operations
+evaluate(qrbs : QRBS, qpu : QPU)
+execute(qrbs : QRBS, qpu : QPU)
WorkingMemory
operations
+WorkingMemory(facts : List<Fact>)
+assert(fact : Fact) : Fact
+retract(fact : Fact) : void
InferenceEngine
operations
+InferenceEngine(rules : List<Rule>, islands : List<KnowledgeIsland>)
+assert_rule(rule : Rule) : Rule
+retract_rule(rule : Rule) : void
+assert_island(island : KnowledgeIsland) : KnowledgeIsland
+retract_island(island : KnowledgeIsland)
-memory
1
-engine
1
MyQlmQPU
operations
+evaluate(circuit : Circuit)
+execute(circuit : Circuit)
«use»
QRBS
operations
+assert_fact(attribute : String, value : String, imprecission : float=0.0) : Fact
+retract_fact(fact : Fact) : void
+assert_rule(lefthandside : LeftHandSide, righthandside : Fact, uncertainty : float=0.0) : Rule
+retract_rule(rule : Rule) : void
+assert_island(rules : List<Rule>) : KnowledgeIsland
+retract_island(island : KnowledgeIsland)

***Figure 1**: Class diagram of the system*

With Figure 1, we model all the classes that conform the system and their relationships, allowing us to understand the system itself as a whole. However, we need to provide specific details of each class, which is presented on tables 1 through 14.

These tables are organized with the logically division in mind, having the "`knowledge_representation`" package (tables 1 through 8) and the "`qrbs`" package (tables 9 through 14).

3.2.1. Knowledge representation

Component	
Name	`Buildable`
Type	Interface
Description	Interface for knowledge elements that can be built into quantum circuits.
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

***Table 1**: Software specification of `Buildable` interface*

Component	
Name	`LeftHandSide`
Type	Abstract class
Description	Abstract class for elements that can be part of the left hand side of a rule. This class is used to model the Composite design pattern, acting as the Component interface.
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

***Table 2**: Software specification of `LeftHandSide` abstract class*

Component	
Name	`Fact`
Type	Class
Description	A Fact is the smallest unit of knowledge that can be represented. This class is used to model the Composite design pattern, acting as the Leaf class.
Attributes	
Name	`attribute`
Type	`String`
Description	Attribute that the fact is representing.
Name	`value`
Type	`String`
Description	Value of the attribute that the fact is representing.
Name	`imprecision`
Type	`Float, optional`
Description	Imprecision of the fact; the certainty of the attribute having said value (0 if not specified).
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

***Table 3**: Software specification of `Fact` class*

Component	
Name	`AndOperator`
Type	Class
Description	An AndOperator relates the statements of its children with an AND relationship. This class is used to model the Composite design pattern, acting as (one of) the Composite class.
Attributes	
Name	`leftChild`
Type	`LeftHandSide`
Description	One of the children which is relating.
Name	`rightChild`
Type	`LeftHandSide`
Description	One of the children which is relating.
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

Table 4: Software specification of `AndOperator` class

Component	
Name	`OrOperator`
Type	Class
Description	An OrOperator relates the statements of its children with an OR relationship. This class is used to model the Composite design pattern, acting as (one of) the Composite class.
Attributes	
Name	`leftChild`
Type	`LeftHandSide`
Description	One of the children which is relating.
Name	`rightChild`
Type	`LeftHandSide`
Description	One of the children which is relating.
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

Table 5: Software specification of `OrOperator` class

Component	
Name	`NotOperator`
Type	Class
Description	A NotOperator negates the statement of its child. This class is used to model the Composite design pattern, acting as (one of) the Composite class.
Attributes	
Name	`child`
Type	`LeftHandSide`
Description	Child whose statement is negating.
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

***Table 6**: Software specification of `NotOperator` class*

Component	
Name	`Rule`
Type	Class
Description	A Rule which establishes a relationship (to some level of uncertainty) between a left hand side element and a right hand side, which in this context is a Fact.
Attributes	
Name	`leftHandSide`
Type	`LeftHandSide`
Description	Left hand side element of the rule (also known as precedent).
Name	`rightHandSide`
Type	`Fact`
Description	Right hand side element of the rule (also known as consecuent).
Name	`uncertainty`
Type	`Float, optional`
Description	Uncertainty of the relationship between precedent and consecuent (0 if not specified).
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

***Table 7**: Software specification of `Rule` class*

Component	
Name	`KnowledgeIsland`
Type	Class
Description	A Knowledge Island is a set of rules that conform the inferential reasoning towards a hypothesis.
Attributes	
Name	`rules`
Type	`List<Rule>`
Description	Set of rules that conform the knowledge island.
Methods	
Name	`build`
Description	Builds the correspondent quantum circuit.
Inputs	None
Outputs	A quantum circuit (`QRoutine`).

