Appendix 1: Formal Description for the UML Metamodel

A1.1 Class Diagram

A class diagram is a 4-tuple $CD = \{\mathbb{C}, \mathbb{A}, \mathbb{R}, WF_{CD}\}$ where

- $\mathbb{C}$ is a non-empty finite set of classes
- $\mathbb{A}$ is a finite set of associations
- $\mathbb{R} \in \mathbb{C}\ x\ \mathbb{C}$ is the relationship between classes
- WF_{CD} is a set of well-formedness rules on the Class Diagram CD

In this subsection, a detailed description of the abstract syntax of UML class diagrams is initially provided followed by a list of formalized well-formedness rules.

- **[CLASS]** A class $\mathtt{C} \in \mathbb{C}$ consists of the following components:
    - $name$ $(\mathbf{C}) \in$ Name where Name is the name space of a class diagram.
    - $upper$ $(\mathbf{C})$ is an optional integer specifying the upper multiplicity.
    - $lower$ $(\mathbf{C})$ is an optional integer specifying the lower multiplicity.
    - $isAbstract$ $(\mathbf{C})$ specifies that the class does not provide a complete declaration.
    - $isRoot$ $(\mathbf{C})$ is a Boolean that specifies whether the class has ancestors or not.
    - $isLeaf$ $(\mathbf{C})$ is a Boolean that specifies whether the class has descendents or not.
- **[ATTRIBUTE]** A class is composed of a set of attributes and operations. An attribute $\boldsymbol{Attr}(\boldsymbol{C})$ of a class is represented by instances of Property and consists of the following components:
    - $visibility$ $(\mathbf{Attr}) \in \{public, private, protected\}$.

- $name$ (**Attr**).
- $type$ (**Attr**) which may be one of the basic types or other classes.
- $upper$ (**Attr**) is an optional integer specifying the upper multiplicity.
- $lower$ (**Attr**) is an optional integer specifying the lower multiplicity.
- $default$ (**Attr**) which is an initial value of the attribute of type $type$ (**Attr**).
- $isReadOnly$ (**Attr**) is a Boolean that specifies whether the attribute is fixed (true) or changeable.
- $isDervied$ (**Attr**) is a Boolean that specifies whether the attribute is derived from other attributes or not.

The default syntax of an attribute declaration given in the UML specification is:

$$[< visibility >][< multiplicity >] < name >:< type > [<= default - value >] \{property\ string\}$$

- **[OPERATION]** An operation $\mathrm{Op}(\mathbb{C})$ of a class is a function that can be performed to alter the behavior of a class. It consists of the following components
  - $visibility$ (**Op**) $\in \{ public, private, protected\}$.
  - $name$ (**Op**).
  - $ret - type$ (**Op**) is an optional return type which may be one of the basic types or other classes.
  - i$sQuery$ (**Op**) is a Boolean that specifies whether its execution changes that system state or not.
  - $isAbstract$ (**Op**) is a Boolean that specifies whether the details of the operation are provided or by a descendent.

- $isUnique$ (**Op**) is a Boolean that specifies whether the return parameter is unique or not.

- **[PARAMETER]** An operation is composed of a list of zero or more formal parameters Param (Op). Each parameter has the following components
  - $name$ (**Param**).
  - $directionKind$ (**Param**) $\in \{in, out, inout, return\}$.
  - $type$ (**Param**) which may be one of the basic types or other classes.
  - $default$ (**Param**) which is an initial value of the parameter of type type(Param).

The default syntax of an operation is given as

$$[< visibility >] < name > [< Parameter\ List >]: [< return - type >]\{property\ string\}$$

and each parameter in the $< Parameter\ List >$ is described as

$$[< directionKind >] < parameter - name >: < parameter - type > [= < default - value >]$$

Classes in a class diagram are related to each other by different types of relationships. Relationships in a UML class diagram are classified into three categories: *Association*, *Generalization* and *Dependency*.

- **[ASSOCIATION]** An association $\mathbb{A}$ consists of an association name and a set of association ends End ($\mathbb{A}$). An association end {e : e ∈ End ($\mathbb{A}$)} consists of the following components:
  - $class$ (**e**) is the class connected to the end.

- $roleName(\mathbf{e})$ which can be used to traverse from the source end to the target end.
- $lower\ (\mathbf{e})$ is an integer that specifies the lower bound on the number of target instances that can be associated with a source instance.
- $upper\ (\mathbf{e})$ is an integer that specifies the upper bound on the number of target instances that can be associated with a source instance.
- $aggregationKind\ (\mathbf{e})$ specifies whether the end is an aggregation with respect to another end. $\text{aggregationKind(e)} \in \{\text{none}, \text{shared}, \text{composite}\}$.
- $navigable\ (\mathbf{e})$ is a Boolean that specifies whether traversing from source to the target instances is possible or not.

- **[GENERALIZATION]** A generalization $\mathbb{G} \in \mathbb{R}$ is a directed relationship between two classes. It consists of:
  - $super\ (\mathbb{G}) \in \mathbb{C}$is the super class.
  - $sub\ (\mathbb{G}) \in \mathbb{C}$ is the sub class.
  - $isSubstitutable\ (\mathbb{G})$ is a Boolean.
- **[ASSOCIATION CLASS]** In the UML Metamodel, an Association Class is a declaration between classes, which has a set of attributes of its own. Association Class is both an Association and a Class. An association class $\mathbb{C}_{\text{assoc}}$ consists of the following component:
  - $name\ (\mathbb{C}_{\text{assoc}}) \in \text{Name}$ where Name is the name space of a class diagram.

Apart from the abstract syntax, the UML specification also provides a set of well-formedness rules. Well-formedness rules for class diagrams written in a formal description can be found in [460]. These set of well-formedness rules (WF) for the UML class diagram are written here in a formal notation. In this subsection, a detailed description of the well-formedness rules of UML class diagrams are provided.

- WF_{CD} **Rule 1:** A well-formed class has unique attribute names

$$\forall\, att_1, att_2 : Attribute\ .$$

$$(att_1 \in \boldsymbol{Attr}(\mathbb{C}) \wedge\ att_2 \in \boldsymbol{Attr}(\mathbb{C})\ \wedge\ att_1 \neq\ att_2) \Rightarrow name(att_1) \neq name(att_2)$$

- WF_{CD} **Rule 2:** Operations can have same names if they differ in scope, types or number of parameters or result type

$$\forall\, op_1, op_2 : Operation\ .$$

$$(op_1 \in \boldsymbol{Op}(\mathbb{C}) \wedge\ op_2 \in \boldsymbol{Op}(\mathbb{C})\ \wedge\ op_1 \neq\ op_2 \wedge\ name(op_1) = name(op_2))$$
$$\Rightarrow \Big(visibility(op_1)$$
$$\neq visibility(op_2) \vee\ diff\_param\big(Param(op_1), Param(op_2)\big)\Big)$$

In this rule diff_param is an auxiliary function that checks whether the parameters (also known as message signature) of the operations are different. This function can be formally written as:

$$diff\_param(Param(op_1), Param(op_2) \equiv$$

$$length\big(Param(op_1)\big) =\ length\big(Param(op_1)\big)$$
$$\Longrightarrow \Big(\exists\, i : i \leq length\big(Param(op_1)\big)\ \wedge type\big(Param(op_1)\big)$$
$$\neq type\big(Param(op_2)\big)\Big)$$

- WF_{CD} **Rule 3:** A class with an abstract operation must be abstract

$$\exists\, op : Operation\,.$$

$$(op \in \boldsymbol{Op}(\mathbb{C}) \wedge isAbstract(op)) \Longrightarrow isAbstract(\mathbb{C})$$

- WF_{CD} **Rule 4:** An abstract class must have at least one abstract operation

$$isAbstract(\mathbb{C}) \Longrightarrow \big(\exists\, op : op \in \boldsymbol{Op}(\mathbb{C}) \wedge isAbstract(op)\big)$$

- WF_{CD} **Rule 5:** Multiplicity of the class must be valid

$$lower(\mathbb{C}) \leq upper(\mathbb{C})$$

- WF_{CD} **Rule 6:** An abstract class must be inherited by another concrete class

$$isAbstract(\mathbb{C}) \Longrightarrow\sim isLeaf(\mathbb{C})$$

- WF_{CD} **Rule 7:** An operation can have at most one return parameter

$$\forall\, op : Operation\,.$$

$$(op \in \boldsymbol{Op}(\mathbb{C}) \Longrightarrow length\big(directionKind\big(Param(op)\big) = return\big) \leq 1$$

- WF_{CD} **Rule 8:** An association is n-ary when n ≥ 2

$$cardinality\big(End(\mathbb{A})\big) \geq 2$$

- WF_{CD} **Rule 9:** Multiplicities of association ends must be well-formed

$$\forall\, e : Association\ End\,.$$

$$\big(e \in End(\mathbb{A}) \Longrightarrow lower(e) \leq upper(e)\big)$$

- WF_{CD} **Rule 10:** An association end with one end as "shared" or "composite" aggregation-kind must be a binary association

$$\forall\, e : Association\ End\ .$$
$$((\, e\ \in End(\mathbb{A})$$
$$\wedge (aggregationKind(e) = composite\ \vee aggregationKind(e) = shared)) \Longrightarrow cardinality(End(\mathbb{A})) = 2)$$

