A Recurring Fulfilments Analysis Pattern

Lubor Sesera
Softec, Ltd.
Kutuzovova 23, 831 03 Bratislava, Slovakia
e-mail: lubor@softec.sk

Abstract. This paper addresses the problem of recurring fulfilments in a complex system of obligations. It recommends decoupling an obligation from a party and generating prescriptions from the obligation at regular time points. The prescription generated in this way holds all necessary data and fulfilments are assigned to it. In this way obligations and their fulfilments can be checked easily at any time. The analysis pattern arose from the insurance business but it can be applied to other domains such as loans, installments, and state social support. It concerns not only money income but money outstanding as well.

Motivation

An insurance company makes an insurance contract with a client and requires a premium. The insurance contract is usually valid for some period of time and a client does not have to pay the entire premium at once. The insurance company wants to know at any time the actual premium sum and how much of it has already been paid. This must be found out quickly as it can be requested, e.g., by a client's phone call or during his / her personal visit.

Context

A party has financial obligations with respect to another party. These obligations are due to an order, a contract or an agreement between these parties or due to another legal document specified by a low (e.g. an approved application for a state benefit). In this paper we use the abstract concept of 'obligation' for all these kinds of documents. Fulfilments are regularly reoccurring; however, the amount of money to pay can differ slightly depending on various conditions. Each party can have many obligations undertaken with many counterparties.

The context can be applied on any of the two parties: the party that fulfils obligations or the party that receives those fulfilments respectively.

Problem

How to find out conveniently and quickly actual obligations and their fulfilments with regard to any specified time point in such a complex system?

Forces

- The system is rather complex: a party can have many obligations simultaneously and, furthermore, these obligations depend on various conditions that can be changeable in time.
- Developers want simple models. Models having many classes with complex collaboration are hard to understand and maintain.
- Obligations and their payments must be discovered quickly and accurately during interactive work of a user.

- The model must be flexible enough to accommodate a variety of obligations, parties and fulfilments.

Solution

Decouple prescriptions from their obligations. Include an operation that generates prescriptions at regular time intervals. Each execution of this operation generates prescriptions that are valid with respect to that time interval. Record all necessary data within the prescription so that they do not have to be calculated in the future. Assign fulfilements to such prescriptions.

Requirements[1]

1. *Define an obligation type.*

   A domain expert defines a type of an obligation including constraints and amount of money. Constraints include:

   - Constraints on party role types.
   - Conditions applied on verification of an obligation.
   - Conditions determining a sum of money when several amounts of money are associated with the obligation type.
   - Conditions applied on a prescription generation.

2. *Manage an obligation.*

   A user, e.g., a registrar creates an obligation or manages its update. Managing an obligation, in addition to the obligation itself, includes the management of parties, their roles, the method of payment etc.

3. *Generate a duty.*

   An actor, e.g., an accountant or a system clock, runs a function automatically generating prescriptions based on *valid from* dates and frequency values in obligations.

4. *Handle a payment.*

   A user, e.g., an accountant, handles a payment of a prescription (s). The system helps in identifying a party and prescriptions paid.

[1] In the form of Use Cases.

Structure

- Subpattern (the core idea)

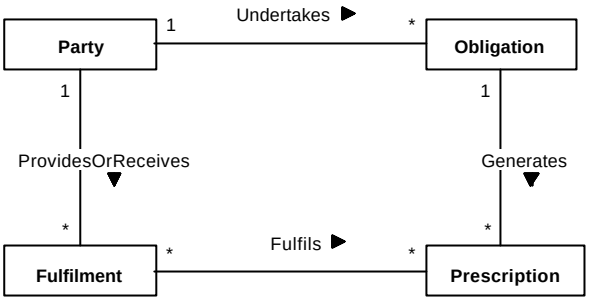

Figure 1 The Recurring fulfilments core subpattern (simplified)

