Model-based specification of design patterns

Model-based specification of design patterns

SOOP-working group[1][2]
Department of Computing Science
Eindhoven University of Technology

Motivation

A considerable interest in design patterns has sprouted in the last couple of years. This interest has been stimulated by the appearance of some excellent books on the subject. A notable example is the book by Gamma et al. [1]. At the time the first ideas of patterns in software engineering were developed, a couple of essential ingredients were available. First, the mechanisms of behavioral and structural abstraction as offered by object-oriented methods (i.e. encapsulation and inheritance). Second, the availability of a notation to communicate the patterns (i.e. OMT [2] as proper predecessor of UML [3]), and last but not least, relevant design-experience with object-oriented systems. The latter is important because patterns are supposed to capture experience on how to solve certain often-occurring problems in system design. Many of the patterns described by Gamma are intended for improving a design by changing class dependencies (possibly without affecting resulting object dependencies) such that groups of classes are de-coupled from others. For example, the observer pattern can be used to make problem specific classes ("concrete subject") independent of the user interface ("concrete observer") such that the changes in the user interface will not affect the implementation of a problem solution. It is interesting to observe that such patterns lean very heavily on inheritance and the related extended object substitutability.

Although the book of Gamma is impressive in its clarity, it has the drawback that it describes the intended application of patterns informally and intuitively. The descriptions of how a pattern works are typically operational. Although they can be understood well by the human reader, such an informal approach hampers specification and implementation of a level of tool-support for design patterns that goes beyond (trivial) diagram manipulation. In addition, it is very difficult to establish correctness of a design or implementation as a whole and the correct application of a pattern in particular. Therefore, the SOOP working group in Eindhoven has set itself the goal to develop a specification formalism that helps to formally specify patterns. The result of the attempt should be the construction of tools supporting pattern application and ultimately the establishment of a formal pattern language.

Requirements for a specification formalism

When one tries to develop a specification formalism that is to be used in practice, one must consider the generally accepted design approach and try to adapt the formalism to it, rather than adapt the design approach to the formalism. Following the principle of direct mapping (see Meyer [4]), design of an object-oriented program starts from a model of the problem domain. This model is successively refined into an implementation adding user-interface classes, adding auxiliary container classes that group objects in a way that facilitates efficient inter-object navigation, and applying design patterns to improve the modularity of the class structure. In our opinion, a specification formalism must tie in with this approach, i.e. it must parallel the decomposition of an object-oriented model and its refinement from analysis to implementation. In fact, decomposition of the implementation and specification should be isomorphic, i.e., yield a one to one correspondence between certain specification expressions and classes in the implementation. A consequence is that a specification of a class's behavior must only involve specifications of classes on which it directly depends. Specification fragments can then be taken from a context and can be reused elsewhere, just as implementation fragments can. Design patterns can be seen as such specification fragments.

Model based specification

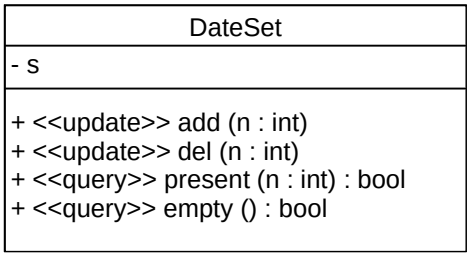
We follow the approach of *Design by Contract* [4,5], particularly advocated by Meyer. A contract is a specification of benefits and obligations in interactions between classes. In each interaction one class is the client, the other the supplier. A contract can be specified as pre- and post-conditions on operations

---

[1] Structured Object-Oriented Programming; email: `soop_l@win.tue.nl`;
[2] members: Lex Bijlsma, Rik van Geldrop, Louis van Gool, Kees Hemerik, Kees Huizing, Ruurd Kuiper, Onno van Roosmalen, Jaap van der Woude, Gerard Zwaan.

offered on the interface of a class. A pre-condition specifies the conditions that must be realized before the operation is invoked by an instance of the client class and the post-condition specifies the obligations of the invoked instance of the class, the supplier.
There are several methods to specify contracts. We think that the most intuitive and concise way to express them is by way of a *model*. A model is a mathematical way to describe the state of an object. It must be detailed enough to express the result of method invocations. A model is intuitively appealing because it is very close to a description in terms of implementation variables. Although it may give a bias towards the actual implementation of the specified behavior, it does not force a particular implementation upon the programmer. In actual fact, rules can be given that relate a model and the method specifications to a specification of behavior in terms of implementation variables.