- WF_{CD} **Rule 11:** An association end with one end as "composite" aggregation-kind must be navigable

$$\forall\, e_1, e_2 : Association\ End\ .$$
$$\big((e_1 \in End(\mathbb{A}) \wedge aggregationKind(e_1)\ = composite\ \wedge\ e_2 \in End(\mathbb{A}) \wedge\ e_2 \neq e_1) \Longrightarrow navigable(e_2)\big)$$

- WF_{CD} **Rule 12:** Only one end of an association can be "shared" or "composite"

$$\forall\, e\ : Association\ End\ .$$
$$\Big(\big(\, e\ \in End(\mathbb{A})$$
$$\wedge (aggregationKind(e) = composite\ \vee aggregationKind(e) = shared)\big) \Longrightarrow\sim (\exists\ e_1 : (e_1 \in End(\mathbb{A}) \wedge e_1 \neq e$$
$$\wedge (aggregationKind(e_1) = composite\ \vee aggregationKind(e_1) = shared))\Big)$$

- WF_{CD} **Rule 13:** An association end with one end as "composite" aggregation-kind, that end cannot have multiplicity greater than 1

$$\forall\, e : Association\ End\ .$$

$$\big((\, e\ \in End(\mathbb{A}) \wedge aggregationKind(e) = composite\,) \Longrightarrow upper(e) \leq 1\big)$$

- WF_{CD} **Rule 14:** In an association, at least one end must be navigable

$$\forall\, e : Association\ End\ .$$

$$\big(\, e\ \in End(\mathbb{A}) \Longrightarrow navigable(e)\big)$$

- WF_{CD} **Rule 15:** In a generalization relationship, the subclass cannot be a root

$$\forall\, g_1 : Generalization\ .$$

$$\Big(g_1 \in \mathbb{R} \Longrightarrow\sim isRoot\big(sub(g_1)\big)\Big)$$

- WF_{CD} **Rule 16:** In a generalization relationship, the super class cannot be a leaf

$$\forall\, g_1 : Generalization\ .$$

$$\Big(g_1 \in \mathbb{R} \Longrightarrow\sim isLeaf\big(super(g_1)\big)\Big)$$

- WF_{CD} **Rule 17:** In a generalization relationship, the subclass cannot redefine the attributes of the super class

$$\forall\, g_1 : Generalization\ .$$

$$g_1 \in \mathbb{R} \wedge \Big(\forall\, att_1, att_2 : \Big(att_1 \in \boldsymbol{Attr}\big(super(g_1)\big) \wedge\ att_2 \in \boldsymbol{Attr}\big(sub(g_1)\big)\Big)$$

$$\Rightarrow name(att_1) \neq name(att_2)\Big)$$

- WF_{CD} **Rule 18:** Each class in the class diagram has a unique name

$$\forall\, c_1, c_2 : Class\ .$$

$$((c_1 \in \mathbb{C}\ \wedge\ c_2\ \in \mathbb{C}\ \wedge\ c_1 \neq c_2) \Longrightarrow name(c_1) \neq name(c_2))$$

- WF_{CD} **Rule 19:** Two different associations relating to a common class cannot have the same name

$$\forall\, a_1, a_2 : Association\ .$$

$$((a_1 \in\ \mathbb{A}\ \wedge\ a_2 \in\ \mathbb{A}\ \wedge a_1 \neq a_2\ \wedge name(a_1) =\ name(a_2)) \Longrightarrow End(a_1) \cap End(a_2) = \{\}$$

- WF_{CD} **Rule 20:** An abstract class in the class diagram must be the super class of at least one concrete class

$$\forall c : Class\ .$$

$$(c\ \in\ \mathbb{C}\ \wedge isAbstract(c)) \Longrightarrow (\exists\ g : Generalization\ .$$

$$super(g) =\ c)$$

- WF_{CD} **Rule 21:** There should be no loops among generalizations in a class diagram

$$\forall\, uc_1 : Use\ Case\ .$$

$$uc_1 \in \mathbb{U}\ \Longrightarrow \sim\big(uc_1 \in allIncluded(uc_1)\big)$$

In this rule allIncluded(uc)is an auxiliary function that returns the transitive closure of all the use cases included by this use case directly or indirectly. This function can be formally written as

$$allIncluded : \mathbb{U} \rightarrow \mathbb{U} - \boldsymbol{set}$$

$$allIncluded(uc) \equiv \{(uc_1) \mid (uc_1) : UseCase \,.$$

$$(\exists\, inc_1 : Inclusion \,.$$

$$inc_1 \in \mathbb{r} \wedge (including(\, inc_1) = \, uc) \wedge$$

$$addition(\, inc_1) \in uc_1 \, \wedge allincluded(addition(\, inc_1)))\}$$

A1.2 Sequence Diagram

A sequence diagram is a 7-tuple $SEQ = \{\mathbb{L}, End, Mes, \mathbb{E}, \leq, fragment, WF_{SEQ}\}$ where

- $\mathbb{L}$ is a finite set of lifelines
- End is a finite set of end locations
- Mes is a finite set of message labels
- $\mathbb{E} \subseteq End \times Mes \times End$ is the relationship (event) between lifelines
- $\leq \subseteq End \times End$ is a partial order providing the position of ends within each of the lifelines
- $fragment$ is an ordered set of fragments in the sequence diagram
- WF_{SEQ} is a set of well-formedness rules on the Sequence Diagram SEQ

Similar to that of the Class diagram, the UML Specification document also describes the Sequence Diagram metamodel by an abstract syntax in the form of a class diagram and the well-formedness rules. In this subsection, a detailed description of the abstract syntax of UML sequence diagrams will be provided.

- **[LIFELINE]** A lifeline l ∈ 𝕃 consists of the following components
  - $name$ (**l**).
  - $endList$ (**l**) ⊆ End is a set of all end locations part of the lifeline whose ordering is provided by using the " ≤ " relational operator.
  - $attributes$ (**l**) is a set of attributes that belongs to a lifeline.
  - $decomposedAs$ (**l**) is the name of the decomposed fragment that shows the interactions for the decomposed lifeline.
- **[DECOMPOSITION]** A decomposed fragment of a lifeline l is given by an external sequence diagram SEQ_l.
- **[END LOCATION]** An end location end ∈ End consists of the following components:
  - $name$ (**end**).
  - $lifeline$ (**end**) ∈ 𝕃 is the lifeline to which this end belongs to.
  - $isGate$ (**end**) is a Boolean that specifies whether the end is a gate or not.
  - $formalGate$ (**end**) ∈ fragment is the fragment to which end belongs if the end is a gate.
- **[MESSAGE]** A message M ∈ 𝔼consists of the following components:
  - $name$ (**M**) ∈ Mes.
  - $messageKind$ (**M**) ∈ $\{syncCall, asyncCall, createMsg, deleteMsg, return\}$.
  - $messageSort$ (**M**) ∈ $\{complete, lost, found, unknown\}$.
  - $retAttr$ (**M**) is an optional attribute to which the return value is assigned.
  - $retVal$ (**M**) is the return value of the message.

- $sendEnd$ $(\mathbf{M}) \in$ End specifies the sending end of a message.
- $recieveEnd$ $(\mathbf{M}) \in$ End specifies the receiving end of a message.

- **[ARGUMENTS]** A message is composed of a list of zero or more argumentsArg (M). Each argument has the following components:
  - $name$ $(\mathbf{Arg})$.
  - val $(\mathbf{Arg})$ is a value assigned to the argument or '-' if not assigned.

The default syntax of a message is given as

$$[< retAttr >'= '] < name > ['('[< Argument\ List >]')'][':' < retVal >]$$

and each argument in the $< Argument\ List >$ is described as

$$([< name >'='] < val >)|\ ' - '$$

Fragments fragment in a sequence diagram are classified into three categories: *Combined Fragments*, *Interaction Use Fragments* and *State Invariants*.

- **[COMBINED FRAGMENT]** A combined fragment cf $\in$ fragment consists of the following components:
  - $covered(\mathbf{cf}) \subseteq \mathbb{L}$ is a set of lifelines covered by the fragment.
  - $operator$ $(\mathbf{cf}) \in$ $\{opt, loop, break, neg, par, alt, assert, strict, seq, consider, ignore\}$.
  - $operand$ $(\mathbf{cf})$ is a set of operands of the combined fragment.
  - $fragmentgate$ $(\mathbf{cf}) \subseteq$ End is a set of gates between the fragment and its enclosing interaction.