***Table 8**: Software specification of `KnowledgeIsland` class*

3.2.2. QRBS

Component	
Name	`WorkingMemory`
Type	Class
Description	A Working Memory is an element of a Rule-Based System that manages its facts, keeping trace of their state.
Attributes	
Name	`facts`
Type	`List<Fact>`
Description	List of facts asserted into the system.
Methods	
Name	`assert_fact`
Description	Asserts a fact into the memory.
Inputs	`fact` (`Fact`) - The fact to be asserted.
Outputs	The asserted fact (`Fact`).
Name	`retract_fact`
Description	Retracts a fact from the memory.
Inputs	`fact` (`Fact`) - The fact to be retracted.
Outputs	`None`

Table 9: Software specification of `WorkingMemory` class

Component	
Name	`InferenceEngine`
Type	Class
Description	An Inference Engine is an element of a Rule-Based System that manages its rules and knowledge islands, providing the tools to evaluate them in order.
Attributes	
Name	`rules`
Type	`List<Rule>`
Description	List of rules established for the system.
Name	`islands`
Type	`List<KnowledgeIsland>`
Description	List of knowledge islands established for the system.
Methods	
Name	`assert_rule`
Description	Asserts a rule into the engine.
Inputs	`rule` (`Rule`) - The rule to be asserted.
Outputs	The asserted `rule` (`Rule`).
Name	`retract_rule`
Description	Retracts a rule from the engine.
Inputs	`rule` (`Rule`) - The rule to be retracted.
Outputs	`None`
Name	`assert_island`
Description	Asserts a knowledge island into the engine.
Inputs	`island` (`KnowledgeIsland`) - The knowledge island to be asserted.
Outputs	The asserted knowledge island (`KnowledgeIsland`).
Name	`retract_island`
Description	Retracts a knowledge island from the engine.
Inputs	`island` (`KnowledgeIsland`) - The knowledge island to be retracted.
Outputs	`None`

Table 10: Software specification of `InferenceEngine` class

Component	
Name	`QRBS`
Type	Class
Description	A Quantum Rule-Based System (QRBS) is a Rule-Based System implemented in a quantum computer, taking advantage of some of its capabilities, like quantum superposition, to represent certain aspects such as imprecision and uncertainty.
Attributes	
Name	`memory`
Type	`WorkingMemory`
Description	The Working Memory of the system.
Name	`engine`
Type	`InferenceEngine`
Description	The Inference Engine of the system.
Methods	
Name	`assert_fact`
Description	Creates a fact and asserts it into the system.
Inputs	`attribute` (`String`) – The attribute of the fact. `value` (`Float`) – The value of the fact. `imprecision` (`Float, optional`) – The imprecision of the rule.
Outputs	The asserted fact (`Fact`).
Name	`retract_fact`
Description	Retracts a fact from the system.
Inputs	`fact` (`Fact`) - The fact to be retracted.
Outputs	`None`
Name	`assert_rule`
Description	Creates a rule and asserts it into the system.
Inputs	`lefthandside` (`LeftHandSide`) – The left hand side of the rule. `righthandside` (`Fact`) – The right hand side of the rule. `uncertainty` (`Float, optional`) – The uncertainty of the rule.
Outputs	The asserted rule (`Rule`).
Name	`retract_rule`
Description	Retracts a rule from the system.
Inputs	`rule` (`Rule`) - The rule to be retracted.
Outputs	`None`
Name	`assert_island`
Description	Creates a knowledge island and asserts it into the system.
Inputs	`rules` (`List<Rule>`) – The rules of the knowledge island.
Outputs	The asserted knowledge island (`KnowledgeIsland`).
Name	`retract_island`
Description	Retracts a knowledge island from the system.
Inputs	`island` (`KnowledgeIsland`) - The knowledge island to be retracted.
Outputs	`None`

***Table 11**: Software specification of `QRBS` class*

Component	
Name	`QRBSHandler`
Type	Static class
Description	This class provides several methods to handle operations related to Quantum Rule-Based Systems and its conversion into a quantum circuit in order to be evaluated and executed.
Methods	
Name	`evaluate`
Description	Evaluates whether a QRBS can be executed on a QPU.
Inputs	`qrbs` (`QRBS`) – The QRBS to be evaluated. `qpu` (`QPU`) – The QPU in which the QRBS must be evaluated.
Outputs	The result of the evaluation: either a boolean set to true or an exception will be raised corresponding to the reason for the unsuccessful evaluation.
Name	`execute`
Description	Executes the QRBS on the QPU.
Inputs	`qrbs` (`QRBS`) – The QRBS to be evaluated. `qpu` (`QPU`) – The QPU in which the QRBS must be evaluated.
Outputs	The result of the execution: an object with several data regarding the execution, such as the parameters of the execution (e.g. number of shots), the time it lasted and the values obtained for the hypotheses of the rules/knowledge islands. Also the working memory will be updated according to the execution of the different rules.

Table 12: Software specification of `QRBSHandler` static class

Component	
Name	`QPU`
Type	Interface or abstract class
Description	This class defines the structure to implement Quantum Processing Units (QPU).
Methods	
Name	`evaluate`
Description	Evaluates whether a circuit can be executed on a QPU.
Inputs	`circuit` (`Circuit`) – The circuit to be evaluated.
Outputs	The result of the evaluation as requested by the QRBSHandler.
Name	`execute`
Description	Abstract method. Executes the given quantum circuit on the QPU.
Inputs	`circuit` (`Circuit`) – The quantum circuit to be executed.
Outputs	The result of the execution as requested by the QRBSHandler.

Table 13: Software specification of a `QPU` element

Component	
Name	`MyQlmQPU`
Type	Class
Description	This class provides the myQLM implementation of a Quantum Processing Unit (QPU). myQLM will be the reference implementation for the `QPU` interface, but the system will flexible enough to include other backends dynamically.
Methods	
Name	`evaluate`
Description	Evaluates whether a circuit can be executed on a QPU.
Inputs	`circuit` (`Circuit`) – The circuit to be evaluated.
Outputs	The result of the evaluation as requested by the QRBSHandler.
Name	`execute`
Description	Executes the given quantum circuit on the myQLM backend.
Inputs	`circuit` (`Circuit`) – The quantum circuit to be executed.
Outputs	The result of the execution as requested by the QRBSHandler.