- Superpattern

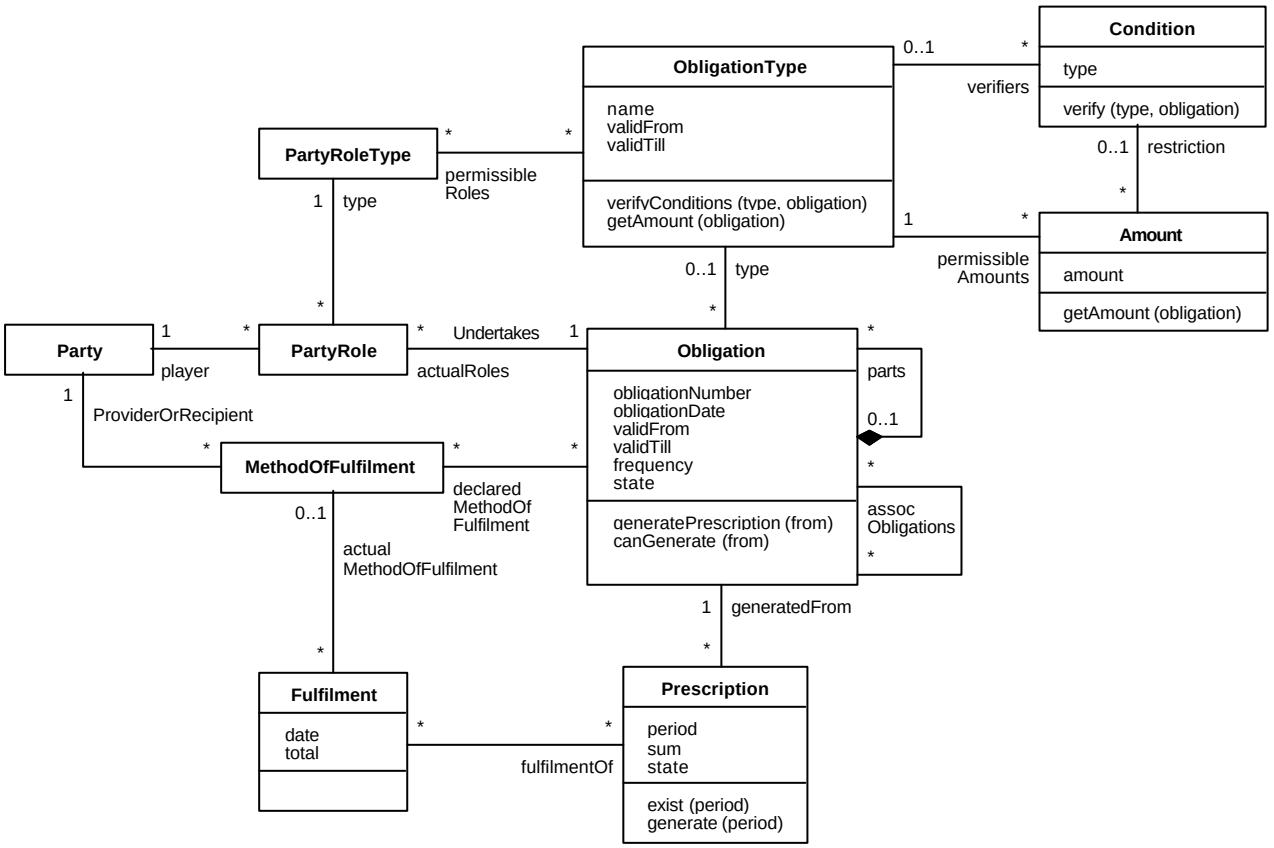

Figure 2 The complete Recurring fulfilment pattern

Participants

The pattern contains classes on two levels of abstraction [Fowler 1997]: the knowledge level (Obligation type, Party role type, Condition, Amount) and the operational level (all other classes).

In the knowledge level an obligation type is associated with conditions that are to apply to obligations of this type in various stages of their 'life-cycle'. Conditions are of several kinds: verifying an obligation, selecting an obligation amount, and recurring verifications when generating a prescription. Furthermore, the obligation type is assigned permissible amounts of money to be paid. An amount can be either an absolute value or an argument of a parametric function (e.g. a percentage tariff in the insurance business etc.) If several amounts are assigned to an obligation type, they are distinguished by conditions that are evaluated when generating a prescription. Finally, the obligation type specifies permissible party roles.

The operational level consists of three sets of classes as prescribed by the use case requirements:

1. Obligation and associated data

   As it was outlined in the Context section, obligation is an abstraction of various kinds of (legal) documents, e.g. a contract, an order, an application, etc.[2] It is decoupled from a party because a party can have several obligations simultaneously (e.g. a client can have several insurance contracts with an insurance company). Obligation can have components, e.g. assured risks are parts of the insurance contract. We show this possibility by recursive aggregation in Figure 2.

   The obligation is associated with actual parties via their roles. In general the obligation can be associated with multiple party roles. When considering a state social support[3], for instance, all family members along with their roles must be mentioned on a family allowance application.