- **[OPERAND]** An operand $\text{opd} \in \text{operand(cf)}$ consists of an interaction constraint constraint(opd) and an operand body. An operand body is given by an inline sequence diagram SEQ_{opd}.
  - **[CONSTRAINT]** constraint(opd) is an interaction constraint given as a Boolean expression which guards the entry into an operand. It includes the following components:
    - $minint$ (**constraint**) is an optional value or an expression that specifies the minimum number of iterations.
    - $maxint$ (**constraint**) is an optional value or an expression that specifies the maximum number of iterations.

The default syntax of an interaction constraint is given by
$('['(< boolean\ expression > \mid '\boldsymbol{else}' \mid)']')$

- **[INTERACTION USE]** An interaction use $\text{ref} \in \text{fragment}$ is given by the same default syntax as that of a message but the name in this case refers to the referred interaction. The referred interaction is an external sequence diagram SEQ_{ref}. An interaction use fragment also consists of $\text{actualgate (ref)} \subseteq \text{End}$ is a set of gates between the fragment and its enclosing interaction.
- **[STATE INVARIANT]** A state invariant $\text{inv} \in \text{fragment}$ consists of the following components
  - $covered$ (**inv**) $\in \mathbb{L}$ is the lifeline covered by the state invariant.
  - $condition$ (**inv**) is the constraint that should hold at runtime.

Also in this subsection, a detailed description of the well-formedness rules of UML sequence diagrams is provided.

- WF_{SEQ} **Rule 1:** If in a sequence diagram a lifeline is decomposed, the sequence of constructs in the diagram such as combined fragments and interaction use covering this lifeline must also appear in the decomposed interaction. This is also known as extra-global.

$$\forall\, l_1 : Lifeline\ .$$

$$l_1 \in \mathbb{L} \wedge decomposedAs(l_1) \neq \emptyset \implies partOf(l_1) = fragment(SEQ_{l_1})$$

In this rule partOf(l)is an auxiliary function that returns all the fragments that the lifeline l is part of. This function can be formally written as

$$partOf : \mathbb{L} \rightarrow fragment - \boldsymbol{set}$$

$$partOf(l) \equiv$$

$$\{f \mid f : fragment\ .$$

$$(\exists\, cf_1 : fragment\ .$$

$$l \in covered(cf_1) \wedge (f) = (cf_1))\}$$

- WF_{SEQ} **Rule 2:** The Send event must be ordered before the receive event if both the send and the receive event belonging to a message are on the same lifeline

$$\forall\, E_1 : Message\ .$$

$$E_1 \in \mathbb{E} \wedge (lifeline(sendEnd(E_1)) = lifeline(recieveEnd(E_1))) \implies sendEnd(E_1) \leq recieveEnd(E_1)$$

- WF_{SEQ} **Rule 3:** If a return attribute is specified in a message, it must be an attribute of the lifeline sending the message

$$\forall E_1 : Message\ .$$

$$E_1 \in \mathbb{E} \wedge isGate\big(recieveEnd(E_1)\big) \wedge \Longrightarrow covered(cf_1) = \mathbb{L}$$

- WF_{SEQ} **Rule 4:** Arguments of a message must be attributes of the sending lifeline or constants

$$\forall E_1 : Message\ .$$

$$E_1 \in \mathbb{E} \implies name\big(\boldsymbol{Arg}(E_1)\big) = attributes\left(lifeline\big(sendEnd(E_1)\big)\right) \vee Nat\_Num$$

- WF_{SEQ} **Rule 5:** Messages inside of a combined fragment should not cross its boundaries or its operands within the combined fragment

$$\forall cf_1 : Fragment\ .$$

$$cf_1 \in Fragment \wedge (\exists\ opd_1,\ opd_2 : Operand\ .$$

$$opd_1 \in operand(cf_1) \wedge opd_2 \in operand(cf_1)) \implies End(\ opd_1) \cap End(\ opd_2) = \{\ \}$$

- WF_{SEQ} **Rule 6:** A combined fragment with operator opt, loop, break or neg must have exactly one operand

$$\forall cf_1 : Fragment\ .$$

$$cf_1 \in fragment \wedge \big((operator(cf_1) = opt) \vee (operator(cf_1) = loop) \vee (operator(cf_1) = break) \vee (operator(cf_1) = neg)\big) \implies size\big(operand(cf_1)\big) = 1$$

- WF_{SEQ} **Rule 7:** The interaction constraint with minint and maxint applies only to a combined fragment with operator loop

$$\forall\, cf_1 : Fragment\ .$$
$$cf_1 \in fragment \wedge \sim (operator(cf_1) = loop)$$
$$\Longrightarrow \left(minint\left(constraint\left(operand(cf_1)\right)\right) = \emptyset\right)$$
$$\wedge \left(maxint\left(constraint\left(operand(cf_1)\right)\right) = \emptyset\right)$$

- WF_{SEQ} **Rule 8:** A combined fragment with operator break should cover all the lifelines within the enclosing sequence diagram

$$\forall\, cf_1 : Fragment\ .$$
$$cf_1 \in fragment \wedge (operator(cf_1) = break) \Longrightarrow covered(cf_1) = \mathbb{L}$$

- WF_{SEQ} **Rule 9:** A combined fragment with operator loop and minint interaction constraint specified then the evaluation of minint should be a non-negative integer

$$\forall\, cf_1 : Fragment\ .$$
$$cf_1 \in fragment \wedge operator(cf_1)$$
$$= loop \wedge \left(minint\left(constraint\left(operand(cf_1)\right)\right) \neq \emptyset\right)$$
$$\Longrightarrow \left(evaluate(minint\left(constraint\left(operand(cf_1)\right)\right)) \geq 0\right)$$

- WF_{SEQ} **Rule 10:** A combined fragment with operator loop and maxint interaction constraint specified then the evaluation of maxint should be a positive integer

$$\forall\, cf_1 : Fragment\ .$$
$$cf_1 \in fragment \ \wedge\ operator(cf_1) = loop \wedge \left(maxint\left(constraint\left(operand(cf_1)\right)\right) \neq \emptyset\right) \Longrightarrow \left(evaluate(maxint\left(constraint\left(operand(cf_1)\right)\right)) > 0\right)$$

- **WF_{SEQ} Rule 11:** A combined fragment with operator loop and both minint and maxint interaction constraint specified, then the evaluation of maxint should be greater than or equal to the evaluation of minint

$$\forall\, cf_1 : Fragment\ .$$
$$cf_1 \in fragment \ \wedge\ operator(cf_1) = loop \ \wedge \left(minint\left(constraint\left(operand(cf_1)\right)\right) \neq \emptyset\right) \wedge \left(maxint\left(constraint\left(operand(cf_1)\right)\right) \neq \emptyset\right) \Longrightarrow \left(evaluate(maxint\left(constraint\left(operand(cf_1)\right)\right)) \geq evaluate(maxint\left(constraint\left(operand(cf_1)\right)\right))\right)$$

A1.3 Use Case Diagram

A use case diagram is a 5-tuple $UC = \{\mathbb{U}, \mathbb{a}, \mathbb{m}, \mathbb{r}, WF_{UC}\}$ where

- $\mathbb{U}$ is a finite set of use cases
- $\mathbb{a}$ is a finite set of actors
- $\mathbb{m} \subseteq \mathbb{a} \times \mathbb{U}$ is a finite set of associations
- $\mathbb{r} \subseteq \mathbb{U} \times \mathbb{U}$ is the relationship between use cases

- WF_{UC} is a set of well-formedness rules on the Use Case Diagram UC

In this subsection, a detailed description of the abstract syntax of UML use case diagrams will be provided.

- **[ACTOR]** An actor $a \in \mathbb{a}$ consists of the following components
  - $name(a)$.
- **[USE CASE]** A use case $\mathrm{uc} \in \mathbb{U}$ consists of the following components
  - $\mathrm{name(uc)}$.
  - $\mathrm{extPoint(uc)}$ is a set of all extension points owned by the use case.
- **[EXTENSION POINT]** An extension Point ep belonging to a use case has a name $\mathrm{name(ep)}$.

The default syntax of an extension point is given by $< name > [: < explanation >]$

- **[ASSOCIATION]** An association relationship $\mathrm{m} \in \mathbb{M}$ consists of the following components:
  - $\mathrm{subject(m)} \in \mathbb{a}$ is the actor.
  - $\mathrm{use(m)} \in \mathbb{U}$ is the use case.

Use cases in a use case diagram are related to each other by different types of relationships. These relationships are generalization, inclusion and extension.

- **[GENERALIZATION]** A generalization relationship $\mathrm{gen} \in \mathbb{r}$ consists of the following components:
  - $\mathrm{super(gen)} \in \mathbb{U}$ is the general use case.

- sub(gen) $\in \mathbb{U}$ is the specialized use case.