Table 14: Reference implementation of a `QPU` element

3.3. Dynamic design

The next step in the process is to perform a dynamic design in which we show how all the classes described in the static design interact with each other. This dynamic design is illustrated in Figure 2 via a sequence diagram, as we previously mentioned. In this diagram we can see the different interactions that occur between the different objects when a QRBS is popularized with the different elements that form it (facts, rules and knowledge islands) and how to subsequently evaluate and execute the QRBS.

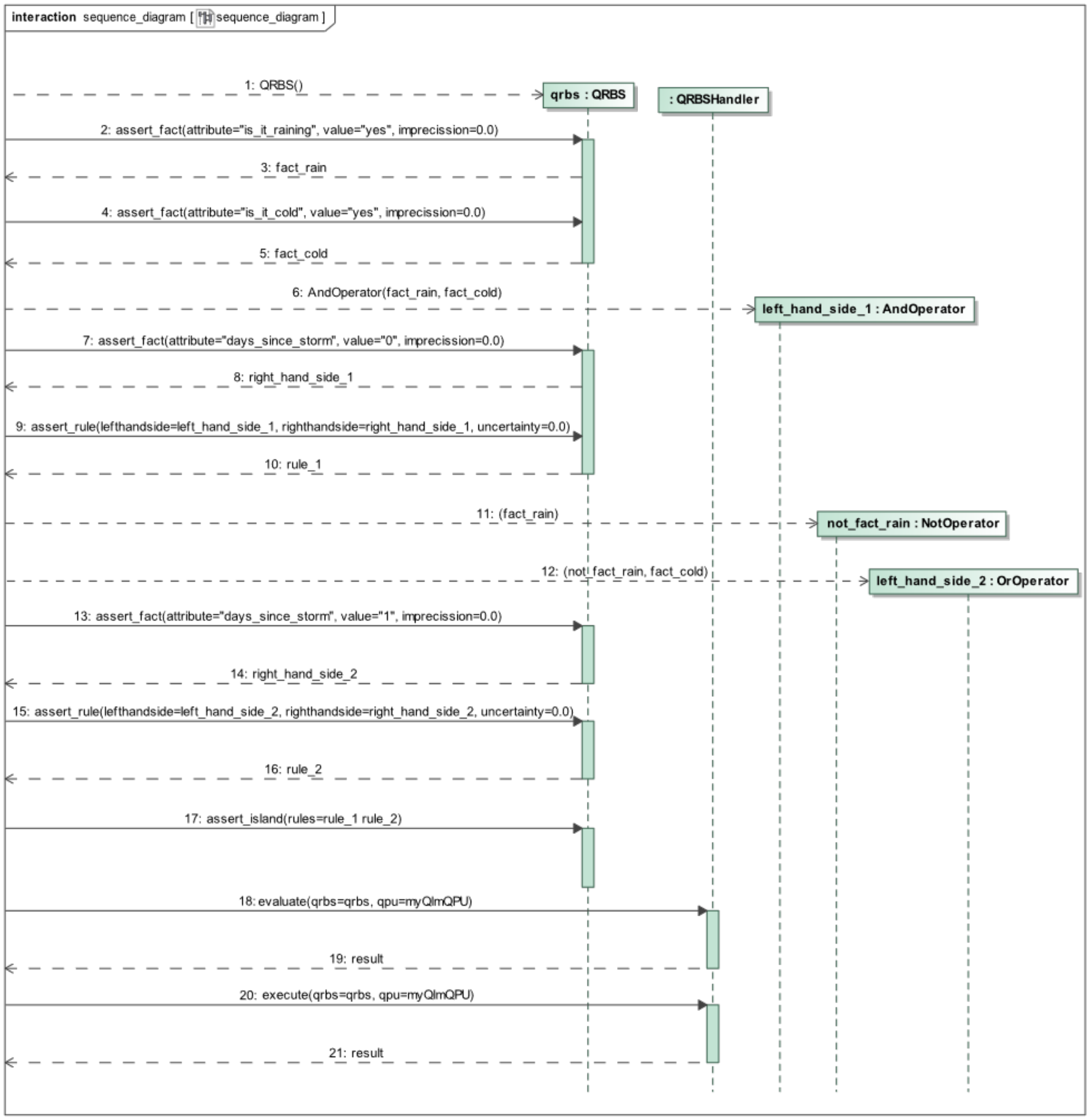

interaction sequence_diagram [sequence_diagram]
qrbs : QRBS
: QRBSHandler
left_hand_side_1 : AndOperator
not_fact_rain : NotOperator
left_hand_side_2 : OrOperator
1: QRBS()
2: assert_fact(attribute="is_it_raining", value="yes", imprecission=0.0)
3: fact_rain
4: assert_fact(attribute="is_it_cold", value="yes", imprecission=0.0)
5: fact_cold
6: AndOperator(fact_rain, fact_cold)
7: assert_fact(attribute="days_since_storm", value="0", imprecission=0.0)
8: right_hand_side_1
9: assert_rule(lefthandside=left_hand_side_1, righthandside=right_hand_side_1, uncertainty=0.0)
10: rule_1
11: (fact_rain)
12: (not_fact_rain, fact_cold)
13: assert_fact(attribute="days_since_storm", value="1", imprecission=0.0)
14: right_hand_side_2
15: assert_rule(lefthandside=left_hand_side_2, righthandside=right_hand_side_2, uncertainty=0.0)
16: rule_2
17: assert_island(rules=rule_1 rule_2)
18: evaluate(qrbs=qrbs, qpu=myQlmQPU)
19: result
20: execute(qrbs=qrbs, qpu=myQlmQPU)
21: result

Figure 2: Sequence diagram of the system

Apart from this sequence diagram we also decided to present, for each use case, a small snippet of pseudocode (choosing a syntax very similar to the final Python implementation that we will obtain later) showing the typical operations described in that particular use case (e.g. create, modify and delete).