   The main counter party, i.e. the provider or the recipient declares a method of fulfilment for the obligation in order for fulfilment can be identified, or the other party knows the method of fulfilment respectively. The method of fulfilment is an abstract class. A bank account, a credit card, a postal address etc. are examples of its concrete subclasses.

2. Prescription

   The prescription is a receivable or a liability of a party to another party associated with a certain time period. It is generated from the obligation. The prescription holds all necessary data that are risky or inefficient to recalculate, especially the sum of money.

3. Fulfilment

   The fulfilment holds actual fulfilments by a party. In the case of receivable it is assigned to the provider; in the case of liability, it is assigned to the recipient. As a party can utilize several methods of fulfilments, this assignment is indirect via the method of fulfilment object. Furthermore, the fulfilment is associated with the prescription or the prescriptions it concerns.

   In the case of receivables there is a specific algorithm behind it. It works in two steps: first it identifies the provider (using the database of declared methods of payment), and then it finds out the prescriptions. The creation of the corresponding links, i.e. to MethodOfFulfilment or to Prescription respectively is the result of each step.

[2] Obligation can be considered a special case of Fowler's Accountability [Fowler 1997].

[3] At least in Slovakia.

Collaborations

In Figure 3 an illustrative sequence diagram for one of the requirements, 'generate prescription' in particular, is outlined. For simplicity, it does not show extensions and exceptions. A message argument in quotes represents passing a direct value (contrary to passing an argument that must be calculated, e.g. an expression or a value of variable).

An actor starts the batch function handled by the Obligation class. The generate prescriptions function runs over the collection of obligations.[4] First it tries to find out whether a prescription should be generated for a given period (the obligation may have expired, the prescription has already been generated, etc.). Then, conditions are evaluated (e.g. validity of some attributes may have expired). Afterwards a sum of money is calculated. If there are several amounts the actual sum of money depends on other conditions. Finally the prescription is generated.

The messages from Figure 3 were used to define operations in the classes of Figure 2.

Consequences

The Recurring fulfilments pattern has some benefits as well as some liabilities:

1. An obligation is decoupled from a party. In many application domains it is clear and both domain experts and software developers are accustomed to it (there are, e.g., explicit contracts in the insurance business). In other domains this may be a bit confusing (e.g., there are no contracts in our country when companies pay their obligations to the National Labour Office[5]). Decoupling the obligation from the party is necessary when a party can have multiple obligations simultaneously. Otherwise these entities can be merged.
2. The obligation can include several parties each in its specific role. It seems there are only a few examples when this is required (e.g. the family allowance example mentioned earlier). Many times one party is enough, i.e. the party being on the opposite side of the party running a software system. In such cases the pattern can be simplified.
3. The pattern outlines a declarative representation of conditions and amount of money. It does not go into much detail concerning this issue, however. In a software system for the state social support we were developing [Phare 1999] these conditions were specified and implemented in much more detail. Unfortunately, that representation was too application specific, something that seems to be almost a rule nowadays.
4. The pattern assumes that fulfilments are assigned to prescriptions. In some systems this is not necessary: they need to balance the overall sum of fulfilments with the overall sum of prescriptions only. Many systems have to be more precise, however. Assignments of fulfilments to prescriptions can be a complex task: one fulfilment can carry out several prescriptions and a prescription can be associated with several fulfilments (this explains the many-to-many cardinality between Prescription and Fulfilment in Figure 2). Algorithms about how fulfilments are assigned to prescriptions are rather domain specific and / or company proprietary; because of that, only the possibility is sketched out here.
5. History is not represented in the model.[6] For instance, a party can change some attributes of its obligation later. Due to this change, prescriptions are calculated differently from that moment but this does not affect previous prescriptions. Previous attribute values should be kept so that

[4] To be more precise we should also introduce collection classes or class functions. For simplicity, this is omitted here.

[5] However, the Obligation class is needed, because a party can pay several obligations simultaneously, e.g., being both an employee and a business person as well.

[6] Although it is in the backgorund to some extent.

previous prescriptions could be checked any time in the future. Similarly, history should be kept on knowledge level classes (e.g., condition, amount, etc.) as well.

Known uses

We have used the Recurring fulfilments analysis pattern in our company when developing software systems for both a traditional insurance (such as asset insurance) [Koop 1999][7], [Sesera 2000b] and a social insurance (health insurance, unemployment insurance etc.) [Health 1996]. Application of the pattern for the property insurance is shown in Figure 4.

