Catalogue of Model Transformations

K. Lano

Dept. of Computer Science, King's College London, Strand, London, WC2R 2LS, UK

This document presents a set of model transformations on UML class and state machine models, as a preliminary version of a comprehensive catalogue for use with UML development.

It is offered as a free resource to UML developers. Each transformation is provided with an explanation of its purpose, examples of its use and conditions necessary for its correct use.

Model transformations in UML can be classified in five categories:

1. Refinements.
2. Quality improvement transformations.
3. Elaborations.
4. Abstractions.
5. Design patterns.

1 Refinement Transformations

These transformations are used to refine a model towards an implementation. For example, PIM to PSM transformations in the Model-driven Architecture. They typically remove certain constructs or structures, such as multiple inheritance, from a model, and represent them instead by constructs which are available in the implementation platform.

1.1 Replacing inheritance by association

Description This transformation replaces an inheritance relationship between two classes by an association between the classes.

Purpose This transformation is useful when refining a PIM towards a PSM for a platform which does not support inheritance, such as the relational data model. It can also be used to remove multiple inheritance for refinement to platforms which do not support multiple inheritance.

Diagram Figure 1 shows the general structure of this transformation. The inheritance of B on A is replaced by a 0..1 to 1 association from B to A.

Correctness conditions Any expression in the original model which has B as contextual classifier, and which uses a feature f inherited from A, must be modified in the new model to use $ar.f$ instead.

The transformation is also used in [5] to improve the quality of models where inheritance would be misapplied, such as situations of dynamic and multiple roles. It is related to the Role pattern of [1].

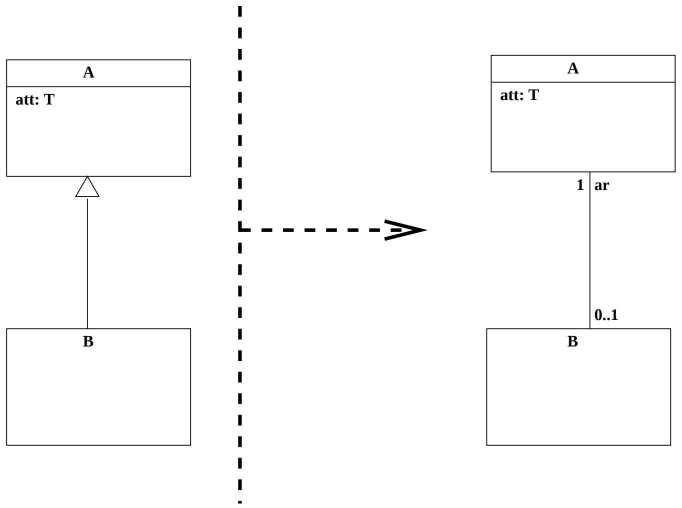

Fig. 1. Replace inheritance by association

1.2 Removal of many-many associations

Description This transformation replaces a many-many association with a new class and two many-one associations.

Purpose Explicit many-many associations cannot be implemented using foreign keys in a relational database – an intermediary table would need to be used instead. This transformation is the object-oriented equivalent of introducing such a table.

Diagram This is shown in Figure 2.

Conditions The new class must link exactly those objects that were connected by the original association, and must not duplicate such links:

$$c1 : C \ \& \ c2 : C \ \& \ c1.ar = c2.ar \ \& \ c1.br = c2.br \ \Rightarrow \ c1 = c2$$

In addition, any constraint with contextual classifier A or a subclass of A, which refers to br, must replace this reference by $cr1.br$ in the new model. Likewise for navigations from B to A.

1.3 Removal of association classes

Description This transformation replaces an association class with a new class and two associations.

Purpose Association classes cannot be represented directly in many object-oriented programming languages.

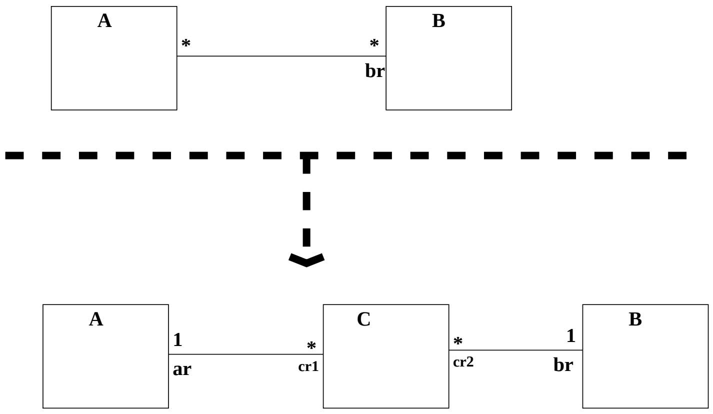

Fig. 2. Removing a many-many association

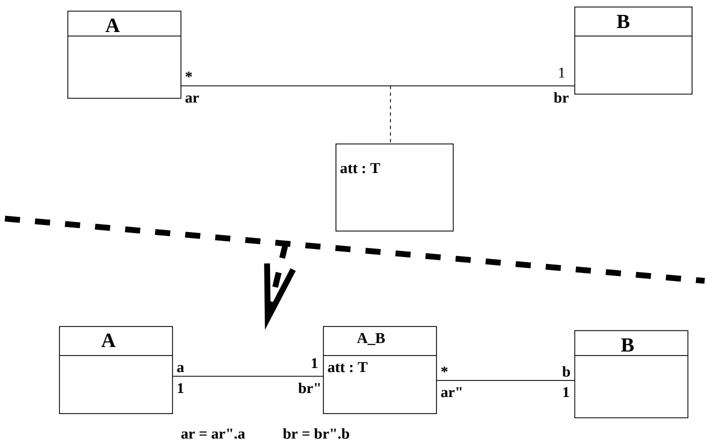

Fig. 3. Removing an association class

Diagram This is shown in Figure 3.

Conditions The new class must link exactly those objects that were connected by the original association, and must not duplicate such links:

$$c1 : A\_B \ \& \ c2 : A\_B \ \& \ c1.a = c2.a \ \& \ c1.b = c2.b \ \Rightarrow \ c1 = c2$$

Any constraint on C must be re-expressed as a constraint of the new class. Constraints of A which refer to br must now be re-expressed using $br''.b$, etc.

1.4 Removing qualified associations

Description A qualified association with qualifier $x : T$ to a class B is replaced with an ordinary association to B enhanced with $x : T$ as an additional attribute, plus additional constraints.

Purpose Qualified associations cannot be directly expressed in some programming languages, nor in relational databases.

Diagram Figure 4 shows the simple case where the qualified association has multiplicity 1 at the target end.

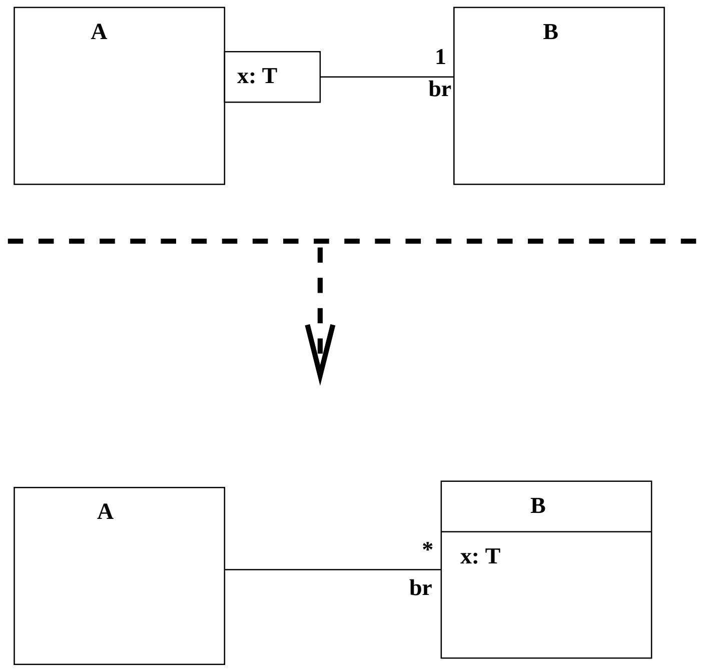

Fig. 4. Removing a qualified association

Conditions Constraints referring to $br[val]$ in the original model must be changed to use $br{\rightarrow}select(x = val)$ instead.

The constraints

$$br.x{\rightarrow}size() \;=\; card(T)$$

and

$$br.x{\rightarrow}size() \;=\; br{\rightarrow}size()$$

hold in the new model.

In the case that the multiplicity of the original association is $*$ at the B end, these constraints are replaced by:

$$w \neq v \;\Rightarrow\; br{\rightarrow}select(x = w) \cap br{\rightarrow}select(x = v) = \{\}$$

Ie., the sets of B objects identified by different qualified index values are disjoint.

1.5 Removing aggregations

Description This transformation replaces a strong (composition) aggregation by an association together with an additional constraint.

Purpose Aggregation cannot normally be represented directly in an OO programming language.

Diagram Figure 5 shows the general situation.

Conditions The new constraint

$$kill_A(a) \;\Rightarrow\; kill_B(br(a))$$

expresses the deletion-propogating nature of the aggregation. The class B should not be or become a supplier to any other class.

1.6 Weakening preconditions or strengthening postconditions

Description An operation precondition can be weakened (so that it is able to be applied in more situations without error) and/or its postcondition strengthened (so that its effect is determined more precisely). Both potentially move the method closer to implementation.

Diagram Figure 6 shows a general situation. It should be borne in mind that the precondition of the operation in any interface used by a client will be the only precondition that such a client can assume. Working to a more general (weaker) precondition in a subclass is an issue of *defensive programming*: improving the robustness of the actual implementation of the operation so that even if the client does call it in an apparently invalid state (according to the interface), the operation will still have defined behaviour.