3.3.1. UC-01: Declarative knowledge management

The management of declarative knowledge (facts) consists in asserting, modifying and/or retracting facts from the working memory.

```
# Create QRBS
qrbs = QRBS()

# Assert fact
fact = qrbs.assert_fact('is_it_raining', 'yes')

# Modify fact
fact.attribute = 'is_it_cloudy'

# Retract fact
qrbs.retract_fact(fact)
```

3.3.2. UC-02: Procedural knowledge management

The management of procedural knowledge (rules) consists in creating, asserting, modifying and/or retracting rules from the inference engine.

```
# Create QRBS
qrbs = QRBS()

# Create LeftHandSide
fact_rain = qrbs.assert_fact('is_it_raining', 'yes')
fact_cold = qrbs.assert_fact('is_it_cold',  'yes')
left_hand_side = AndOperator(fact_rain, fact_cold)

# Create RightHandSide (Fact)
right_hand_side = qrbs.assert_fact('days_since_storm', 1)

# Assert rule
rule = qrbs.assert_rule(left_hand_side, right_hand_side)

# Modify rule
rule.leftHandSide = OrOperator(NotOperator(fact_rain), fact_cold)

# Retract rule
qrbs.retract_rule(rule)
```

3.3.3. UC-03: Knowledge islands management

Rules are organized in knowledge islands that can be asserted, modified and/or retracted as a whole from the inference engine.

```
# Create QRBS
qrbs = QRBS()

# Create rules
fact_rain = qrbs.assert_fact('is_it_raining', 'yes')
fact_cold = qrbs.assert_fact('is_it_cold', 'yes')

left_hand_side_1 = AndOperator(fact_rain, fact_cold)
right_hand_side_1 = qrbs.assert_fact('days_since_storm', '0')
rule_1 = qrbs.assert_rule(left_hand_side_1, right_hand_side_1)

left_hand_side_2 = OrOperator(NotOperator(fact_rain), fact_cold)
right_hand_side_2 = qrbs.assert_fact('days_since_storm', 1)
rule_2 = qrbs.assert_rule(left_hand_side_2, right_hand_side_2)

# Assert knowledge island
island = qrbs.assert_island([rule1, rule2])

# Modify knowledge island
island.rules = [rule2]

# Retract knowledge island
qrbs.retract_island(island)
```

3.3.4. UC-04: Uncertainty management

Uncertainty management consists in allowing the addition of imprecision to facts (declarative knowledge) and uncertainty to rules (procedural knowledge).

```
# Create QRBS
qrbs = QRBS()

# Create elements
fact_rain = qrbs.assert_fact('is_it_raining', 'yes')
fact_cold = qrbs.assert_fact('is_it_cold', 'yes')

left_hand_side = AndOperator(fact_rain, fact_cold)
right_hand_side = qrbs.assert_fact('days_since_storm', '1')

rule = qrbs.assert_rule(left_hand_side, right_hand_side)

# Add imprecision
fact_rain.imprecision = 0.3
fact_cold.imprecision = 0.8

# Add uncertainty
rule.uncertainty = 0.5
```

3.3.5. UC-05: QRBS management

In the QRBS management, the user must have the ability to create the QRBS, initialize the QRBS with facts and rules (that may be affected by imprecision and uncertainty, respectively), modify these facts and rules and, finally, delete the whole QRBS.

```python
# Create QRBS
qrbs = QRBS()

# Populate QRBS
fact_rain = qrbs.assert_fact('is_it_raining', 'yes')
fact_cold = qrbs.assert_fact('is_it_cold', 'yes')

left_hand_side_1 = AndOperator(fact_rain, fact_cold)
right_hand_side_1 = qrbs.assert_fact('days_since_storm', '0')
rule_1 = qrbs.assert_rule(left_hand_side_1, right_hand_side_1)

left_hand_side_2 = OrOperator(NotOperator(fact_rain), fact_cold)
right_hand_side_2 = qrbs.assert_fact('days_since_storm', '1')
rule_2 = qrbs.assert_rule(left_hand_side_2, right_hand_side_2)

island = qrbs.assert_island([rule1, rule2])

# Modify knowledge island
island.rules = [rule1]

# Modify rule
rule1.leftHandSide = OrOperator(NotOperator(fact_rain), fact_cold)

# Modify fact
fact_rain.attribute = 'is_it_cloudy'

# Modify uncertainty
fact_cold.imprecision = 0.8
rule1.uncertainty = 0.5

# Delete QRBS
del qrbs
```

3.3.6. UC-06: QRBS evaluation

The evaluation provides the user information of the system regarding a specific QPU, to ensure proper execution.

```python
# Create QRBS
qrbs = QRBS()

# Populate QRBS
# For the sake of visibility we assume the QRBS is populated

# Create a QPU implementation
myQlmQPU = MyQlmQPU()

# Evaluate QRBS
result = QRBSHanlder.evaluate(qrbs, myQlmQPU)
```

3.3.7. UC-07: QRBS execution

The execution of a system on a specific QPU returns several values obtained after said execution, such as the corresponding measurements and characteristics or duration of the execution.

```
# Create QRBS
qrbs = QRBS()

# Populate QRBS
# For the sake of visibility we assume the QRBS is populated

# Create a QPU implementation
myQlmQPU = MyQlmQPU()

# Execute QRBS
result = QRBSHanlder.execute(qrbs, myQlmQPU)
```

4. Testing

4.1. Traceability with use cases

One important part of a software development is the ability to trace requirement artifacts through the different stages of software development. Understanding traceability as "The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another."