Example



Consider a class DateSet depicted in the above UML diagram. The purpose of objects of this class is to keep a number of dates of the month, e.g. to be used for certain appointments. We adopt the following model for this class.

model : $s \in \wp(Z)$

The restriction on the values of dates in s is introduced using a, so called, *model invariant*.

inv I(s) : $(\forall i : i \in s : 1 \le i \le 31)$

Naturally, the possible use and implementation of this class entirely depends on the offered operations and their specified behavior, and not on this model.

{1≤n≤31} DateSet::add (n : int) { modify({s}) ∧ s = s. ∪ {n} }
{1≤n≤31} DateSet::del (n : int) { modify({s}) ∧ s = s. \ {n} }
{ true } DateSet::present(n : int) : bool {return (n∈s)}
{ true } DateSet::empty() : bool { return (s = ∅)}

The modify clause is a shorthand for the restriction that all other variables of the (current) model except the ones specified in the set, must be left unchanged by the operation. Absence of this clause implies no restrictions. However, queries may only return values and the restriction modify(∅) is always implicitly assumed. The dotted variable, e.g. s. , indicates the old value of s, i.e. the value in the precondition. Consider now an implementation of DateSet. We assume that a boolean array is used to keep track of which days belong to the set s . Hence a[i-1] has the value true if i∈s . This relation can be expressed in a, so called, *implementation invariant.*

inv Im(a,s) : (∀i : 1≤i≤31 : a[i-1] = i∈s)

The pre- and post-conditions applicable to an operation m() implemented using this representation, i.e. in terms of the implementation variables must satisfy the following refinement rule for any of its operations m() with pre- and post-conditions Pre_m and $Post_m$:

$\{ (\exists s::I(s) \wedge Im(a,s) \wedge Pre_m(s)) \}$ DateSet::m() $\{ (\exists s::I(s) \wedge Im(a,s) \wedge Post_m(s)) \}$ (1)

From this rule we can easily derive the following pre- and post-conditions for, e.g., the operation DateSet::add in terms of the implementation variables a[i].

{1≤n≤31} DateSet::add(n : int) { modify(a[n-1]) ∧ a[n-1] }

This suggests the simple implementation of setting a[n-1] equal to true.
End of example

An identical relationship to the refinement rule (1) exists between specifications of methods in subclasses of classes. In that case the implementation invariant in (1) has to be replaced by the *subtype invariant* which relates the variables of the sub-class to those of the super-class. Meyer [4] provides some rules as to how pre- and post-conditions and invariants may change under inheritance and when invariants should hold. In general it is too restrictive to require that the invariant holds during the complete execution of a method. The usual scheme is that every method has to restore the invariant at the end of the method and that it may assume that the invariant holds at the beginning. In the presence of **re-entrancy**, however, this scheme is too weak. In the next section we will deal with this problem.

Proof obligations and re-entrancy

For simplicity, in this section we assume that the model and the implementation are identical, so we leave out the implementation invariant and only mention the model invariant I. Furthermore, we assume that all constructors establish the invariant. Consider the following general form of a method with its specification and implementation:

$$\{ Pre_m \}\ m()\ \{ Post_m \}\ :\ \{Q_1\}\ S1;\ \{ R_1\}\ o1\text{->}m1();\ \{ Q_2 \}\ S2;\ \{ R_2\}\ o2\text{->}m2();\ldots\ Sn\ \{R_n\}$$