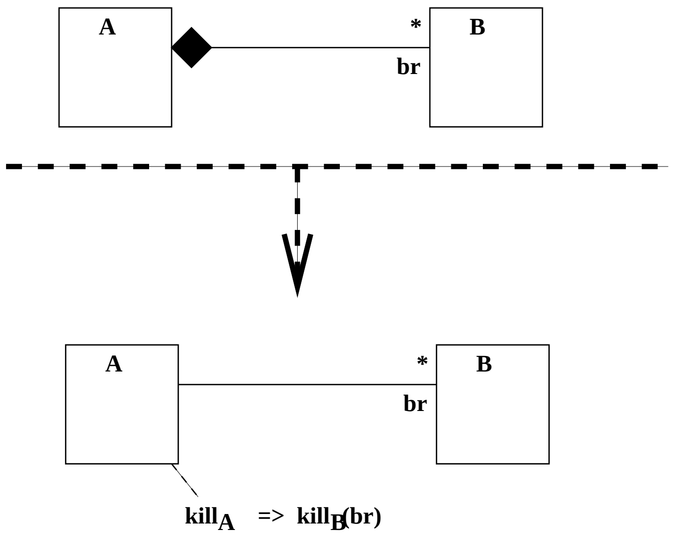

Fig. 5. Removing aggregation

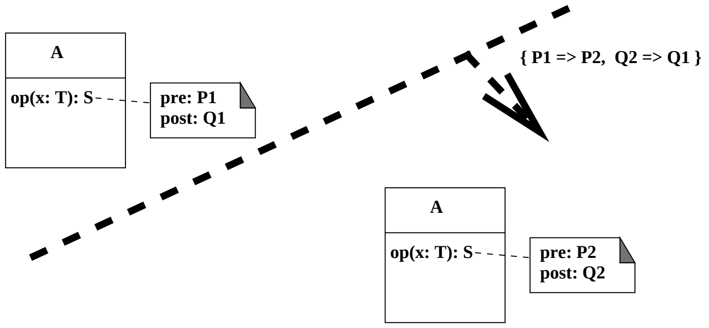

Fig. 6. Weakening preconditions/strengthening postconditions

1.7 Replace association by attribute

Description In UML 2.0, attributes of a class are equivalent to certain forms of association from the class. Embedding an association into a class as an attribute can be used as a step towards implementing it as an attribute of the class.

Diagram Figure 7 shows the equivalence of associations and attributes. Multiplicities,

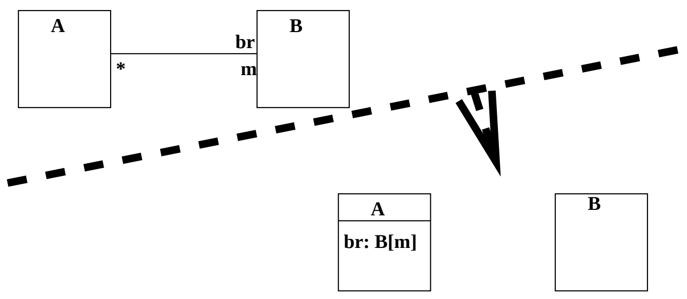

Fig. 7. Equivalence of attributes and associations

visibilities, the {*ordered*} and {*unique*} constraints, and the indication / that the feature is derived, can all be mapped from the association notation into identical attribute notation.

Conditions Qualified associations cannot be represented as attributes, as there is no corresponding attribute notation, ie, for specifying map data structures. In addition the multiplicity at the owning entity end of an association cannot be generally expressed in the attribute version. For an ordinary attribute this multiplicity is $*$ and for an identity attribute (primary key) it is 0..1.

1.8 Introduce primary key

Description This transformation applies to any persistent class. If the class does not already have a primary key, it introduces a new identity attribute, of integer type, for this class, together with extensions of the constructor of the class, and a new *get* method, to allow initialisation and read access of this attribute.

Purpose This is an essential step for implementation of a data model in a relational database.

Diagram This is shown in Figure 8.

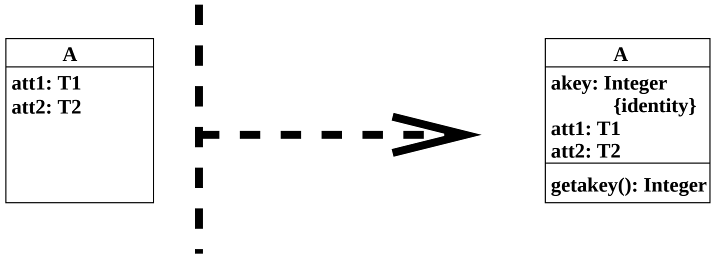

Fig. 8. Introduce primary key

Conditions A new constraint expressing the primary key property is added to the new model:

$$A.allInstances() \rightarrow size() = A.allInstances().akey \rightarrow asSet() \rightarrow size()$$

This must be maintained by the constructor, for example:

```
A(att1x : T1, att2x: T2, akeyx: Integer)
  pre: A.allInstances().akey->excludes(akeyx)
```

1.9 Replace association by foreign key

Description This transformation applies to any explicit many-one association between persistent classes. It assumes that primary keys already exist for the classes linked by the association. It replaces the association by embedding values of the key of the entity at the ‘one’ end of the association into the entity at the ‘many’ end.

Purpose This is an essential step for implementation of a data model in a relational database.

Diagram This is shown in Figure 9.

Conditions $b.akey$ is equal to $a.akey$ exactly when $a \mapsto b$ is in the original association. This correspondence must be maintained by implementing *addbr* and *removebr* operations in terms of the foreign key values.

Navigation from an A instance to its associated br set must be replaced by

$$B.allInstances() \rightarrow select(akey = self.akey)$$

in the new model. Likewise for navigation from B to A.

1.10 Amalgamate subclasses into superclass

Description This transformation amalgamates all features of all subclasses of a class C into C itself, together with an additional flag attribute to indicate which class the current object really belongs to.

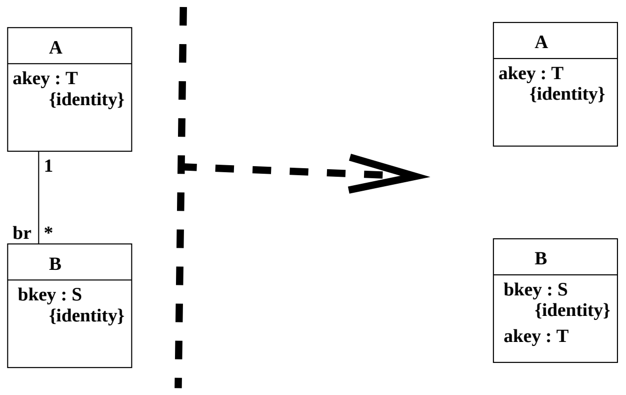

Fig. 9. Replacing association by foreign key

Purpose This is one strategy for representing a class hierarchy in a relational database. It is preferable to replacing inheritance by an association if the merged classes are quite similar and differ only in a few attributes. Otherwise the resulting relational database tables will contain many blank columns.

Diagram This is shown in Figure 10.

Conditions Constraints of the subclasses must be re-expressed as constraints of the amalgamated class, using the flag attribute, as illustrated in Figure 10.

Operations must also be redefined using explicit tests on *flag* instead of dynamic binding. For example an operation *op* defined in all three original classes would have the new postcondition definition

```
post: if (flag = isA)
      then PostA
      else if (flag = isB)
      then PostB
      else if (flag = isC)
      then PostC
```

where the respective original postconditions are *PostA*, etc.

This transformation is related to the Collapsing Hierarchy refactoring of [3].

1.11 Make association into index

Description This replaces an association *i* which identifies a member of an ordered role by an integer index.

Purpose The resulting data structures are simpler and more efficient to implement.

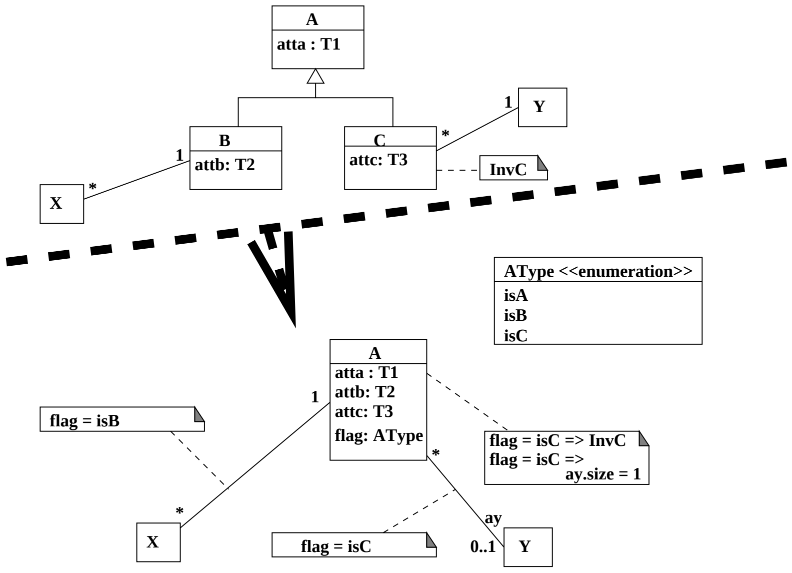

Fig. 10. Amalgamation of subclasses transformation

Diagram Figure 11 shows this transformation where i in the first model is implemented by $br[index]$ in the second.

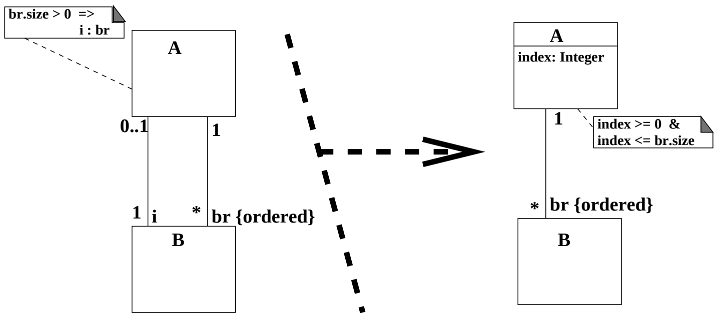

Fig. 11. Transforming index association into an attribute

Condition References to i in the constraints of the original model need to be replaced by

$$br \rightarrow at(index)$$

under an assumption $index > 0$ in the new model.