- **[INCLUSION]** An inclusion relationship inc $\in \mathbb{r}$ consists of the following components:
  - addition(inc) $\in \mathbb{U}$ is the use case that is to be included.
  - including(inc) $\in \mathbb{U}$ is the use case that will include the addition.
- **[EXTENSION]** An extension relationship ext $\in \mathbb{r}$ consists of the following components:
  - extended(ext) $\in \mathbb{U}$ is the use case that is being extended (base).
  - extension(ext) $\in \mathbb{U}$ is the use case that is represents the extension.
  - condition(ext) is condition that must hold for the extension to take place.
  - extLoc(ext) is an ordered list of extension points ep where fragments of the extending use case are to be inserted.
- **[ACTOR GENERALIZATION]** An actor generalization relationship gen_aconsists of the following components:
  - super(gen_a) $\in \mathbb{a}$ is the general actor.
  - sub(gen_a) $\in \mathbb{a}$ is the specialized actor.

Also in this subsection, a detailed description of the well-formedness rules of UML use case diagrams is provided.

- WF_{UC} **Rule 1:** An actor must have a name

$$\forall\, a_1 : Actor\ .$$

$$a_1 \in \mathbb{a} \implies name(a_1) \neq \emptyset$$

- WF_{UC} **Rule 2:** A use case must have a name

$$\forall\, uc_1 : Use\ Case\ .$$

$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq \emptyset$$

- WF_{UC} **Rule 3:** A use case cannot include use cases that directly or indirectly include it.

$$\forall\, uc_1 : Use\ Case\ .$$

$$uc_1 \in \mathbb{U} \implies \sim\big(uc_1 \in allIncluded(uc_1)\big)$$

In this rule allIncluded(uc) is an auxiliary function that returns the transitive closure of all the use cases included by this use case directly or indirectly. This function can be formally written as

$$allIncluded : \mathbb{U} \to \mathbb{U} - \boldsymbol{set}$$

$$allIncluded(uc) \equiv$$

$$\{(uc_1) \mid$$

$$(uc_1) : UseCase\ .$$

$$(\exists\, inc_1 : Inclusion\ .$$

$$inc_1 \in \mathbb{r} \wedge (including(\, inc_1) = uc) \wedge$$

$$addition(\, inc_1) \in uc_1 \wedge allincluded(addition(\, inc_1)))\}$$

- WF_{UC} **Rule 4:** An extension point must have a name

$$\forall\, uc_1 : Use\ Case\ .$$

$$uc_1 \in \mathbb{U} \implies name(extPoint((uc_1)) \neq \emptyset$$

- WF_{UC} **Rule 5:** The extension locations referenced by the extend relationship must belong to the use case being extended

$$\forall\, ext_1 : Extension\ .$$

$$ext_1 \in \mathbb{r} \implies extLoc(ext_1) = extPoint(extended(ext_1))$$

Appendix 2: Model Refactoring Catalog

This section provides the specification of all model level refactorings. These refactorings are grouped into three categories based on the model they transform: Use Case, Class and Sequence. Each refactoring is described in detail. Refactoring pre-conditions and post-conditions are defined using notations and functions described in Appendix 1. These refactorings are provided as a Java API (library – jar). In order to invoke these refactorings, the UML model should be parsed and used as a DOM tree. The document node of that tree is passed on each invocation.

A2.1 Use Case Model Refactoring

1. Create Use Case

Description: This refactoring creates a new empty use case without any associated actors and any associated interaction.

Origin: From Rui [286] [page 134]

Parameters: String *newUC*

Preconditions: The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$\forall\, uc_1 : Use\ Case\ .$

$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC$$

Mechanism &Verification: The behavior of the use case model is not affected with the addition of the newly created use case. The precondition ensures preservation of distinct entity name invariant.

Implementation:

- Method Name: create_UseCase
- Arguments: Document doc, String name where
    - *doc* is the document node of the source model
    - *name* is the name for the use case
- Return Value: String *status*

2. Create Actor

Description: This refactoring creates a new actor without any reference to a use case(s).

Origin: From Rui [286] [page 135]

Parameters: String *newActor*

Preconditions: The name of the new actor ($newActor$) does not conflict with the name of an existing actor within the model.

$\forall\, a_1 : Actor$.

$$a_1 \in \mathbb{a} \implies name(a_1) \neq newActor$$

Post-conditions:

$$\exists\, a \in \mathbb{a}' \wedge a \notin \mathbb{a} \wedge name(a) = newActor$$

Mechanism &Verification: The newly created actor does not interact with any use case and is isolated from other actors. Therefore, the behavior of the use case model does not change with the addition of a new actor. The precondition ensures preservation of distinct entity name invariant.

Implementation:

- Method Name: create_Actor
- Arguments: String name where
    - *name* is the name for the actor
- Return Value: String *status*

3. Delete Use Case

Description: This refactoring deletes an unreferenced use case from the use case model.

Origin: From Rui [286] [page 137]

Parameters: Use case *uc*

Preconditions: The use case is isolated from other use cases and actors. Isolation from other use cases means

- No inclusions
- No extensions

- Not included and extended by other use cases
- Not a super use case to other use cases

$$\exists\, uc\colon UseCase\ .$$
$$\forall\, m\colon Association.$$
$$m \in \mathbb{M} \wedge use(m) \neq uc$$
$$\forall\, rel\colon Relationship.$$
$$rel \in \mathbb{r} \wedge including(rel) \neq uc \vee extended(rel) \neq uc \vee super\,(rel) \neq uc$$
$$\forall\, uc_1\colon Use\ Case.$$
$$uc_1 \in \mathbb{U} \wedge uc \notin allIncluded(uc_1) \wedge uc \notin allExtended(uc_1)$$

Post-conditions:

$$uc \notin \mathbb{U}'$$

Mechanism &Verification: Since the use case is isolated from other use cases and actors, it does not affect interactions between them. Hence, this deletion does not change the behavior of the use case model.

Implementation:

- Method Name: delete_UseCase
- Arguments: String name where
  - *name* is the name of the use case
- Return Value: String *status*

4. Delete Actor

Description: This refactoring deletes an unreferenced actor from the use case model.

Origin: From Rui [286] [page 138]

Parameters: Actor *a*

Preconditions: The actor is isolated from other use cases and actors. Isolation from other actors means that the actor is not a super-actor to any other actor.

$$\exists\, a{:}\, Actor\ .$$
$$\forall\, m{:}\, Association.$$
$$m \in \mathbb{M} \ \wedge\ subject(m) \neq a$$
$$\forall\, rel{:}\, Relationship.$$
$$rel \in \mathbb{g} \ \wedge\ super\ (rel)\ \neq a$$

Post-conditions:

$$a \notin \mathbb{a}'$$

Mechanism &Verification: Since the actor is isolated from other use cases and actors, it does not participate in interactions between them. Hence, this deletion does not change the behavior of the use case model.

Implementation:

- Method Name: delete_Actor
- Arguments: String name where
    - *name* is the name of the actor
- Return Value: String *status*

5. Generalize Use Cases

Description: This refactoring creates a generalization relationship between two or more use cases. This refactoring reduces redundancy in use cases by moving common interactions to the parent use case and hence improves reusability.

Origin: From Rui [286] [page 154]

Parameters: A set of use cases $\{uc_1, uc_2 \ldots uc_n\}$, String *newUC*

Preconditions:

(i) The use cases $\{uc_1, uc_2 \ldots uc_n\}$ are used by the same set of actors. In order to formally write this condition, we define an auxiliary function that returns all the actors associated with a given use case. This function can be written as

$$
\begin{aligned}
&allActors : \ \mathbb{U} \rightarrow \mathbb{A} - set \\
&allActors(uc) \equiv \\
&\{(a_i) | \\
&\qquad (a_i) : Actor \ . \\
&\qquad (\exists \ m : Association \ . \\
&\qquad m \in \mathbb{M} \ \wedge \ use(m) = uc \ \wedge \\
&\qquad subject(m) \in a_i)\}
\end{aligned}
$$

Then the precondition can be written as

$$
\begin{aligned}
&\forall uc : UseCases \in \{uc_1, uc_2 \ldots uc_n\} \ . \\
&\qquad uc \in \mathbb{U} \ \wedge \ \forall \ uc_i, uc_j : \in \{uc_1, uc_2 \ldots uc_n\} \wedge \\
&\qquad allActors(uc_i) = allActors(uc_j)
\end{aligned}
$$

(ii) There is no relationship among the use cases {uc_1, uc_2… uc_n}. These use cases are not referenced by any other use case.

$$\forall\, uc: UseCases \in \{uc_1, uc_2 \ldots uc_n\}\,.$$
$$uc \in \mathbb{U} \wedge allIncluded(uc) = \{\} \wedge allExtended(uc) = \{\}$$

(iii) The name of the new super use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1: Use\ Case\,.$$
$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

A new use case is created and it is the parent of use cases {uc_1, uc_2… uc_n}.

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC \wedge$$
$$\forall\, uc \in \{uc_1, uc_2 \ldots uc_n\}\ .$$
$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge super(rel) = newUC \wedge sub(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the parent or super use cases of the given use cases. In the behavioral view, common interaction fragment is moved to this use case.