The proof obligations for methods are, to begin with:

1. $(Pre_m \wedge I_{this}) \rightarrow Q_1$ and $R_n \rightarrow (Post_m \wedge I_{this})$
2. $\{ R_i \}$ oi->mi() $\{ Q_{i+1} \}$
3. $\{Q_i\}$ Si $\{ R_i \}$

Since various forms of mi() may be invoked depending on the dynamic type of the object that o1 refers to, the following rules apply. Assume that the static type of o1 is C and D is a sub-type (sub-class) of C we write D<:C. Since, by hypothesis, that the subtype invariant between D and C is trivial, we find from the refinement rule (1):

$$Pre_{C::mi} \rightarrow Pre_{D::mi} \quad \text{and} \quad (Post_{D::mi} \wedge I_D) \rightarrow (Post_{C::mi} \wedge I_C)$$

This is intuitive because D::mi() can be actually invoked wherever mi() appears. This states that the post-condition and invariants must be strengthened in such subclasses and preconditions must be weakened.

As follows from proof obligation 1, the invariant may be assumed when a method starts and need not be established by the caller. The reason is that the client object will be found to satisfy its invariant automatically if every operation on the object satisfies the invariant. This reasoning, however, is not necessarily correct. The problem here is **re-entrancy**. Assume in the above program sketch that the object o invokes operations of other objects and that ultimately control re-enters the object, this, executing m(). At that point we have no guarantee that the invariant holds unless we demand that:

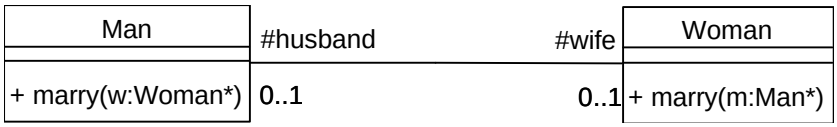
$$R_i \rightarrow I_{this}$$

Thus it seems necessary to demand some stronger property of the invariant namely that it holds whenever control leaves an object.

Vulnerability of invariants

Another problem is possible sensitivity of invariants with respect to modifications in other objects. It is possible that the invariant I_{this} is formulated in terms of properties of other object this depends on, e.g. the object referred to by o in the above program. This is no problem in the above method m() because we explicitly require restoration of the invariant by execution of this method. However, if methods of the object referred to by o are invoked directly, i.e. without going through this, it can no longer be guaranteed that the invariant I_{this} holds when entering m(). We say that I_{this} is **vulnerable** to changes in the objects o can refer to.

We consider the *forward-backward problem* as described by Meyer [4]. It concerns a hypothetical variant of asymmetric marriage relations.



We assume asymmetric model for men and women.

Man: **model** wife : Woman*
inv M : wife $\neq\emptyset \rightarrow$ wife->husband = this
Woman: **model** husband : Man*

Thus it is always required that if a man has a wife, his wife has him as a husband. The converse is not required. Although the following method implementation seems to preserve the invariant

{ w$\neq\emptyset$ } w->marry(this); wife = w; {modify({wife}) $\wedge$ wife = w $\wedge$ wife->husband=this},

even in the strong sense (at invocation of w->marry(this)) it is not preserved in general. For example, the execution of the following program fragment

```
Man m1,m2;
Woman w;
m1.marry(&w);
w.marry(&m2);  { m1.wife->husband = &m2 }  ,
```

that may appear anywhere, will break the invariant of the object m1. Note that introduction of a tighter specification for the marry operation in Woman would not solve the problem because new methods can be introduced in derived classes that could destroy the invariant of men in a similar fashion. We have to solve this problem because we do not want correctness of a class-implementation to depend on code that can appear just anywhere and could even be added in later program extensions. The solution we propose is to introduce a so-called *post-invariant* that restricts changes in women, i.e. must be added as a conjunction to all methods' post-conditions.