1.12 Source splitting

Description This transformation refines the behaviour of a state machine by introducing substates of a state s and splitting a transition from s into cases for each of these substates.

Purpose A simple behaviour may need to be refined into subcases, in particular if a new attribute or other structural feature is introduced in a class.

Diagram A simple case with two states is shown in Figure 12. Any number of new states and corresponding transitions can be introduced.

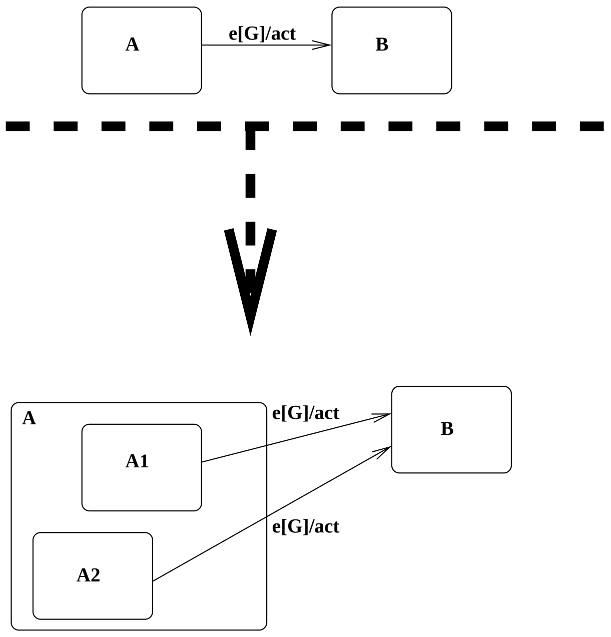

Fig. 12. Source splitting

Condition All new substates of s must be sources of new transitions derived from the transition of s. These new transitions can have additional postconditions/actions, but must have the same trigger and guard as the original transition. The targets of the new transitions can be substates of the original target.

This transformation is due to Cook and Daniels [2].

1.13 Target splitting

Description This transformation replaces a single transition in a state machine by two or more transitions distinguished by disjoint firing conditions, and with possibly distinct actions and target states.

Purpose This is used to refine behaviour by making distinct different cases which were amalgamated in the abstract model. It can be used to define a state machine for a subclass, so that the subclass state machine is behviourally compatible with the superclass machine.

Diagram Figure 13 shows the structure of this transformation in the case of a split into two transitions. One state of the source model is split into two, and a transformation into the state is also split in two, such that $G \equiv G1 \lor G2$ and $G1 \Rightarrow \neg\ G2$. Postconditions can be strengthened:

$$\begin{aligned} Post1 &\Rightarrow Post \\ Post2 &\Rightarrow Post \end{aligned}$$

Any number of new substates of t can be introduced.

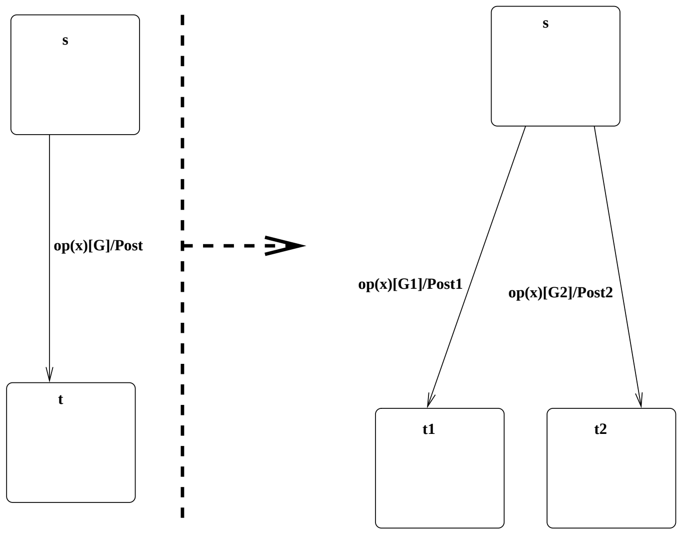

Fig. 13. Target splitting

This transformation is also due to Cook and Daniels [2].

1.14 Add orthogonal state machine

Description If a state machine SM of class C is made into a region in a concurrent composite state machine $SM1$ of a class D, with two regions, where the other region of $SM1$ has a disjoint set of trigger events from SM, and both regions have only call triggers, then $SM1$ refines SM.

Purpose This transformation corresponds to the definition of a subclass D of C which defines its own incremental behaviour independently of C and without modifying C's behaviour.

Diagram This is shown in Figure 14.

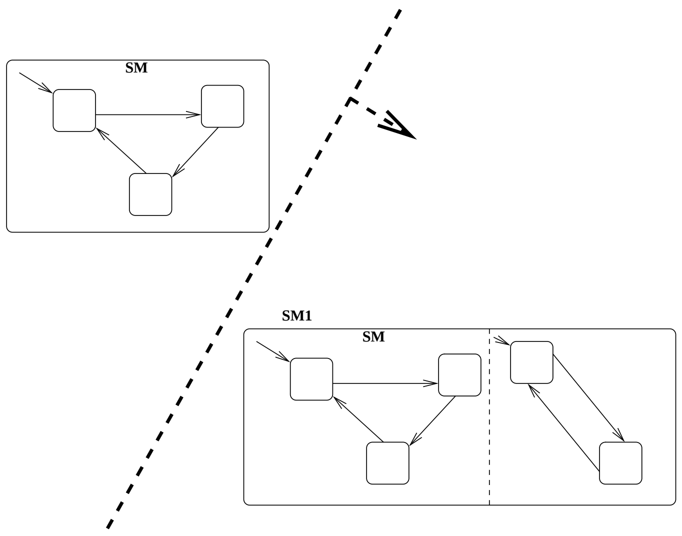

Fig. 14. Adding a region

1.15 Express state machine in pre/post constraints

Description This transformation expresses the protocol state machine SM of a class C, which only has call triggers on its transitions, as new data and pre/post conditions of C. A new enumerated type: $States_C$, is introduced, with an element for each basic state configuration of the statemachine, and an attribute $stateC : States_C$ of C, together with operation pre- and postconditions expressing the behaviour of all the state machine transitions.

If operation $\alpha(p)$ has transitions $t\alpha_1$, ..., $t\alpha_{n\alpha}$ from state configurations $s\alpha_1$, ..., $s\alpha_{n\alpha}$ to state configurations $p\alpha_1$, ..., $p\alpha_{n\alpha}$ with guards $G\alpha_1$, ..., $G\alpha_{n\alpha}$, then the precondition of α is augmented with the condition

$$(stateC = s\alpha_1 \ \& \ G\alpha_1) \ or \ ... \ or \ (stateC = s\alpha_{n\alpha} \ \& \ G\alpha_{n\alpha})$$

and the postcondition is augmented by conjuncts

$$(stateC@pre = s\alpha_i \ \& \ G\alpha_i@pre) \ \Rightarrow \ stateC = p\alpha_i$$

for $i = 1, ..., n\alpha$.

Each state invariant Inv_s of a state s becomes a new class invariant

$$stateC = s \ \Rightarrow \ Inv_s$$

and each attribute of SM becomes an attribute of C.

Purpose The encoding of the state machine as explicit data and updates to this can facilitate the generation of executable code to ensure that objects of the class obey the dynamic behaviour it describes.

1.16 Express state machine as role classes

Description This transformation expresses the protocol state machine *SM* of a class C, which only has call triggers on its transitions, as a hierarchy of new role classes attached to C.

A new supplier class *SM* is introduced, with a subclass for each topmost state of the statemachine, and these in turn have subclasses for each of their states. Concurrent state machines are represented as an aggregate of the classes representing each region. Each attribute of a state becomes an attribute of the corresponding class, and likewise for constraints. Internal actions of a state become operations of the corresponding class.

History states can be represented as additional associations recording the state object associated to an object of C at exit from the composite state containing the history state.

Purpose The representation of the state machine as classes makes its semantics explicit in terms of class diagram semantics.

Diagram Figure 15 shows a typical example of this transformation.

1.17 Flattening a state machine

Description This transformation removes composite states and expresses their semantics in terms of their substates instead: a transition from a composite state boundary becomes duplicated as a transition from each of the enclosed states (if they do not already have a transition for that event). A transition to the composite state boundary becomes a transition to its default initial state.

Purpose The transformation reduces the complexity of the constructs used to express dynamic behaviour, making this behaviour easier to verify, although the size of the model will be increased.

Diagram Figure 16 shows a typical case of elimination of a composite state, Figure 17 shows the elimination of a concurrent composite state. In this example the transformation uses a synchronisation semantics for transitions with the same trigger in two components of the same concurrent composite state: these transitions must synchronise. This is a semantic variation point in UML – semantics in which transitions for the same trigger can also occur independently are also possible, and can be expressed by a variation on this transformation.