The precondition (i) ensures that the use cases {uc_1, uc_2… uc_n} has the same set of actors. According to the definition of generalization relationship, moving common interaction elements to the parent use case does not change the behavior of the use cases.

Precondition (ii) ensures that the use cases are isolated. Precondition (iii) ensures distinct entity name for the newly added parent use case.

Implementation:

- Method Name: generalize_UseCases
- Arguments: ArrayList *subUCNames*, String *superUCName* where
    - *subUCNames* are the names of the child use cases
    - *superUCName* is the name of the parent use case
- Return Value: String *status*

6. Generalize Actors

Description: This refactoring creates a generalization relationship between two or more actors using a common set of use cases. A new actor is created which uses the above common set of use cases.

Origin: From Rui [286] [page 157]

Parameters: A set of actors $\{a_1, a_2 \ldots a_n\}$, String *newActor*

Preconditions:

(i) The actors $\{a_1, a_2 \ldots a_n\}$ use a common set of use cases $\{uc_1, uc_2 \ldots uc_n\}$. In order to formally write this condition, we define an auxiliary function that returns all the actors associated with a given use case. This function can be written as

$$allUC : \mathbb{A} \rightarrow \mathbb{U} - set$$

$$allUC(a) \equiv$$

$$\{(uc_i) |$$

$$(uc_i) : UseCase \,.$$

$$(\exists\, m : Association \,.$$

$$m \in \mathbb{M} \wedge subject(m) = a \wedge$$

$$use(m) \in uc_i)\}$$

Then the precondition can be written as

$$\forall\, a : Actor \in \{a_1, a_2 \ldots a_n\} \,.$$

$$a \in \mathbb{a} \wedge \forall\, a_i, a_j : \in \{a_1, a_2 \ldots a_n\} \wedge$$

$$allUC(a_i) = allUC(a_j)$$

(ii) There is no actor relationship among actors $\{a_1, a_2 \ldots a_n\}$, and any other actor does not reference them.

$$\exists\, a_1, a_2 : Actors \,.$$

$$a_1, a_2 \in \mathbb{a} \wedge a_1 \neq a_2 \wedge$$

$$\nexists\, rel \in \mathbb{g}. super(rel) = a_1 \wedge sub(rel) = a_2$$

(iii) The name of the new super actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1 : Actor \,.$$

$$a_1 \in \mathbb{a} \implies name(a_1) \neq newActor$$

Post-conditions:

(i) A new actor is created and it is the parent of actors $\{a_1, a_2 \ldots a_n\}$.

$$\exists\, a \in \mathbb{a}' \wedge a \notin \mathbb{a} \wedge name(a) = newActor \wedge$$

$$\forall\, a \in \{a_1, a_2 \ldots a_n\}\ .$$

$$\exists\, rel \in \mathbb{g}' \wedge rel \notin \mathbb{g} \wedge super(rel) = newActor \wedge sub(rel) = a$$

(ii) The new actor has association relationship with use cases $\{uc_1, uc_2 \ldots uc_n\}$.

$$\forall\, uc \in \{uc_1, uc_2 \ldots uc_n\}.$$

$$\exists\, m \in \mathbb{M}' \wedge m \notin \mathbb{M} \wedge subject(m) = newActor \wedge use(m) = uc$$

(ii) Association relationships between use cases $\{uc_1, uc_2 \ldots uc_n\}$ and actors $\{a_1, a_2 \ldots a_n\}$ are removed. Actors inherit these relationships from the parent actor newActor.

$$\forall\, a \in \{a_1, a_2 \ldots a_n\}.$$

$$allUC(a) \not\subset \{uc_1, uc_2 \ldots uc_n\}$$

Mechanism &Verification:

A generalization relationship between actors means that the child actors participate in all relationships of the parent actor. All common use cases are associated with the new parent actor and are removed from the child actors.

No new association between actors and use cases are added. Actors $\{a_1, a_2 \ldots a_n\}$ inherit association relationships between newActor and use cases $\{uc_1, uc_2 \ldots uc_n\}$. Hence all interactions between actors and use cases are preserved. Precondition (ii) ensures that actors $\{a_1, a_2 \ldots a_n\}$ are isolated from other actors so that the newActor does not affect

other actors. Precondition (iii) ensures distinct entity name for the newly added parent actor.

Implementation:

- Method Name: create_ActorGeneralization
- Arguments: ArrayList s*ubActorNames*, String *superActorName* where
    - *subActorNames*are the names of the child actors
    - *superActorName*is the name of the parent actor
- Return Value: String *status*

7. Merge Use Cases

Description: This refactoring merges two independent use cases that are used by the same set of actors. This refactoring helps manage the use case granularity by avoiding fragment use cases.

Origin: From Rui [286] [page 152]

Parameters: Use case $uc_1$and uc_2

Preconditions:

(i) Use cases uc_1 and uc_2 are not referenced by any use case.

$$\forall\, uc \in \{uc_1, uc_2\}.$$
$$uc \in \mathbb{U} \wedge$$
$$\forall\, rel: Relationship.$$
$$rel \in \mathbb{r} \wedge including(rel) \neq uc \vee extended(rel) \neq uc \vee super\,(rel) \neq uc$$
$$\forall\, uc_i: Use\ Case.$$

$$uc_i \in \mathbb{U} \wedge uc \notin allIncluded(uc_i) \wedge uc \notin allExtended(uc_i)$$

(ii) Use cases uc_1 and uc_2 are used by the same set of actors.

$$\forall uc_1, uc_2 : UseCases \,.\; allActors(uc_1) = allActors(uc_2)$$

Post-conditions:

The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring keeps one use case and deletes the other one.

The precondition (i) ensures that the use cases {uc_1, uc_2} are isolated from other use cases. This ensures that merging them together does not affect the behavior of the use case model. Precondition (ii) ensures that the use cases are used by the same set of actors.

Implementation:

- Method Name: merge_UseCases
- Arguments: String *UC1*, String *UC2* where
    - *UC1* and*UC2* are the names of the use cases to be merged
- Return Value: String *status*

8. Merge Actors

Description: This refactoring merges two actors into one. This refactoring helps manage actors.

Origin: From Rui [286] [page 156]

Parameters: Actor $a_1$and a_2

Preconditions:

Actors a_1 and a_2 are not referenced by any other actor. However, actor a_2 can be the parent of actor a_1.

$$\exists\, a_1, a_2 : Actors\ .$$
$$a_1, a_2 \in \mathbb{a} \ \wedge\ a_1 \neq a_2 \wedge$$
$$\nexists\, rel \in \mathbb{g}\ .\ super(rel) \in \{a_1, a_2\} \vee\ sub(rel) \in \{a_1, a_2\}$$
$$\exists\, rel \in \mathbb{g}\ .\ super(rel) = a_2 \Longrightarrow sub(rel) = a_1$$

Post-conditions:

(i) Use case references by actor a_2 are used by the actor a_1.

$$\forall\, uc : \in allUC(a_2)\ .$$
$$\exists\, m \in \mathbb{M}' \wedge m\ \notin\ \mathbb{M}\ \wedge subject(m) = a_1 \wedge use(m) = uc$$

(ii) The actor a_2 is deleted.

$$a_2 \notin\ \mathbb{a}'$$

Mechanism &Verification: This refactoring keeps one actor and deletes the other one.

The precondition ensures that the actors$\{a_1, a_2\}$ are isolated from other actors. This ensures that merging them together does not affect the behavior of the use case model.

Implementation:

- Method Name: merge_Actors
- Arguments: String *A1*, String *A2* where
    - *A1* and*A2* are the names of the actors to be merged
- Return Value: String *status*

9. Merge Use Case Generalization

Description: This refactoring merges two use cases that are related to each other by generalization and the interaction of the parent use case is empty (abstract). This refactoring helps maintain the abstraction level of use cases.

Origin: From Rui [286] [page 146]

Parameters: Use Case uc_1 and its parent uc_2

Preconditions:

(i) There is a generalization relationship between use cases uc_1 and uc_2.

$$\exists\, uc_1, uc_2 \in \mathbb{U}$$

$$\exists\, rel \in \mathbb{r}\,.\; super(rel) = uc_2 \wedge sub(rel) = uc_1$$

(ii) The use cases uc_2 is not referenced by any other use case except uc_1.

$$\exists\, uc_2 \in \mathbb{U}$$
$$\forall\, rel{:}\, Relationship.$$
$$rel \in \mathbb{r} \wedge including(rel) \neq uc_2 \vee extended(rel) \neq uc_2 \vee (super\ (rel) = uc_2 \Rightarrow sub(rel) = uc_1)$$
$$\forall\, uc_i{:}\, Use\ Case.$$
$$uc_i \in \mathbb{U} \wedge uc_2 \notin allIncluded(uc_i) \wedge uc_2 \notin allExtended(uc_i)$$

Post-conditions:

(i) Use case uc_1 takes over all association relationships between use case uc_2 and its actors.

$$\exists\, uc_1, uc_2 \in \mathbb{U}'.$$
$$allActors(uc_2) \subseteq allActors(uc_1)$$

(ii) The generalization relationship between uc_1 and uc_2 is deleted.

$$\nexists\, rel \in \mathbb{r}'.\ super\ (rel) = uc_2 \wedge sub\ (rel) = uc_1$$

(ii) The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring merges the parent use case into the child use case.