Woman: **model** husband : Man*
post P: husband**.** $\neq\emptyset$ $\wedge$ husband**.**->wife = this $\rightarrow$ husband = husband**.**

Furthermore we have to restrict calls to Woman::marry to those women who are not someone's wife by strengthening the precondition:

```
{ m≠∅ ∧ (husband=∅ ∨ husband->wife≠this)}
Woman::marry(Man* m)
{ modify( {husband} ) ∧ husband = m }
```

With this specification of Woman it can now be shown that the invariant for men can not be broken by methods of Woman. Note the appearance of the dot-notation in the post-invariant. Note that we have indeed arrived at a situation where we only need to consider the local view from class Man to satisfy our proof obligations. This is what we want to achieve in general. It would mean that proofs can be provided class by class in an incremental fashion. The feasibility of this goal has not yet been established.

Note that in the specification of the class Man other possible side-effects on women, e.g. a change of a possible variable name is not specified. Since the modify clause is defined as only referring to local variables, it does not forbid these side-effects. We could extend the semantics of the modify clause along these lines, but then again we would end up with a non-local specification. Therefore, we propose to formulate restriction locally by explicitly specifying which methods may be called. For example the post-condition of Man::marry method we extend to:

modify({wife}) $\wedge$ calls({w->marry}) $\wedge$ wife = w $\wedge$ wife->husband=this

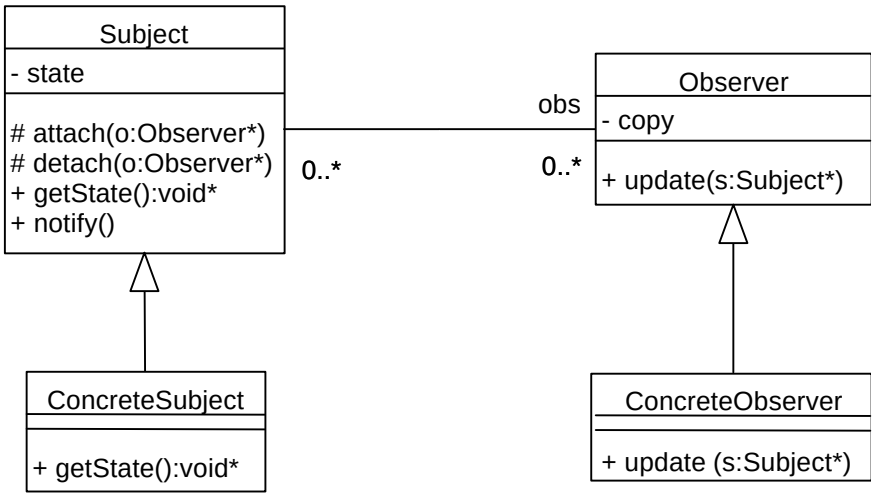
The calls-clause implies the possible (external) modifications as expressed in the modify-clauses of the callable methods. Note that calls({w->marry}) does not imply that w->marry will definitely be invoked.

Tracing the calls-clauses it is possible to establish whether re-entrancy is at all possible or not. Consider our husbands and wives. It can be seen that it is not possible to enter any instance of the class Man before the Man::marry() method is finished. In this case we can then relax the condition that the invariant is maintained in the strong sense and even the following implementation can be proved correct.

{ w≠∅ ∧ (husband=∅ ∨ husband->wife≠this)}
wife = w; w->marry(this);
{modify({wife}) ∧ calls({w->marry}) ∧ wife = w ∧ wife->husband=this }

The Observer pattern

We will briefly show how the sketched approach can be applied to the Observer pattern as described in [1]. In addition to the classes Subject and Observer, the pattern describes concrete versions of these classes that can be derived from them. In this process some methods must be defined