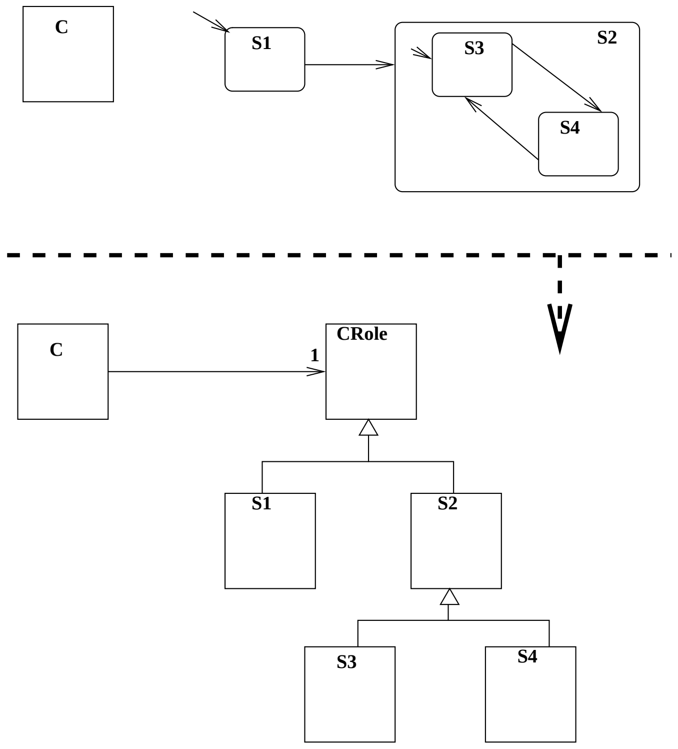

Fig. 15. Representing state machine as classes

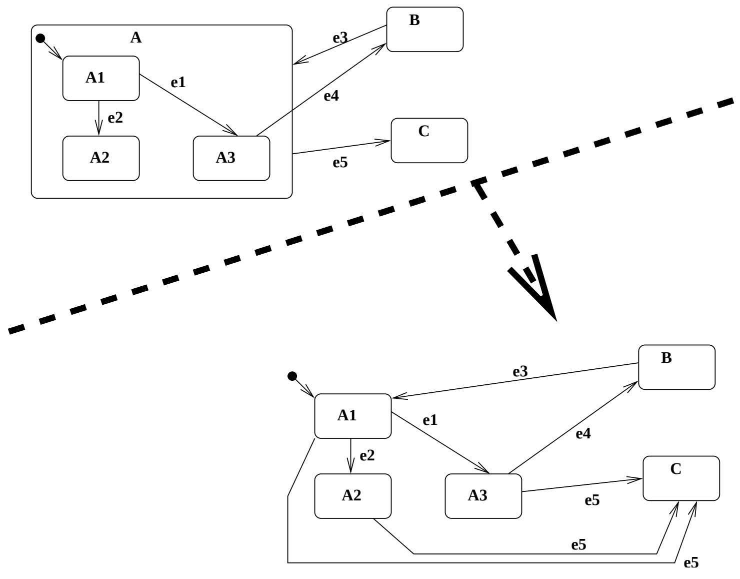

Fig. 16. Eliminating a composite state

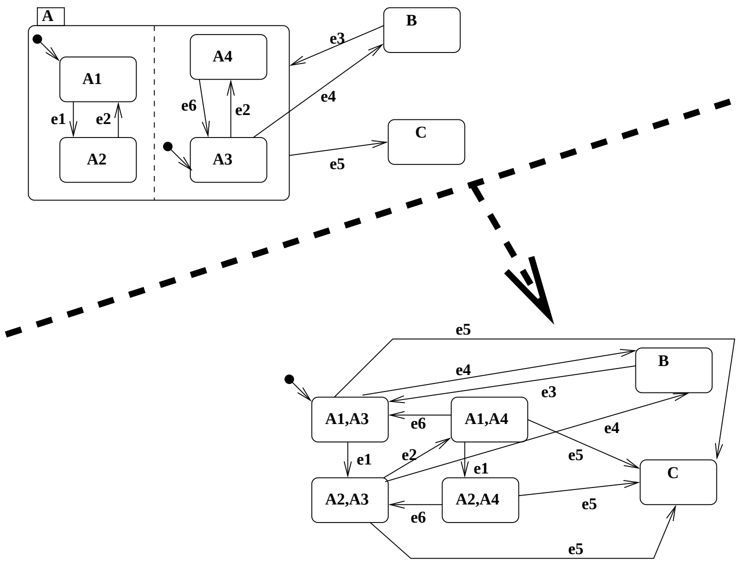

Fig. 17. Flattening a concurrent composite state

2 Quality Improvement Transformations

The transformations in this category aim to improve the structure of a model, making it conform more closely to normal uses of UML notation, or improving its precision or its flexibility for extension and adaption.

Two general categories of quality improvement transformation are the removal of redundancy and the factoring out/decomposition of elements.

The ‘introduce superclass’ transformation is one example of the removal of redundancies from a model. Normally redundancies should be eliminated – they complicate a model unnecessarily and, if implemented, lead to extra work and possibilities for program flaws. Other examples of such transformations are the removal of redundant inheritances and associations.

2.1 Introduce superclass

Description Introduces a superclass of several existing classes, to enable common features of these classes to be factored out and placed in a single location.

Purpose In general, this transformation should be applied if there are several classes A, B, ... which have common features, and there is no existing common superclass of these classes. Likewise if there is some natural generalisation of these classes which is absent in the model.

Diagram Figure 18 shows a generic example, where the existing classes have both common attributes, operations and roles.

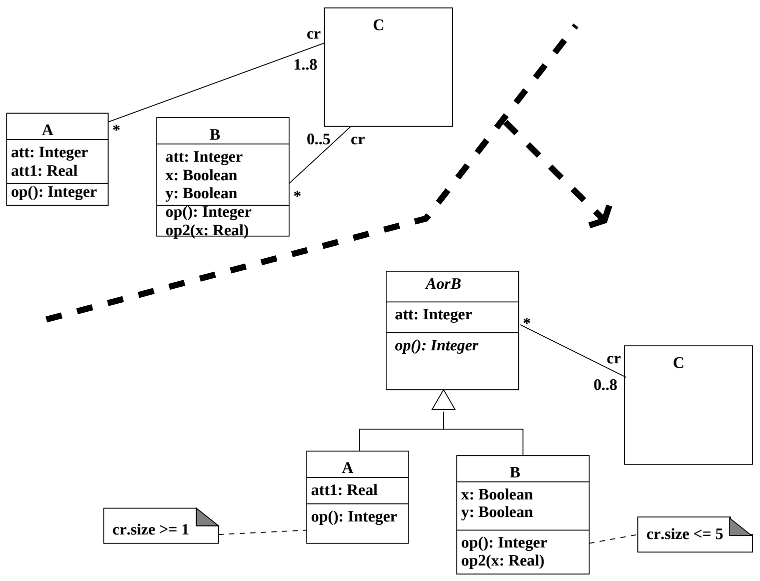

Fig. 18. General superclass introduction

Conditions The features that are placed in the superclass must have the same intended meaning in the different subclasses, rather than an accidental coincidence of names.

The properties of the features in the superclass are the disjunction of their properties in the individual subclasses. For common roles, this means that their multiplicity on the association from the superclass is the 'strongest common generalisation' of their multiplicities on the subclass associations. Eg, if the subclass multiplicities were $m1..n1$ and $m2..n2$, the superclass multiplicity would be $min(m1, m2)..max(n1, n2)$. For common operations, the conjunction of the individual preconditions can be used as the superclass operation precondition, and the disjunction of the individual postconditions as the superclass operation postcondition.

Common constraints of the subclasses can also be placed on the superclass.

Variations include situations where a common superclass already exists, but some common features of its subclasses are missing from it. In this case the common features are simply moved up to the superclass. The 'Pull up method' refactoring of [3] is one case of this situation.

2.2 Remove redundant association

Description An explicit association which can be computed as a derived association from other model elements is made implicit and derived, or eliminated completely.

Purpose Such associations duplicate information needlessly. In an implementation the overhead of maintaining consistency between the association and the information it duplicates could be considerable, and prone to errors.

Diagram Figure 19 shows a situation where the redundant association is the relational composition of two others, in Figure 20 it is the inverse of another association.

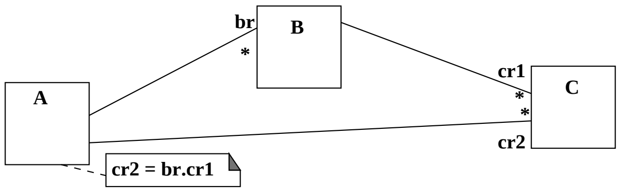

Fig. 19. Redundant composition association

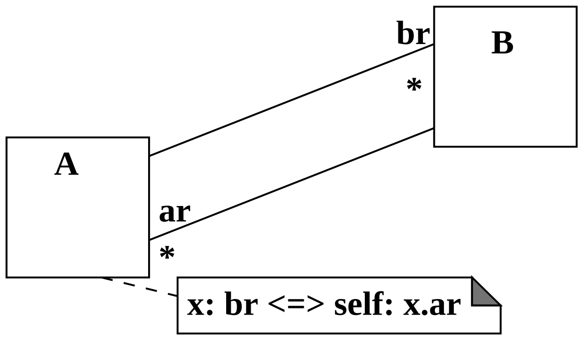

Fig. 20. Redundant inverse association

Conditions The removed explicit association must be expressible as a derived association, ie, there must be a simple rule for computing it in terms of other model elements. Constraints referring to the roles of the association must be re-expressed in terms of the expressions these roles are derived from.

In certain cases, eg, the Observer pattern, maintaining explicit inverse associations is a valid decision, however even in these cases it is preferable to avoid such bidirectional dependence and communication if possible, eg, by the *update* method providing all needed information to the observer objects directly, without the need for the observers to callback to the observable.

2.3 Eliminate redundant inheritance

Description If a class inherits another by two or more different paths of inheritance, remove all but one path, if possible.

Purpose A redundant inheritance conveys no extra information, but complicates the model.

Diagram Figure 21 shows a typical situation where class A directly inherits from class C, and also indirectly via a more specific class B. The first inheritance is redundant and can be removed. (In some languages, such as Java, such inheritances would actually be invalid. In UML, according to the 'object-oriented resolution' of operation behaviour, any operations defined in C potentially make the model ill-formed [6]).