In the previous deliverable we established a traceability hierarchy (see (Moret-Bonillo et al., 2021), Figure 1, page 8) that starts in user needs and traces to features, uses cases and supplementary specification and, finally, test cases. This connection allows us to make a reverse traceability and to check that, by successfully fulfilling these tests, we are fulfilling use cases that can be traced to user features and needs.

The traceability between use cases and tests was finally included in a table that relates each use case to several test cases (see (Moret-Bonillo et al., 2021), Table 15, page 24). Each test case represents a possible scenario within that particular use case. As an example, we have included in this deliverable the use case UC-01 (Table 15) that shows how to manage declarative knowledge. The three identified scenarios (creation of a fact, deletion of a fact and modification of a fact) were traced to three test cases (see Table 16).

Use case ID	UC-01
Name	Declarative knowledge management
Description	User must be able to create, initialize and handle facts.
Actor	User
Basic flow	1. User selects to create a fact 2. User provides the attribute and the value of the fact. 3. User adds the fact (asserts) to the working memory.
Alternative flows	1a. User selects to delete a fact 1. User selects the fact to be deleted. 2. The fact is deleted (retracted) from the working memory. 1b. User selects to modify a fact 1. User selects the fact to be modified 2. User modifies the attribute or the value of the fact. 3. The fact is updated in the working memory.
Scenarios	1. Creation of a fact: Basic flow 2. Deletion of a fact: Basic flow, alternative 1a 3. Modification of a fact: Basic flow, alternative 1b
Additional information	

Table 15: Specification of use case UC-01

Test case ID	Scenario number	Test case ID	Test case name
UC-01	1	T-01-1	Creation of a fact
UC-01	2	T-01-2	Deletion of a fact
UC-01	3	T-01-3	Modification of a fact

Table 16: Test cases for use case UC-01

The work in this deliverable is to trace this test cases to specific classes in our test suite, and also with specific methods that runs the specific actions related with that tests.

4.2. Tests

Following the strategy defined in the previous section we managed to identify the following test classes and test methods in our software specification (see Table 17).

Test case ID	Test method	Test class
T-01-1	`test_fact_creation`	`TestFact`
T-01-2	`test_fact_deletion`	`TestFact`
T-01-3	`test_fact_modification`	`TestFact`
T-02-1	`test_rule_creation`	`TestRule`
T-02-2	`test_rule_deletion`	`TestRule`
T-02-3	`test_rule_modification`	`TestRule`
T-03-1	`test_island_creation`	`TestKnowledgeIsland`
T-03-2	`test_island_deletion`	`TestKnowledgeIsland`
T-03-3	`test_island_modification`	`TestKnowledgeIsland`
T-03-4	`test_island_creation_error`	`TestKnowledgeIsland`
T-04-1	`test_imprecision`	`TestUncertainty`
T-04-2	`test_uncertainty`	`TestUncertainty`
T-04-3	`test_imprecision_uncertainty`	`TestUncertainty`
T-05-1	`test_qrbs_creation`	`TestQRBS`
T-05-2	`test_qrbs_deletion`	`TestQRBS`
T-05-3	`test_qrbs_modification`	`TestQRBS`
T-06-1	`test_positive_evaluation`	`TestEvaluation`
T-06-2	`test_negative_evaluation`	`TestEvaluation`
T-07-1	`test_successful_default`	`TestExecution`
T-07-2	`test_failed_default_evaluation`	`TestExecution`
T-07-3	`test_failed_default_execution`	`TestExecution`
T-07-4	`test_successful_specified`	`TestExecution`
T-07-5	`test_failed_specified_evaluation`	`TestExecution`
T-07-6	`test_failed_specified_execution`	`TestExecution`

Table 17: Test suite

It is important to note that the tests we will include in the library will not be limited to the ones on Table 17. These are the tests that connect us with the different scenarios of the use cases and, subsequently, with the features and user's needs. So we can say that these are the highest level tests, but that does not prevent other tests from being included to test lower level aspects of the library.

5. Conclusions

In this deliverable we have made the software specification necessary for the implementation of a QRBS. This specification has been divided into two parts: the Static Model, which represents the program's structure based on packages and interrelated objects, and the Dynamic Model, which represents the program's behavior based on communications between objects.

We have used the Unified Modeling Language (UML) for the specification. Here it is important to emphasize that UML is only a language, it is not a methodology. Therefore, UML is processed independently and its artifacts form a common language that eases the communication between developers. In our case, we have followed an iterative and incremental methodology, which has allowed us to obtain a robust overall design, while satisfying the use cases established during the requirement analysis.