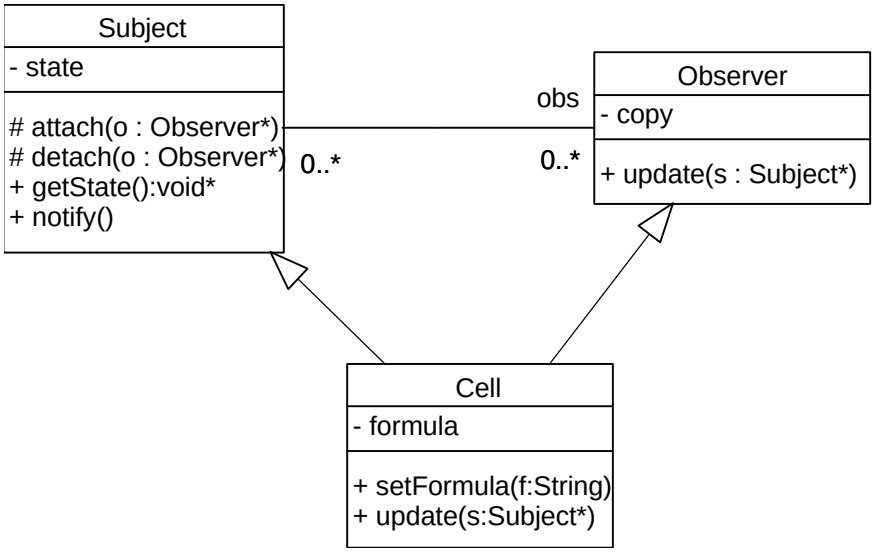
Subject **model** state : void*
obs ∈ ℘(Observer*)
post (∀o : o ∈obs : o->copy(this) =state)

{ true } Subject::attach(o:Observer*) { modif({obs}) ∧ calls({o->update}) ∧ obs = obs. ∪ {o} }
{ true } Subject::detach(o:Observer*) { modif({obs}) ∧ obs = obs. \ {o} }
{ true } Subject::getState() : void* { modif(∅) ∧ calls(∅) ∧ return(state) }
{ true } Subject::notify() { modif(∅) ∧ calls({ o->update | o∈obs }) }

Observer **model** copy ∈ Subject → void*

{ this ∈ s->obs }
Observer::update(s:Subject*)
{ modif({copy(*s)}) ∧ calls({s->getState}) ∧ copy(*s)=s->state }

A spreadsheet cell fulfills the roles of a subject and an observer simultaneously. This is readily implemented by multiple inheritance. The model for class Cell contains variables corresponding to the models of both Subject and Observer.



Cell: **model** state : Number
obs : $\wp$ (Observer)
copy : Cell $\rightarrow$ Number
formula : (Cell $\rightarrow$Number) $\rightarrow$ Number
inv state = formula(copy) $\wedge$ ($\forall$p: this$\in$ FREEVAR(p.formula) $\rightarrow$p$\in$ obs)
post ($\forall$o: o$\in$ obs: o->copy(this) = state)

Just consider the specification for Cell::update :

{ this $\in$ s->obs }
Cell::update(s : Subject*)
{ modif({copy(*s)}) $\wedge$ calls({s->getState, notify }) $\wedge$ copy(*s)=s->state }

Note that according to the modify-clause in Subject, Subject::state may not be changed by Subject::notify. In cell, however, Cell::state must be kept equal to cell::formula(copy) which is modified by Cell::update. A cyclic dependency of a cell to itself that would lead to such self-modifications is excluded by this specification. Hence, an implementation of Cell would inlcude an invariant that forbids cyclic dependencies.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *"Design Patterns, Elements of ReusableObject-Oriented Software"*, Addison-Wesley, 1995.

[2] J.Rumbaugh, M. Blaha, W. Premerlani and F. Eddy, *"Object Oriented Modeling and Design"*, Prentice Hall, 1991.

[3] Grady Booch, James Rumbaugh and Ivar Jacobson; *The Unified Modeling Language User Guide*, Reading Massachusetts: Addison-Wesley, 1995.

[4] Bertrand Meyer, *"Object Oriented Software Construction"*, second edition, Prentice Hall, 1997.

[5] C. Szyperski, *"Component Software, Beyond Object-Oriented Programming"*, Addison Wesley, 1997.