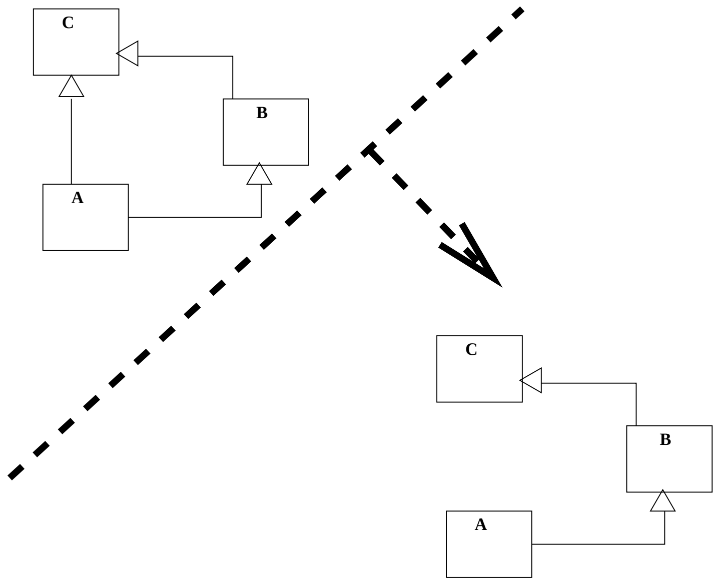

Fig. 21. Redundant inheritance removal

Conditions The removed inheritance, eg, of E inheriting from F, must be genuinely redundant, ie, there must exist another chain of inheritances from E to F via other intermediary classes.

2.4 Remove redundant classes

Description Classes may be redundant because they are essentially duplicates of other classes in the model but with a different name (synonyms), or because they are not needed in the system being defined.

Purpose Duplication of classes will lead to over-complex models which are difficult to modify and analyse.

Diagram Figure 22 shows a typical case where class *A* and class *B* are almost identical and can be replaced by a single class.

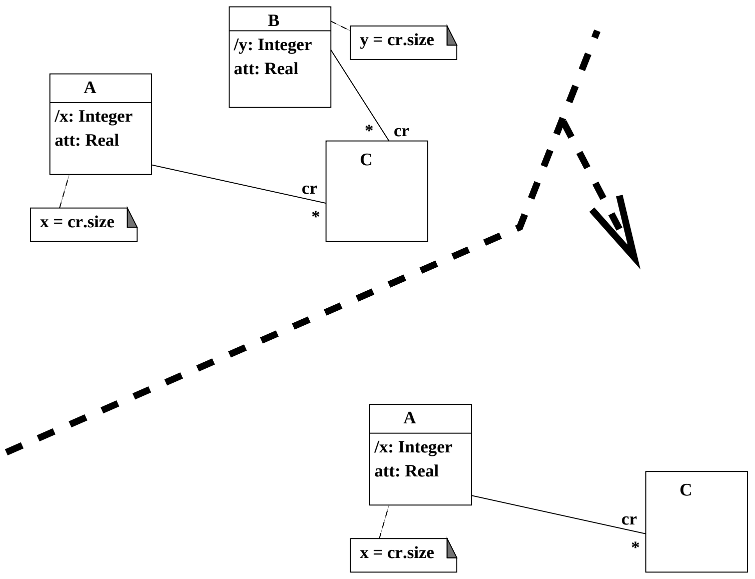

Fig. 22. Redundant class removal

Conditions The constraints of the removed class should be expressed equivalently in the new model, if they are required for the system.

A related case is when a class has been introduced as an intermediary delegate between two other classes but is later recognised as redundant, because no constraints refer to it, and any operations it has are carried out by delegation. This is the 'inline class' refactoring of [3].

2.5 Remove redundant transitions

Description In a state machine, a transition has priority over another if both are triggered by the same event, and the first has a source state which is a substate of the source of the second. Therefore if the two transitions have identical guard, trigger, target and actions, the higher-priority transition is redundant and can be removed from the diagram.

Purpose Duplication of transitions complicates the model unnecessarily.

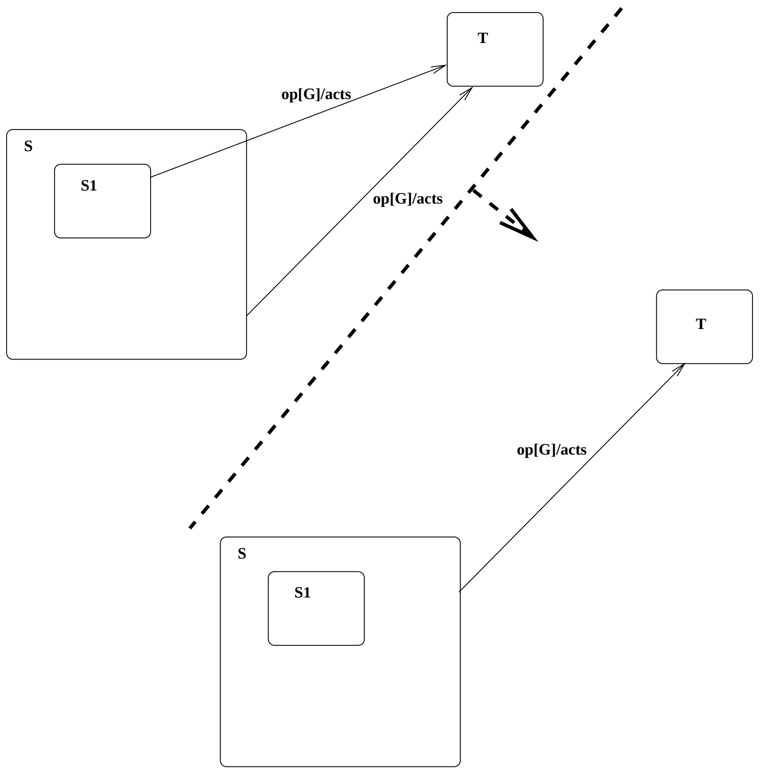

Fig. 23. Redundant transition removal

Diagram Figure 23 shows a typical case.

2.6 Merge partial classes

Description Classes may define only part of a coherent concept, other parts may be expressed in different classes and their commonalities have not been recognised. This transformation merges such classes into a single class.

Diagram Figure 24 shows a typical case where class *User* and class *Person* represent parts of the same concept and can probably be merged into a single class. The existence of 1-1 or 1-0..1 associations between classes is often an

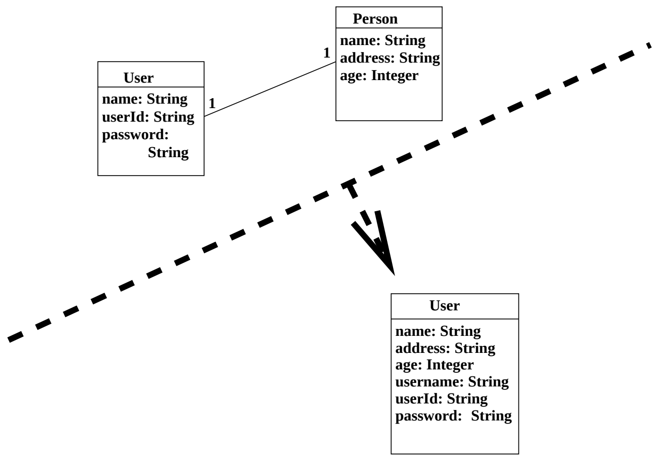

Fig. 24. Merging partial classes

indicator of unnecessary separation of a single concept into two classes (in the second case it may indicate a missing specialisation-generalisation relationship, with the superclass being at the 1 end of the association).

Another group of transformations improve a model by factoring out certain groups of features from a class into new classes or interfaces, or factoring common features of a set of states into a new superstate.

2.7 Introduce interface(s) for supplier class(es)

Description If class A is a client of class B, but only uses some of B's operations, introduce an interface B_I of B which has the subset of operations of B that are used by A. Make A a client of B_I instead of B.

Purpose This reduces the dependencies in the model and enables separate development of A and B, and permits them to be placed in different layers or tiers of the system.

Diagram Figure 25 shows this transformation.

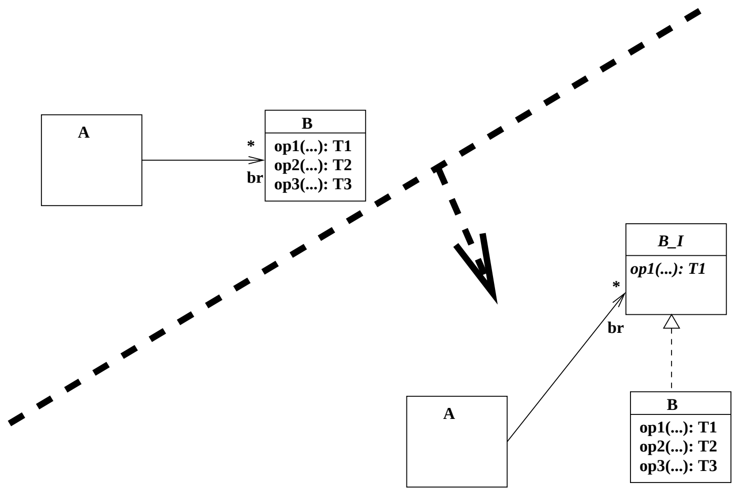

Fig. 25. Introducing an interface

Conditions A must only depend on the specifications of operations of B, not on their implementation.

An example where it is important to factor out different interfaces of a class for different clients is a password database (Figure 26): general users can only use the *check* and *setPassword* operations, whilst the system administrator can delete and create user records.

2.8 Disaggregation

Description A class is factored into component classes.

Purpose A class may become large and unmanageable, with several loosely connected functionalities. It should be split into several classes, such as a master/controller class and helper classes, which have more coherent functionalities and data.

Diagram Figure 27 shows a generic example.