We keep following the agile version of the Unified Process, that means that we are not only *designing* the software but we are also *prototyping* the different ideas that came up during the design, which will be a part of the final system. This prototype helps to obtain a better comprehension of the system designed, avoiding coming up with errors on later stages. This prototype is available on its respective NEASQC repository. We want to remark that it has not been officially published yet, since we are still working on it, but we hope to announce its release as soon as possible.

On parallel, there has been work carried out on a separate library, which will provide additional functionality to the one developed here, providing functionalities to parse QRBS defined on JSON files. We hope this library comes in handy to the user of our package, so we will keep informing about its development.

List of Acronyms

Term	Definition
QRBS	Quantum Rule-Based System
IDC	Invasive Ductal Carcinoma
UML	Uniform Modeling Language

Table 18: Acronyms and Abbreviations

List of Figures

List of Tables

Bibliography

Ambler, S. (2002). *Agile modeling: Effective practices for extreme programming and the unified process*. John Wiley & Sons.

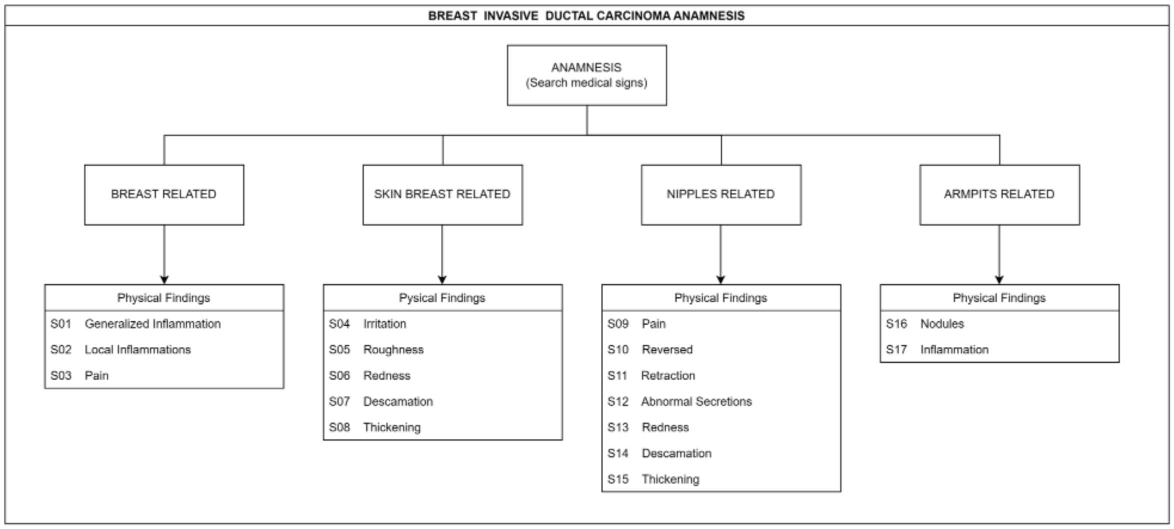
Bourque, P., & Fairley, R. E. (2014). *Guide to the software engineering body of knowledge (SWEBOK©)*. IEEE Computer Society Press.

Kruchten, P. (2004). *The rational unified process: An introduction*. Addison-Wesley.

Moret-Bonillo, V., Mosqueira-Rey, E., & Magaz-Romero, S. (2021). D6.5 Quantum Rule-Based System (QRBS) Requirement Analysis. https://doi.org/10.5281/zenodo.5949157

Pressman, R. S. (2005). *Software engineering: A practitioner's approach*. Palgrave macmillan.

A. Appendix: Preliminary knowledge model for breast cancer (Invasive Ductal Carcinoma - IDC) management



RULES
1. If (Any or Some) S01... S17 (Is or Are) Present Then IDC is Possible
2. If IDC is Possible Then Perform Diagnostic Tests