The precondition (i) ensures a generalization relationship between the use cases. Precondition (ii) isolates the use case uc_2 from other use cases than uc_1. Since the use case

uc_2has an empty interaction, it can be merged into the use case uc_1. The interaction between the use case uc_2 and related actors is not changed. Hence behavior is preserved.

Implementation:

- Method Name: merge_UCGeneralization
- Arguments: String *subUC*, String *superUC* where
    - *subUC* is the name of the child use case
    - *superUC* is the name of the parent use case
- Return Value: String *status*

10. Merge Use Case Inclusion

Description: This refactoring merges two use cases that are related to each other by inclusion relationship. The included use case is merged into the base use case. This refactoring helps manage use case granularity and maintain the abstraction level of use cases.

Origin: From Rui [286] [page 147]

Parameters: Base Use Case uc_1 and included Use Case uc_2

Preconditions:

(i) There is an inclusion relationship between use cases uc_1 and uc_2. The use case uc_1 includes the use case uc_2.

$$\exists\, uc_1, uc_2 \in \mathbb{U}$$

$$\exists\, rel \in \mathbb{r}\,.\; addition(rel) = uc_2 \;\wedge including(rel) = uc_1$$

(ii) Use case uc_2 is not referenced by other use cases except uc_1.

$$\exists\, uc_2 \in \mathbb{U}$$

$$\forall\, rel{:}\, Relationship.$$

$$rel \in \mathbb{r} \;\wedge\; including(rel) \neq uc_2 \;\vee extended(rel) \;\neq uc_2 \;\vee super\;(rel) \neq uc_2$$

$$\vee\, sub\;(rel) \neq uc_2$$

$$\forall\, uc_i{:}\, Use\ Case.$$

$$uc_i \in \{\mathbb{U} - uc_1\} \;\wedge uc_2 \;\notin allIncluded(uc_i) \wedge uc_2 \;\notin allExtended(uc_i)$$

Post-conditions:

(i) The inclusion relationship between uc_1 and uc_2 is deleted.

$$\nexists\, rel \in \mathbb{r}'.\; addition(rel) \;= uc_2 \wedge including(rel) \;= uc_1$$

(ii) The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring merges the inclusion use case into the base use case at the point of inclusion.

The precondition (i) ensures an inclusion relationship between the use cases. Precondition (ii) isolates the use case uc_2 from other use cases than uc_1. Merging the included use case into its base use case does not alter the behavior of the use case model. Hence behavior is preserved.

Implementation:

- Method Name: merge_UCInclusion
- Arguments: String *incUC*, String *baseUC* where
    - *incUC* is the name of the inclusion use case
    - *baseUC*is the name of the base use case
- Return Value: String *status*

11. Merge Use Case Extension

Description: This refactoring merges two use cases that are related to each other by extension relationship. The extending use case is merged into the base use case. This refactoring helps manage use case granularity and maintain the abstraction level of use cases.

Origin: From Rui [286] [page 148]

Parameters: Base Use Case $uc_1$and extending Use Case uc_2

Preconditions:

(i) There is an extension relationship between use cases uc_1 and uc_2. The use case $uc_{2\ Push}$ extends the use case uc_1.

$$\exists\, uc_1, uc_2 \in \mathbb{U}$$

$$\exists\, rel \in \mathbb{r}\,.\; extension(rel) = uc_2 \wedge extended(rel) = uc_1$$

(ii) Use case uc_2 is not referenced by other use cases except uc_1.

$$\exists\, uc_2 \in \mathbb{U}$$
$$\forall\, rel: Relationship.$$
$$rel \in \mathbb{r} \wedge including(rel) \neq uc_2 \vee extended(rel) \neq uc_2 \vee super\,(rel) \neq uc_2 \vee sub\,(rel) \neq uc_2$$
$$\forall\, uc_i: Use\ Case.$$
$$uc_i \in \{\mathbb{U} - uc_1\} \wedge uc_2 \notin allIncluded(uc_i) \wedge uc_2 \notin allExtended(uc_i)$$

Post-conditions:

(i) The extension relationship between uc_1 and uc_2 is deleted.

$$\nexists\, rel \in \mathbb{r}'.\ extension(rel) = uc_2 \wedge extended(rel) = uc_1$$

(ii) The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring merges the extension use case into the base use case at the point of extension.

The precondition (i) ensures an extension relationship between the use cases. Precondition (ii) isolates the use case uc_2 from other use cases than uc_1. Merging the extension use case into its base use case does not alter the behavior of the use case model. Hence behavior is preserved.

Implementation:

- Method Name: merge_UCExtension
- Arguments: String *extUC*, String *baseUC* where
    - *extUC* is the name of the extension use case
    - *baseUC*is the name of the base use case
- Return Value: String *status*

12. Split Use Case

Description: This refactoring splits one use case into two use cases. This refactoring helps manage use case granularity.

Origin: From Rui [286] [page 159]

Parameters: Use Case *uc* and String *newUC*

Preconditions:

(i) The use case uc is not referenced by any other use case.

$$\exists\, uc \in \mathbb{U}$$
$$\forall\, rel{:}\, Relationship.$$
$$rel \in \mathbb{r} \wedge including(rel) \neq uc \vee extended(rel) \neq uc \vee super\,(rel) \neq uc \vee sub\,(rel) \neq uc$$
$$\forall\, uc_i{:}\, Use\ Case.$$
$$uc_i \in \mathbb{U} \wedge uc \notin allIncluded(uc_i) \wedge uc \notin allExtended(uc_i)$$

(ii) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .$$
$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

(i) The new use case newUC is created.

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC$$

(ii) The new use case newUC is used by all actors that have an association relationship with the use case uc.

$$\exists\, uc, newUC \in \mathbb{U}'.$$
$$allActors(uc) \subseteq allActors(newUC)$$

(iii) There is no use case relationship between uc and newUC.

$$\exists\, uc,\ newUC \in \mathbb{U}'$$
$$newUC \notin allIncluded(uc) \wedge newUC \notin allExtended(uc)$$

Mechanism &Verification: This refactoring splits one use case into two use cases. The new use case has no relationship with the split use case.

The precondition (i) ensures that the use case uc has no relationship with the other use cases so that splitting uc does not change the behavior of other use cases. Precondition (ii) ensures distinct entity name invariant.

Implementation:

- Method Name: split_UC
- Arguments: String *UC*, String *newUC* where
    - *UC* is the name of the use case to be used for splitting
    - *newUC* is the name of the new use case
- Return Value: String *status*

13. Split Actor

Description: This refactoring splits one actor into two actors. This refactoring helps manage granularity. It also improves reusability of the use case model.

Origin: From Rui [286] [page 166]

Parameters: Actor a and String newActor

Preconditions:

(i) The actor a interacts with one use case in the use case model.

$$\exists\, a \in \mathbb{a}$$
$$length\left(allUC(a)\right) = 1$$

(ii) The actor a has no actor relationship with any other actor.

$$\exists\, a{:}\, Actor\,.$$
$$\forall\, rel \in \mathbb{g}\,.$$
$$super\,(rel)\; \neq a\; \vee sub(rel) \neq a$$

(iii) The name of the new actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1 : Actor\ .$$
$$a_1 \in \mathbb{a} \implies name(a_1) \neq newActor$$

***Post-conditions**:*

(i) A new actor a' with name newActor is created.

$$\exists\, a' \in \mathbb{a}' \wedge a' \notin \mathbb{a} \wedge name(a') = newActor$$

(ii) The new actor newActor interacts with all use cases that the actor a interacts with.

$$\exists\, a, newActor \in \mathbb{a}'.$$
$$allUC(a) \subseteq allUC(newActor)$$

(iii) There is no actor relationship between a and a'.

$$\exists\, a, a' \in \mathbb{a}'.$$
$$\nexists\, rel \in \mathbb{g}'\ .$$
$$(super\ (rel) = a \vee sub(rel) = a') \vee (super\ (rel) = a' \vee sub(rel) = a)$$

Mechanism &Verification: This refactoring splits one actor into two actors. The new actor interacts with the same use cases uc that the old actor interacts with. The interaction between actor a and the use case u is preserved by the interaction between the actor a' and the use case u. Hence behavior is preserved.

The precondition (i) ensures that actor a interacts with only one use case. This simplifies the definition if the refactoring. Precondition (ii) ensures that the actor a has no

relationship with the other actors so that splitting a does not change the behavior of other actors. Precondition (iii) ensures distinct entity name invariant.

Implementation:

- Method Name: split_Actor
- Arguments: String *Actor*, String *newActor* where
    - *Actor* is the name of the actor to be used for splitting
    - *newActor* is the name of the new actor
- Return Value: String *status*

14. Use Case Generalize Generation

Description: This refactoring splits one use case into two and creates a generalization relationship between two use cases. This refactoring helps manage use case granularity. It is a special case of the "Generalize Use Case" refactoring.