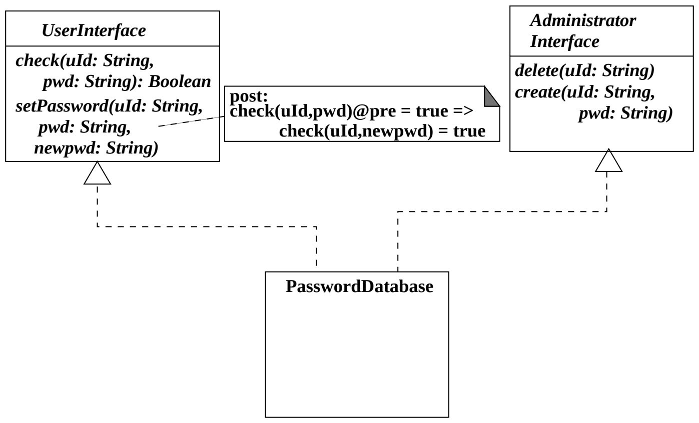
UserInterface
check(uId: String, pwd: String): Boolean
setPassword(uId: String, pwd: String, newpwd: String)
post: check(uId,pwd)@pre = true => check(uId,newpwd) = true
Administrator Interface
delete(uId: String)
create(uId: String, pwd: String)
PasswordDatabase

Fig. 26. Example of interface factoring

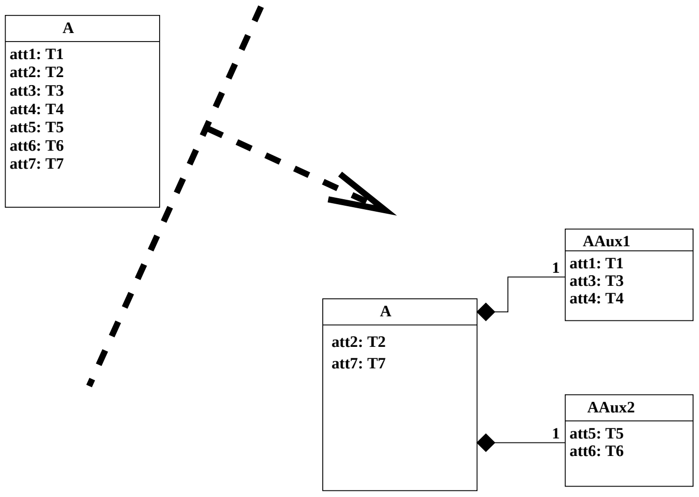

Fig. 27. Disaggregation

Conditions The helper/component objects always exist when the master object delegates operations to them. Constraints of A which refer to the attributes which have been placed in auxillary classes must replace the attribute reference by a suitable navigation expression.

Figure 28 shows an example of disaggregation for classes in a student accommodation agency web system.

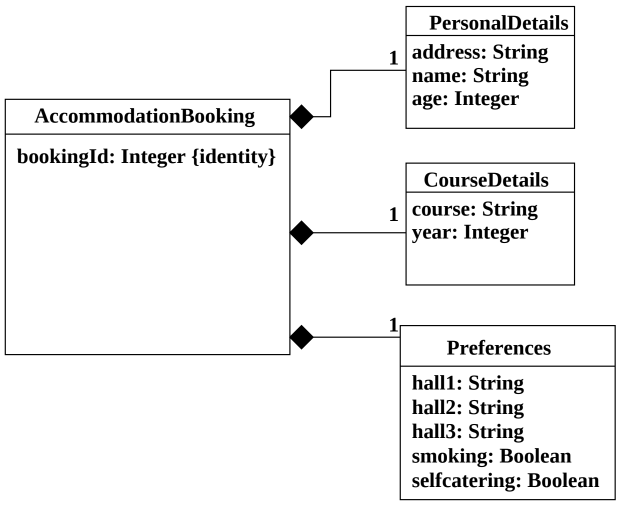

Fig. 28. Disaggregation for web forms

This transformation is related to 'Extract Class' in [3].

2.9 Factor out sub-operations

Description An operation is factored into sub-operations.

Purpose An operation may involve complex or repeated sub-computations. These can be factored into private helper operations of the same class, invoked from the operation.

Diagram Figure 29 shows a generic example where a complex expression *exp* is factored out into a separate operation $m1$.

Conditions It should be checked that the helper operations do not already exist in the class or in other classes before they are created. The helper operations should be query operations.

This transformation, combined with 'introduce superclass', gives the template method pattern in the case that methods in two separate classes have the same remainder after their helper method code is factored out.

A similar transformation introduces derived features:

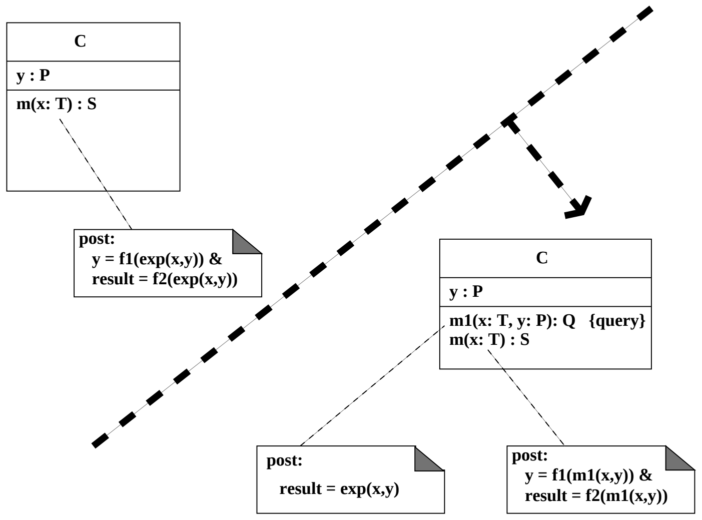

Fig. 29. Factoring an operation

2.10 Introduce derived features

Description An expression e built from local features of a class, which reoccurs several times in a specification, is replaced by a new derived feature f of the class, plus the constraint $f = e$.

Purpose Complex repeated expressions lead to inefficient implementations. A derived feature representing the expression need only be recomputed when one of its defining features changes value.

2.11 Introduce superstate

Description If states s_1, ..., s_n of a statechart all have a common set of outgoing transitions, ie, for a non-empty set α_1, ..., α_m of events they have transitions t_{s_1,α_1}, ..., t_{s_n,α_1}, etc, such that, for a given j, the t_{s_i,α_j} all have the same guards, actions and target states, then introduce a new superstate s of the s_i, and replace the t_{s_i,α_j} by new transitions t_{s,α_j} from s to the common target of the t_{s_i,α_j}, and with the same guard and actions. Common invariants of the substates can be placed on the superstate.

Purpose This reduces the complexity of the diagram (the number of transitions is reduced by $(n-1) * m$) and may identify a conceptually significant state that was omitted from the original model.

Diagram Figure 30 shows this transformation.

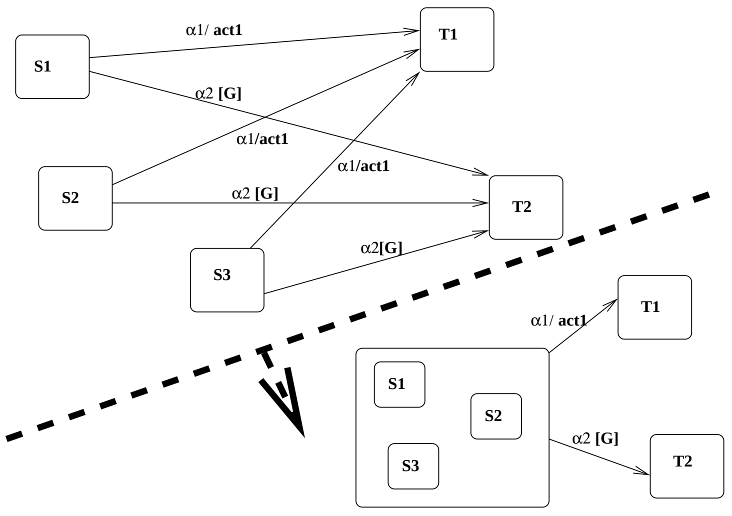

Fig. 30. Introduce superstate

Conditions The new state can only be introduced if it does not overlap with other states (except those it entirely contains or is contained by).

2.12 Introduce entry or exit actions of a state

Description If all transitions t_1, ..., t_n into a state s have the same final sequence act of actions, factor these out and define the entry action of s to be act instead.

Likewise, if all transitions with source s have a common initial action act', this can become the exit action of s.

Purpose This reduces the complexity and redundancy of the diagram. It may correspond to a more efficient or modular implementation.

Diagram Figure 31 shows an example of this transformation.

2.13 Raise supplier abstraction level

Description If class A is a client of class B, a subclass of C, make A a client of C instead if A only uses features and properties of C.

Purpose If A depends on an over-specific class, this reduces the independence of parts of the system and makes it less easy to modify.

Diagram Figure 32 shows this transformation.

Conditions The client must genuinely be independent of the specific features of its current supplier.

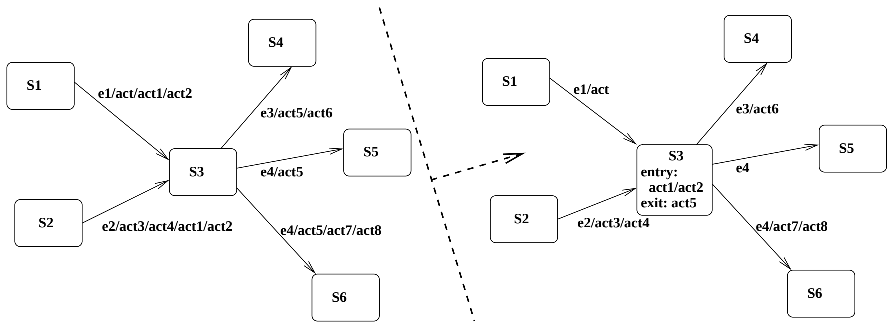

Fig. 31. Introducing entry and exit actions

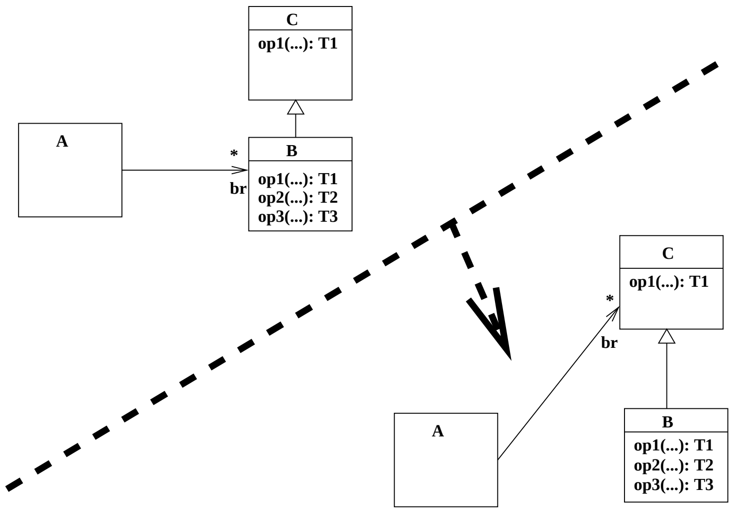

Fig. 32. Raise supplier abstraction

2.14 Express OCL constraints graphically

Description Replaces textual constraints by equivalent diagram elements.