Figure 3: Breast invasive ductal carcinoma anamnesis

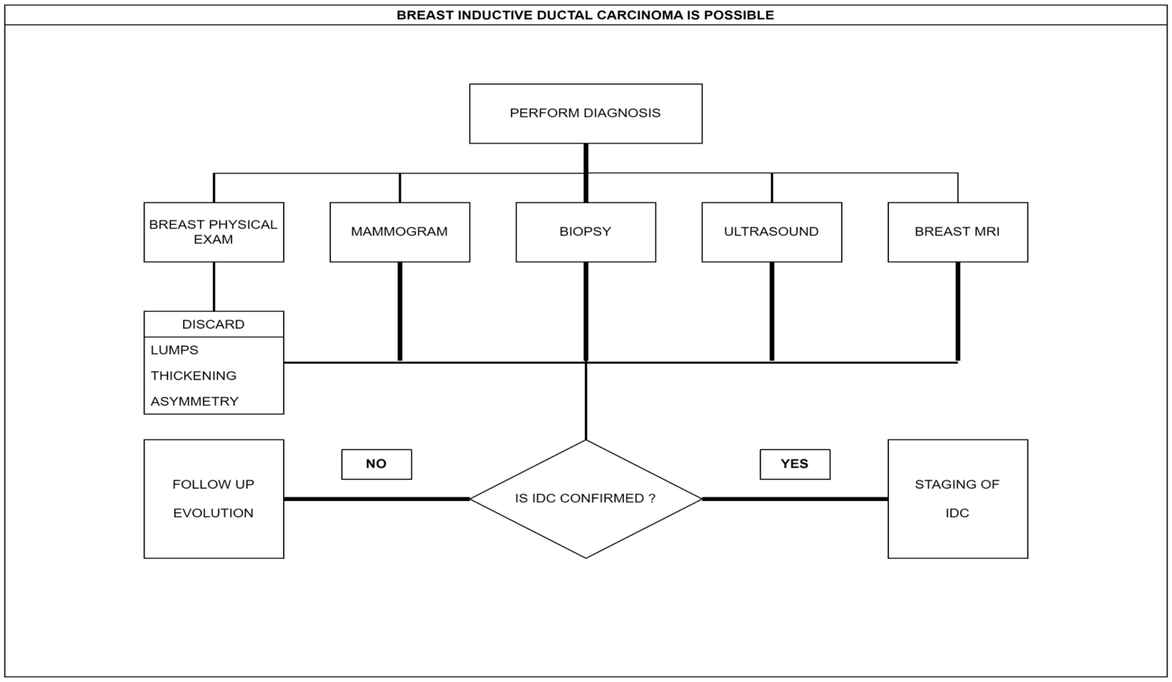


Figure 4: Actions if breast inductive ductal carcinoma is possible

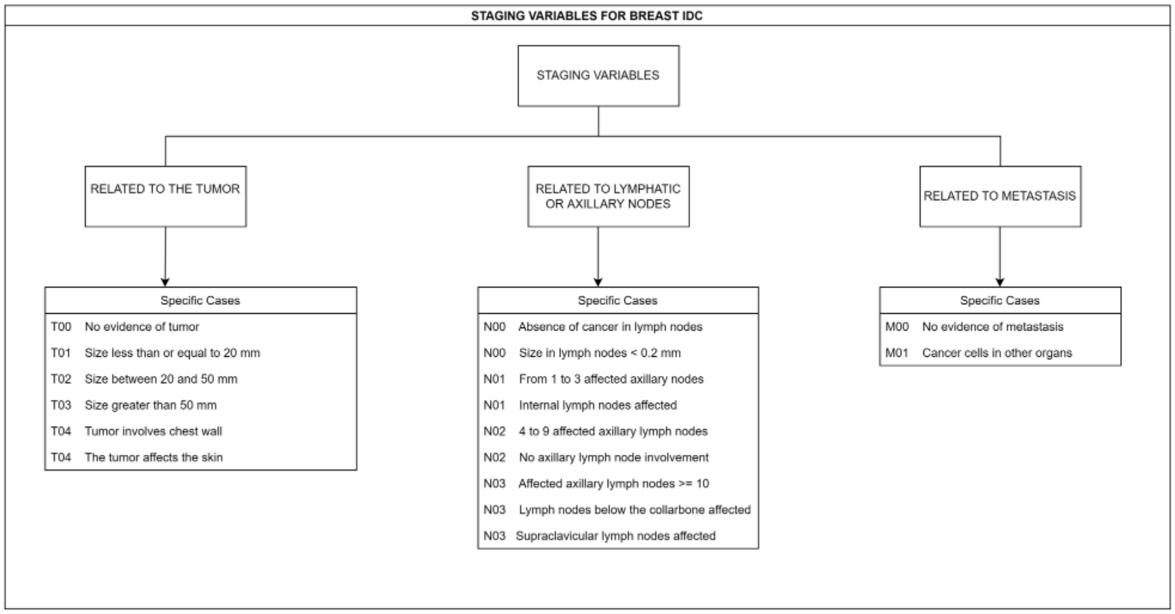
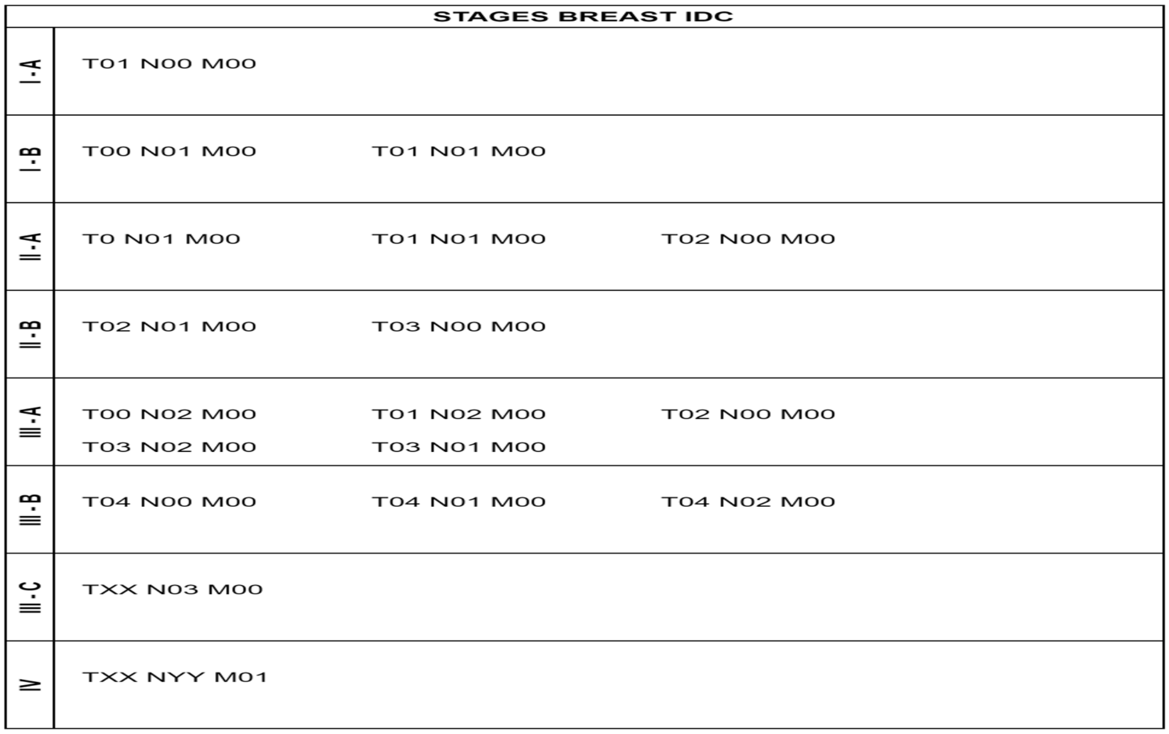
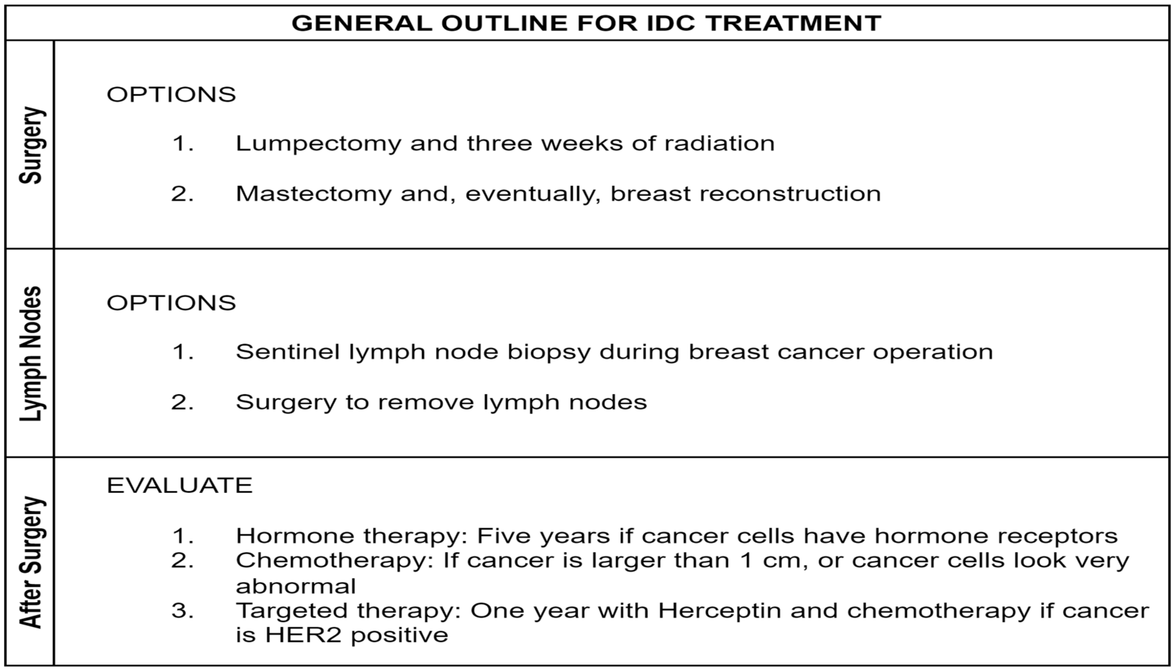