Origin: From Rui [286] [page 161]

Parameters: Use case *uc*, String *newUC*

Preconditions:

(i) The use case is not referenced by any other use case.

$$\forall\, uc : UseCase \;.\; uc \in \mathbb{U} \;\wedge\; allIncluded(uc) = \{\} \wedge allExtended(uc) = \{\}$$

(ii) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .$$

$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

A new use case is created and it is the parent of the use case uc.

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC \wedge$$

$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge super(rel) = newUC \wedge sub(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the parent or super use cases of the given use case uc. In the behavioral view, common interaction fragment is moved to this use case.

The precondition (i) ensures that the use case is isolated. Precondition (ii) ensures distinct entity name for the newly added parent use case.

Implementation:

- Method Name: generate_UCGeneralization
- Arguments: String *subUCName*, String *superUCName* where
    - *subUCName* is the names of the child use case
    - *superUCName* is the name of the parent use case
- Return Value: String *status*

15. Use Case Inclusion Generation

Description: This refactoring splits one use case into two and creates an inclusion relationship between the two use cases. This refactoring helps manage use case granularity and reduce redundancy.

Origin: From Rui [286] [page 162]

Parameters: Use case *uc*, String *newUC*

Preconditions:

(i) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .\quad uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

(i) A new use case uc' with the name newUC is created.

$$\exists\, uc' \in \mathbb{U}' \wedge uc' \notin \mathbb{U} \wedge name(uc') = newUC$$

(ii) The use case uc includes the newly created use case uc'

$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge addition(rel) = uc' \wedge including(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the inclusion use case of the given base use case uc. The precondition (i) ensures distinct entity name for the newly added inclusion use case.

Implementation:

- Method Name: generate_UCInclusion
- Arguments: String *baseUC*, String *newUC* where
    - *baseUC* is the name of the base use case
    - *newUC* is the name of the inclusion use case
- Return Value: String *status*

16. Use Case Extension Generation

Description: This refactoring splits one use case into two and creates an extension relationship between the two use cases. This refactoring helps manage use case granularity and reduce redundancy.

Origin: From Rui [286] [page 163]

Parameters: Use case *uc*, String *newUC*

Preconditions:

(i) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$\forall\, uc_1 : Use\ Case\ .$

$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

(i) A new use case uc' with the name newUC is created.

$$\exists\, uc' \in \mathbb{U}' \wedge uc' \notin \mathbb{U} \wedge name(uc') = newUC$$

(ii) The newly added use case uc' extends the use case uc.

$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge extension(rel) = uc' \wedge extended(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the extension use case of the given base use case uc. The precondition (i) ensures distinct entity name for the newly added extensionuse case.

Implementation:

- Method Name: generate_UCExtension
- Arguments: String *baseUC*, String *newUC* where
    - *baseUC* is the name of the base use case
    - *newUC* is the name of the extension use case
- Return Value: String *status*

17. Actor Generalize Generation

Description: This refactoring splits one actor into two and creates a generalization relationship between the two actors. This refactoring helps manage improve the understandability and reusability of the use case model.

Origin: From Rui [286] [page 168]

Parameters: Actor *a*, String *newActor*

Preconditions:

(i) The actor does not have a parent actor.

$$\forall\, a{:}\,Actor\ .$$
$$a \in \mathbb{a} \ \wedge$$
$$\forall\, rel \in \mathbb{g}\ .\ sub(rel) \neq a$$

(ii) The name of the new actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1{:}\,Actor\ .$$
$$a_1 \in \mathbb{a} \implies name(a_1) \neq newActor$$

Post-conditions:

(i) A new actor a' with the name newActor is created and it is the parent of actors a.

$$\exists\, a' \in \mathbb{a}' \wedge a' \notin \mathbb{a} \wedge name(a') = newActor \ \wedge$$
$$\exists\, rel \in \mathbb{g}' \wedge rel \notin \mathbb{g} \wedge super(rel) = a' \wedge sub(rel) = a$$

Mechanism &Verification: A new actor is created and is assigned as the parent or super actor of the given actor a. The precondition (i) ensures unique parent. Precondition (ii) ensures distinct entity name for the newly added parent actor.

Implementation:

- Method Name: generate_ActorGeneralization
- Arguments: String Actor, String newActor where

- *Actor* is the names of the actor used for splitting
- *newActor* is the name of the new parent actor

- Return Value: String *status*

A2.2 Class Model Refactoring

1. Pull Up Attribute

Description: This refactoring removes one attribute from a class or a set of classes and inserts it into one of its superclasses. It is the analogous to Fowler et al.'s Pull Up Attribute for Code Refactoring. If you pull up an attribute, the new visibility should be set to the maximum visibility of this attribute in the subclasses. At least all subclasses should still have access to the attribute after refactoring.

Origin: From Mantz [461] [page 95]

Parameters: String *superClass*, String *attr*

Preconditions:

(i) The attribute ($attr$) is owned by the same type by all classes that has the super class ($superClass$) as their parent class.

$$\forall\, g_1 : Generalization\ .$$

$$\exists\, atrib : name(atrib) = attr.$$

$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; atrib \in \boldsymbol{Attr}\big(sub(g_1)\big) \wedge type(atrib) = type\big(\boldsymbol{Attr}\big(sub(g_1)\big)\big)$$

(ii) The super class ($superClass$) must not have an attribute with the same name.

$$\forall\, g_1 : Generalization\ .$$
$$\exists\, atrib : name(atrib) = attr.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; atrib \notin \boldsymbol{Attr}\big(super(g_1)\big)$$

Post-conditions:

(i) The super class ($superClass$) has an attribute with the same name and type as the attributes in the subclasses.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \in \boldsymbol{Attr}\big(super(g_2)\big)$$

(ii) The child classes of the super class ($superClass$) has no attribute with the name ($attr$).

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \notin \boldsymbol{Attr}\big(sub(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the pulling up of the attribute. Based on the laws of inheritance, these attributes can still be accessed from the super class and since the attribute visibility is changed to the maximum (either public or protected); they can be accessed from the subclasses without any restriction.

Implementation:

- Method Name: pullup_Attribute
- Arguments: String *superClass* , String *attr* where
    - *superClass* is the name of the parent class

- *attr* is the name for the attribute to be pulled into the parent class

- Return Value: String *status*

2. Pull Up Method

Description: This refactoring moves a method of a class to its super class. Usually this refactoring is used simultaneously on several classes which inherit from the same super class. The aim of this refactoring is often to extract identical methods. This refactoring is analogous to Fowler et al.'s Pull Up Method for Code Refactoring. In order to keep the view consistent, Pull Up Method is often used with Pull Up Attribute. In most cases, it is also important that the operation is still visible in the subclass after refactoring Pull Up Method.

Origin: From Mantz [461] [page 106]

Parameters: String *superClass*, String *method*, ArrayList *signature*

Preconditions:

(i) The super class ($superClass$) must not have a method with the same name and signature.

$$\forall\, g_1 : Generalization\ .$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

(ii) All the sub classes of the parent ($superClass$) must have a method with the same name and signature.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \in \boldsymbol{Op}\big(sub(g_1)\big) \wedge same\_param(Param(op), Param\,\big(\boldsymbol{Op}\big(sub(g_1)\big)\big))$$

In the above precondition, we define an auxiliary function $same\_param$ that checks whether the parameters (also known as method signature) of the methods are same. This Function can be formally written as

$$same\_param\big(Param(op_1), Param(op_2)\big) \equiv$$
$$length\big(Param(op_1)\big) = length\big(Param(op_2)\big)$$
$$\Longrightarrow (\exists i : i \leq length\,\big(Param(op_1)\big) \wedge type\big(Param(op_1)\big) = type\big(Param(op_2)\big))$$

Post-conditions:

(i) The super class ($superClass$) has a method with the same name and signature as the method in the subclasses.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(ii) The child classes of the super class ($superClass$) has no method with the name ($method$) and signature.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \notin \boldsymbol{Op}\big(sub(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the pulling up of the method. Based on the laws of inheritance, this method can still be accessed from the super class and since the method visibility is changed to the maximum (either public or protected); it can be accessed from the subclasses without any restriction.

Implementation:

- Method Name: pullup_Method
- Arguments: String *superClass* , String *method,* ArrayList *signature* where
    - *superClass* is the name of the parent class
    - *method* is the name for the method to be pulled into the parent class
    - *signature* is the parameter list of the method to be pulled
- Return Value: String *status*

3. Push Down Attribute

Description: Refactoring Push Down Attribute moves an attribute to all subclasses. In the literature, refactoring Push Down Attribute is often limited to subclasses that require the attribute. In case of code refactoring these classes can be indicated. In case of UML models this is usually not possible, but it can be nevertheless useful to push down a property to all subclasses e.g. as preparation before deleting the superclass.