Purpose This transformation makes the properties of the model more immediately visually apparent, more readable and comprehensible by a wider range of people and analysis tools, and more concise.

Examples are multiplicity constraints on association ends, cardinality constraints on classes, subset constraints between associations, etc. If a particular kind of constraint is needed frequently in a domain or system, then a new UML stereotype abbreviating the constraint can be introduced by means of a profile, and used in the diagram instead of the constraint formula. The 'identity' (primary key) stereotype of attributes is an example of this approach. UML allows new graphical icons or display styles to be used to indicate stereotyping, in addition (eg, the identity attributes could be written in bold font, instead of labelling them with '$\{identity\}$').

2.15 Make partial roles total

Description A 0..1 multiplicity role of a class A may be turned into a 1 multiplicity role by either moving the role to a superclass of its current target, or by moving the other end to a subclass of A on which the association is total.

Purpose Total associations are generally easier to implement and manage than partial associations.

Diagram Figure 33 shows the 'generalise target' version of this transformation. Figure 34 the 'specialise source' version.

Conditions In the first version we need the condition

$$br \rightarrow isEmpty() \;\Rightarrow\; not(cr \rightarrow isEmpty())$$

r is the union of br and cr.

2.16 Introduce module

Description This transformation groups together related classes into a single unit (eg, a UML package or subsystem).

Purpose This transformation improves the modularity and hence analysability of the model.

Diagram Figure 35 shows this transformation.

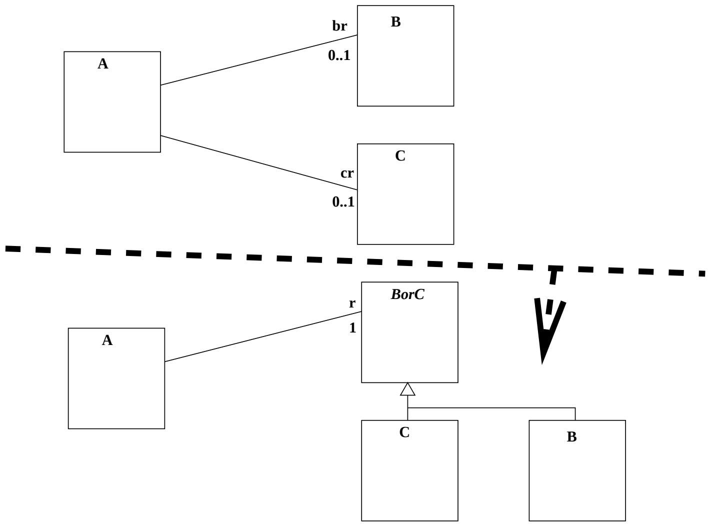

Fig. 33. Making partial roles total (a)

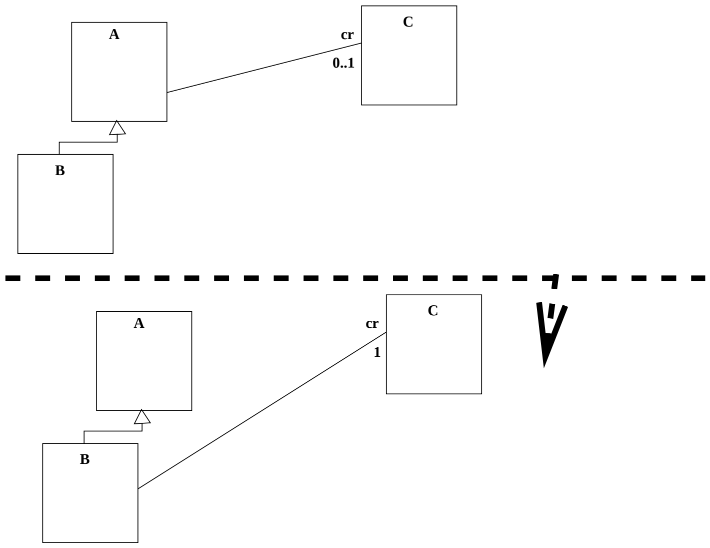

Fig. 34. Making partial roles total (b)

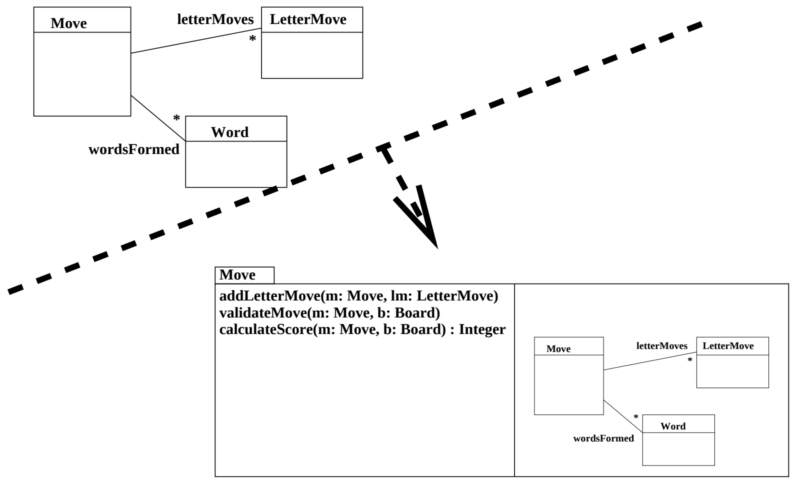
Move
letterMoves
LetterMove
*
*
Word
wordsFormed
Move
addLetterMove(m: Move, lm: LetterMove)
validateMove(m: Move, b: Board)
calculateScore(m: Move, b: Board) : Integer
Move
letterMoves
LetterMove
*
*
Word
wordsFormed

Fig. 35. Introducing modules

Conditions The classes must represent a coherent unit. Normally if a class is included in a module so are its subclasses, aggregate part classes, and those classes which are suppliers to it and not suppliers to any other class (ie, subordinate/auxiliary classes).

2.17 Simplify postconditions

Description A number of logical transformations can be made on postconditions of an operation to simplify its specification.

Two postconditions with the forms $A \Rightarrow B$, $A \Rightarrow C$ can be combined into a single postcondition $A \Rightarrow B \ \& \ C$.

Two postconditions with the forms $A \Rightarrow C$, $B \Rightarrow C$ can be combined into $A \ or \ B \Rightarrow C$.

If the left hand side P of a postcondition $P \Rightarrow Q$ is a test on the prestate, and is implied by a precondition of the operation, then the postcondition can be simplified to Q.

Purpose It is important to make postconditions as clear and simple as possible, to improve analysability and readability.

3 Design Patterns

Many design patterns can be considered as model transformations, restructuring a model of a system without application of the pattern into an improved model in which the pattern is applied. These transformations can be either quality improvements or refinements (or both) in their effect.

3.1 Introduce Abstract Factory pattern

Description This pattern allows a decrease in the level of coupling between classes in a system by enabling a client class to create objects of a general kind without needing to know what particular implementation subtype they belong to.

Diagram The before and after structures of a system to which this pattern has been applied are given in Figure 36. The notation of [4] has been used: a dashed arrow indicates a creation dependency, whilst a solid arrow indicates clientship.

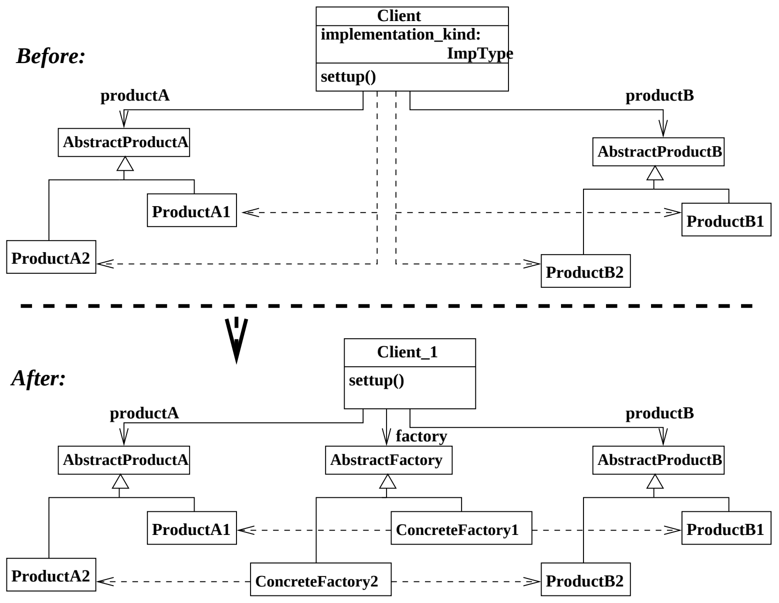

Fig. 36. Application of Abstract Factory Pattern

In the initial description of a system, we could have the following form of postcondition for the *settup* operation:

```
if $implementation\_kind = < type1 >$
then
  $(productA.oclIsTypeOf(ProductA1)$ &
   $productA.oclIsNew()$ &
   $productB.oclIsTypeOf(ProductB1)$ &
   $productB.oclIsNew())$
else
  $(productA.oclIsTypeOf(ProductA2)$ &
   $productA.oclIsNew()$ &
```

$productB.oclIsTypeOf(ProductB2)$ &
$productB.oclIsNew())$

The disadvantage of this approach is the necessity for a case statement and knowledge in *Client* of the names of the implementation classes *ProductA*1, *ProductA*2, etc.