STAGING VARIABLES FOR BREAST IDC
STAGING VARIABLES
RELATED TO THE TUMOR
RELATED TO LYMPHATIC OR AXILLARY NODES
RELATED TO METASTASIS
Specific Cases
T00 No evidence of tumor
T01 Size less than or equal to 20 mm
T02 Size between 20 and 50 mm
T03 Size greater than 50 mm
T04 Tumor involves chest wall
T04 The tumor affects the skin
Specific Cases
N00 Absence of cancer in lymph nodes
N00 Size in lymph nodes < 0.2 mm
N01 From 1 to 3 affected axillary nodes
N01 Internal lymph nodes affected
N02 4 to 9 affected axillary lymph nodes
N02 No axillary lymph node involvement
N03 Affected axillary lymph nodes >= 10
N03 Lymph nodes below the collarbone affected
N03 Supraclavicular lymph nodes affected
Specific Cases
M00 No evidence of metastasis
M01 Cancer cells in other organs

***Figure 5**: Staging variables for breast IDC*



STAGES BREAST IDC			
I-A	T01 N00 M00		
I-B	T00 N01 M00	T01 N01 M00	
II-A	T0 N01 M00	T01 N01 M00	T02 N00 M00
II-B	T02 N01 M00	T03 N00 M00	
III-A	T00 N02 M00 T03 N02 M00	T01 N02 M00 T03 N01 M00	T02 N00 M00
III-B	T04 N00 M00	T04 N01 M00	T04 N02 M00
III-C	TXX N03 M00		
IV	TXX NYY M01		

***Figure 6**: Stages of breast IDC*



GENERAL OUTLINE FOR IDC TREATMENT	
Surgery	OPTIONS 1. Lumpectomy and three weeks of radiation 2. Mastectomy and, eventually, breast reconstruction
Lymph Nodes	OPTIONS 1. Sentinel lymph node biopsy during breast cancer operation 2. Surgery to remove lymph nodes
After Surgery	EVALUATE 1. Hormone therapy: Five years if cancer cells have hormone receptors 2. Chemotherapy: If cancer is larger than 1 cm, or cancer cells look very abnormal 3. Targeted therapy: One year with Herceptin and chemotherapy if cancer is HER2 positive

***Figure 7**: General outline for IDC treatment*