Origin: From Mantz [461] [page 109]

Parameters: String *superClass*, String *attr*

Preconditions:

(i) No direct subclass contains an attribute with the same name as the attribute that is being pushed down.

$$\forall\, g_1 : Generalization\ .$$

$$\exists\, atrib : name(atrib) = attr.$$

$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; atrib \notin \boldsymbol{Attr}\big(sub(g_1)\big)$$

Post-conditions:

(i) The attribute $(attr)$ is defined in all subclasses.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \in \boldsymbol{Attr}\big(sub(g_2)\big)$$

(ii) The attribute $(attr)$ does not exist in the super class any more.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \notin \boldsymbol{Attr}\big(super(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the pushing down of the attribute. Precondition (i) ensures that the attribute is not overwritten in the sub classes. Any subclass not using the attribute can be later deleted.

Implementation:

- Method Name: pushdown_Attribute
- Arguments: String *superClass* , String *attr*, where
    - *superClass* is the name of the parent class
    - *attr* is the name for the attribute to be pushed down into the child classes

- Return Value: String *status*

4. Push Down Method

Description: The refactoring Push Down Method pushes a method down to all its subclasses. It is analogous to Fowler et al.'s refactoring Push Down Method. In the literature, the Push Down Operation refactoring is often limited to subclasses that really require the operation. In case of code refactoring these classes can be indicated. However, in case of UML models the necessity of pushing down an operation can usually not be automatically construed (a possible solution is to inspect sequence diagrams).

Origin: From Mantz [461] [page 115]

Parameters: String *superClass*, String *method*, ArrayList *signature*

Preconditions:

(i) The super class ($superClass$) has subclasses.

$$\forall\, g_1 : Generalization\ .$$

$$g_1 \in \mathbb{R} \Rightarrow super(g_1) = superClass$$

(ii) The method ($method$) does not exist in any direct subclass.

$$\forall\, g_1 : Generalization\ .$$

$$\exists\, op : name(op) = method.$$

$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(sub(g_1)\big)$$

Post-conditions:

(i) The method ($method$) does not exist anymore in the super class ($superClass$).

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \ \wedge op \notin \boldsymbol{Op}(super(g_2))$$

(ii) The method ($method$) exists in all subclasses.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \ \wedge op \in \boldsymbol{Op}(sub(g_2))$$

Mechanism &Verification: The behavior of the class model is not affected with the pushing down of the method. Precondition (ii) ensures that the method is not overwritten in the sub classes. Any subclass not using the method can be later deleted.

Implementation:

- Method Name: pushdown_Method
- Arguments: String *superClass* , String *method*, ArrayList *signature* where
    - *superClass* is the name of the parent class
    - *method* is the name for the method to be pushed into the child classes
    - *signature* is the parameter list of the method to be pushed
- Return Value: String *status*

5. Remove Empty Superclass

Description: A set of classes has an empty super class which shall be removed. This refactoring often follows Push Down Attribute and Push Down Method Refactoring or in

the intermediate version also by the Pull Up Attribute or Pull Up Method Refactoring. In the intermediate version of this refactoring the empty super class inherits from a super class.

Origin: From Mantz [461] [page 112]

Parameters: String *superClass*

Preconditions:

(i) The super class ($superClass$) has no attributes and methods (it should be empty).

$$\forall\, g_1 : Generalization\ .$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

Post-conditions:

(i) The super class ($superClass$) does not exist anymore.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(ii) All classes still inherit all operations and attributes of potential super classes of the ($superClass$)

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \notin \boldsymbol{Op}\big(sub(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the deletion of the superclass. Precondition (i) and (ii) ensures that the class is empty and isolated from other attribute and method references. Precondition (iii) ensures that no

behavior is lost with the refactoring as the super class was an abstract class. Postcondition (ii) ensures that any inheritance relationship that exists between the superClass and other classes (i.e. the deleted super class was a sub class to other super classes) is preserved as these features are inherited in all the sub classes.

Implementation:

- Method Name: remove_SuperClass
- Arguments: String *superClass* where
    - *superClass* is the name of the parent class to be removed
- Return Value: String *status*

6. Remove Empty Subclass

Description: Refactoring Remove Empty Subclass removes an empty subclass from the model.

Origin: From Mantz [461] [page 99]

Parameters: String subClass

Preconditions:

(i) The subclass ($subClass$) has no attributes and methods (it should be empty).

$\forall\ g_1 : Generalization\ .$

$\quad \exists\ op : name(op) = method.$

$$g_1 \in \mathbb{R} \wedge sub(g_1) = subClass \Rightarrow op \notin \boldsymbol{Op}(sub(g_1))$$

Post-conditions:

(i) The subclass ($subClass$) and its inheritance relation do not exist anymore.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}(super(g_2))$$

Mechanism &Verification: The behavior of the class model is not affected with the deletion of the subclass. Precondition (i) ensures that the class is empty and isolated from other attribute and method references.

Implementation:

- Method Name: remove_SubClass
- Arguments: String *subClass* where
    - *subClass* is the name of the child class to be removed
- Return Value: String *status*

7. Create Super Class

Description: Refactoring Create Super Class is used to create a super class for at least one class which is normally followed by Pull Up Attribute and Pull Up Method Refactorings. In addition, this refactoring can create an intermediate super class that is a super class that is introduced between a set of classes and their former super classes.

Origin: From Mantz [461] [page 103]

Parameters: String newClass, ArrayList subClasses, Boolean abstract_flag, Boolean intermediate

Preconditions:

(i) The class name for the new super class ($newClass$) must be unique.

$$\forall\, g_1 : Generalization\ .$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

(ii) In the case that the ($intermediate$) flag is true, the classes within the selected set of classes ($subClasses$) must have at least one common super class.

$$\forall\, g_1 : Generalization\ .$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

Post-conditions:

(i) The new Class ($newClass$) exists in the Class Model.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(ii) There exists an inheritance relation to the super class ($newClass$) for each input class in the set ($subClasses$).

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(iii) If the super class ($newClass$) is an intermediate one, it must inherit from all common super classes of the selected set of subclasses ($subClasses$). Furthermore, there is no direct relation anymore between these super classes and the classes of this set ($subClasses$).

$$\exists\ g_2 \in\ \mathbb{R}' \wedge super(g_2) = superClass\ \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the creation of the new super class. Precondition (i) ensures that the new class is unique to the class model. In case the new class is an intermediate class between an existing inheritance, Precondition (ii) and Postcondition (iii) ensure that the new class inherits from all the common superclasses of the set of subclasses and that these subclasses have no more direct access to the superclasses.

Implementation:

- Method Name: create_SuperClass
- Arguments: String *newClass*, ArrayList *subclasses*, Boolean *isabstract*, Boolean *intermediate* where
    - *newClass* is the name of the new class to be created
    - *subClasses* is the set of classes that will be child classes to the newly created super class
    - *isabstract* is a flag that identifies whether the newly created flag is set to either abstract or concrete.

- *intermediate* is a flag which is set when the newly created flag is an intermediate class in an existing inheritance relationship.

- Return Value: String *status*

A2.3 Sequence Model Refactoring

1. Create Lifeline

Description: Create Lifeline Refactoring is used to introduce a new lifeline into a sequence diagram.

Origin: From Meng and Barbosa [462]

Parameters: String *newLifeline*

Preconditions:

(i) The lifeline ($newLifeline$) must be unique in the sequence diagram.

$$\forall\, l_1 : Lifeline\ .\quad l_1 \in \mathbb{L} \implies name(l_1) \neq newLifeline$$

Post-conditions:

$$\exists\, l \in \mathbb{L}' \wedge l \notin \mathbb{L} \wedge name(l) = newLifeline$$

Mechanism &Verification: The behavior of the sequence model is not affected with adding a new lifeline since there is no message exchanges between the new lifeline and

the exisiting lifelines within the sequence diagram. The precondition (i) ensures that the new lifeline is unique to the sequence model.

Implementation:

- Method Name: create_Lifeline
- Arguments: String *newLifeline* where
    - *newLifeline* is the name of the new lifeline to be added
- Return Value: String *status*

2. Remove Lifeline

Description: Refactoring Remove Lifeline is used to remove a lifeline that does not interact with other participants and has no local actions within the sequence diagram.

Origin: From Meng and Barbosa [462]

Parameters: String *Lifeline*

Preconditions: The lifeline is isolated from other particpants of the sequence diagram. Isolation from other participants means

- No message exchanges
- No local actions

$\exists\, l: SEQ\ .$

$$name(l) = Lifeline\ \wedge\ length\big(endList(l)\big) = 0$$

Post-conditions:

$l \notin \mathbb{L}'$

Mechanism &Verification: Since the lifeline is isolated from other participating lifelines within the sequence diagram, it does not affect interactions between them. Hence, this deletion does not change the behavior of the sequence model.

Implementation:

- Method Name: remove_Lifeline
- Arguments: String *lifeline* where
    - *lifeline* is the name of the lifeline
- Return Value: String *status*