In the revised version, we factor out the implementation dependence into the factory objects:

```
post:
```
$productA = factory.CreateProductA()$ &
$productB = factory.CreateProductB()$

An initialisation action to set *implementation_kind* in *Client* will become an action creating *factory* in *Client_*1. The concrete subtypes of *Factory* are *ConcreteFactory*1, whose *CreateProductA* and *CreateProductB* operations return new instances of *ProductA*1 and *ProductB*1 respectively. Similarly for *ConcreteFactory*2. The *implementation_kind* attribute is therefore replaced by polymorphic behaviour depending upon which subclass *ConcreteFactory*1 or *ConcreteFactory*2 of *AbstractFactory* the *factory* object belongs to.

Conditions An important correctness property which must be true for any pattern which introduces an intermediate class such as *Factory* to implement attributes of a client class, is that objects of this intermediate class should not be shared between distinct clients.

For example, if another object had access to the *factory* of a *Client* object *obj* then it could delete or change the class of *factory* during the execution of $obj.settup()$, so invalidating the above restructuring.

3.2 Introduce State pattern

Description An operation may consist of many cases, one for each element of an enumerated type. This explicit conditional choice can be made implicit by using operation polymorphism and separating the behaviour for each case into a separate subclass.

Purpose The pattern reduces the complexity of operations in the original class, and increases cohesion by localising aspects relating to a particular state into a separate class.

Diagram Figure 37 shows a general situation.

Condition Membership of the subclass for a state (eg, *A*1 for *state*1) in the refined model should coincide with the state attribute having that state as its value ($att = state1$) in the original model.

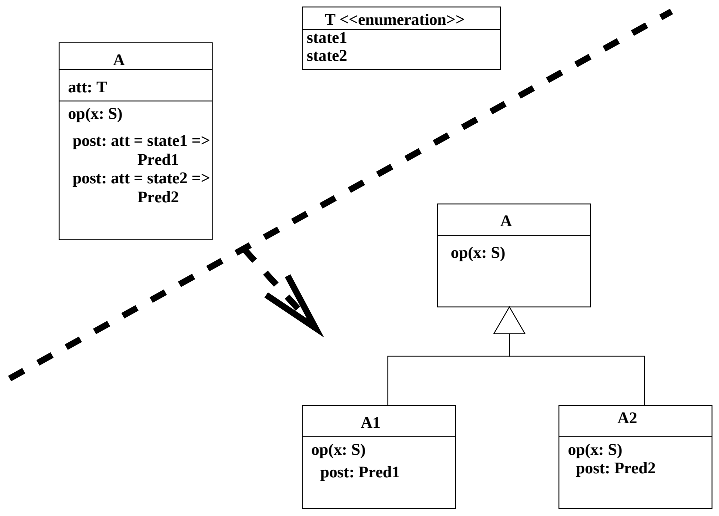

Fig. 37. Introducing the State pattern

3.3 Introducing the Facade pattern

Description The facade pattern aims at simplifying the dependencies between classes by bundling up a group of classes into a new subsystem: these classes are typically used as common suppliers by several client subsystems.

The direct dependencies between clients and suppliers are replaced by dependencies of the clients on a new facade class, which acts as an interface for the new subsystem. This facade then invokes operations of the original suppliers to implement the services required by clients.

Purpose This transformation increases the modularity of the system, and reduces the potentially $C * S$ dependencies between the clients and suppliers to $C + S$, where C is the number of clients and S the number of their suppliers.

Diagram Figure 38 shows a typical example.

Conditions The sets of clients and suppliers should be disjoint. Each new operation of the facade interface must reproduce exactly the functionality expected by the client that uses the operation to replace its original call(s) on the suppliers.

3.4 Introduce Singleton pattern

Description The Singleton pattern gives a standard design for a class which must have only one object instantiation.

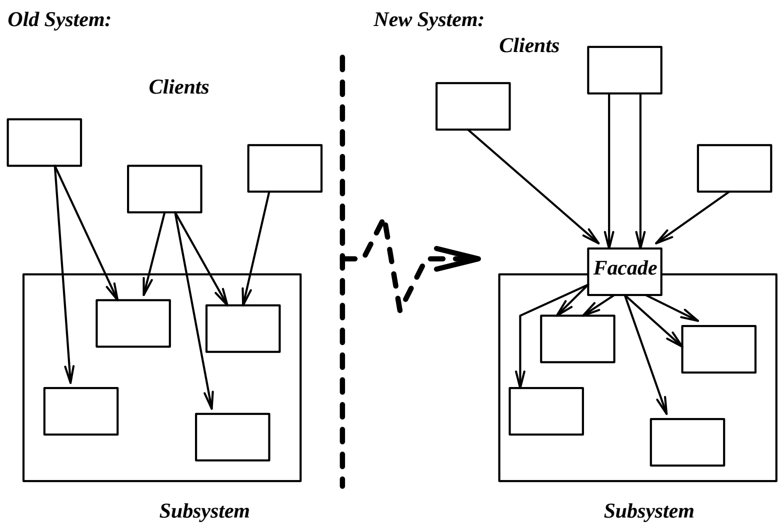

Fig. 38. Facade Pattern Design Step

Purpose This pattern implements a constraint

$$C.allInstances() \rightarrow size() \ \leq \ 1$$

for a class C.

Diagram Figure 39 shows the structure of a typical Singleton class after application of this pattern. The constraint is ensured by the definition of the constructor as private, and the $getInstance()$ operation.

3.5 Replacing synchronous invocation by asynchronous

Description This pattern replaces a synchronous call of a client on a supplier by an asynchronous call, using an intermediate active object.

Purpose This transformation aims to improve efficiency by reducing the time that the client is blocked waiting for the supplier to complete its action.

Diagram This is a transformation both on the class diagram of the system (Figure 40) and on the state machine (Figure 41).

Condition The required action $m(val)$ should not return any result required by the client.

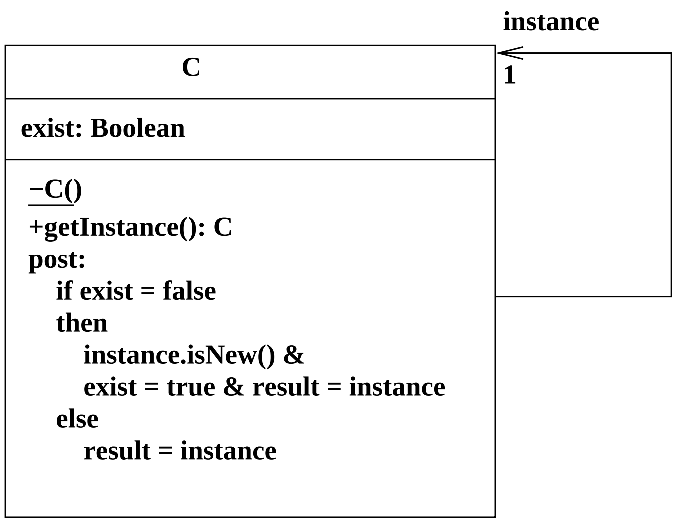

Fig. 39. Introduction of Singleton class

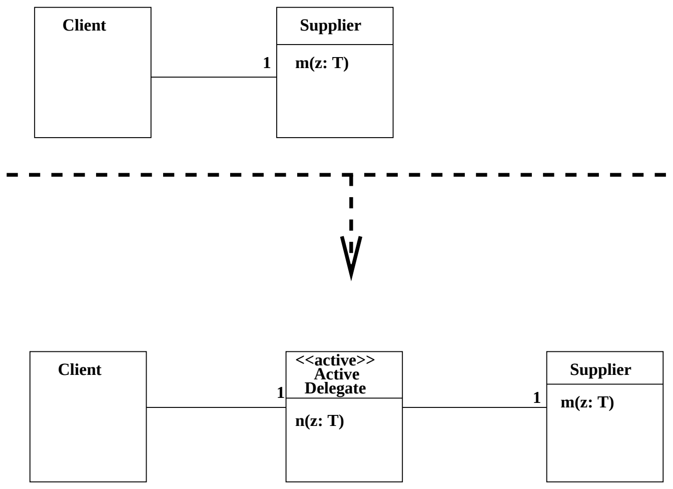

Fig. 40. Replacing synchronous by asynchronous: Class diagram

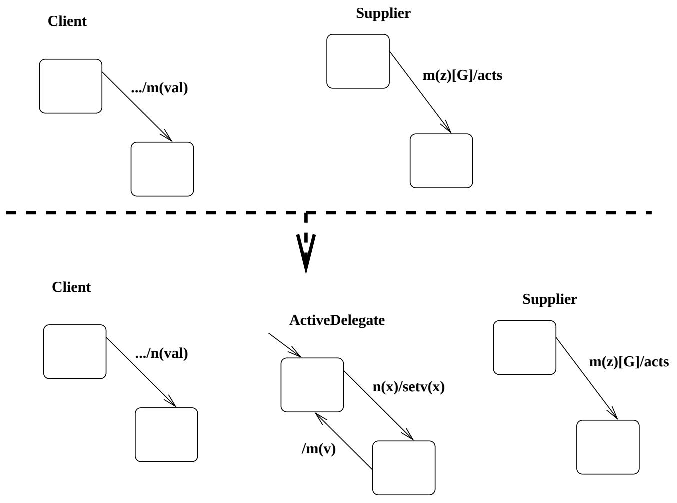

Fig. 41. Replacing synchronous by asynchronous: State machine

References

1. D. Bämer, et al, *Role Object*, Pattern Languages of Program Design, Addison-Wesley, 2000.
2. S. Cook and J. Daniels, *Designing Object Systems*, Prentice Hall, 1994.
3. M. Fowler, *Refactoring: Improving the design of existing code*, Addison-Wesley, 2000.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, 1994.
5. M. Grand, *Patterns in Java*, Wiley, 1998.
6. OMG, *UML 2.0 Superstructure*, 2005.