The pattern could be used in the loan and installment domains although we do not have explicit references.

All pattern applications mentioned above concern money income (receivables). However, the pattern can be used for money outstanding (liabilities) as well. For instance, we used it in the ISOP state social support information system [Phare 1999], [Sesera+ 2000a]. The application of the Recurring fulfilments pattern for the state social support is shown in Figure 5.

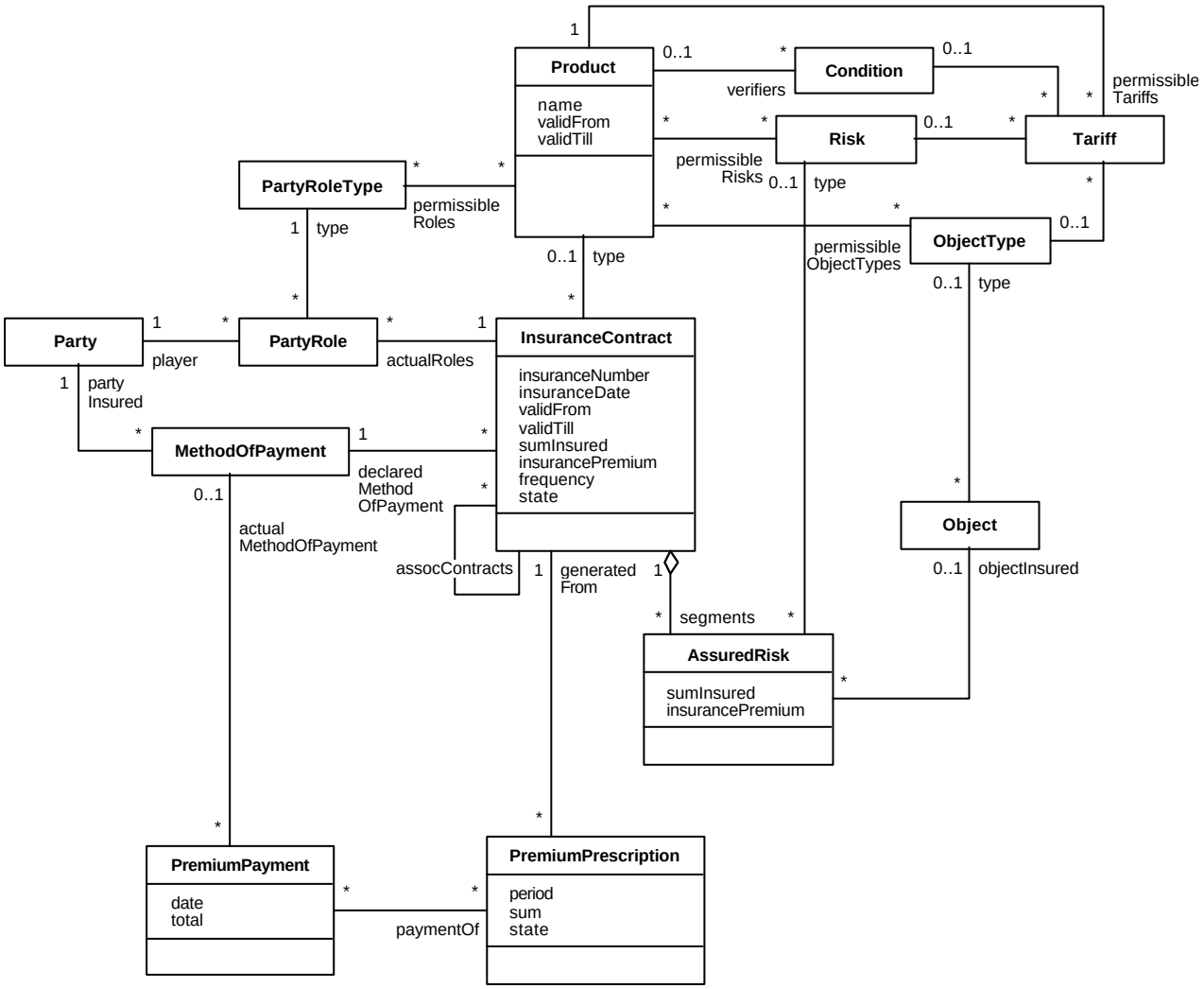

Figure 4 Application in the property insurance domain

[7] The Kooperativa insurance company is a subsidiary of Wiener Stadtische Versicherung.

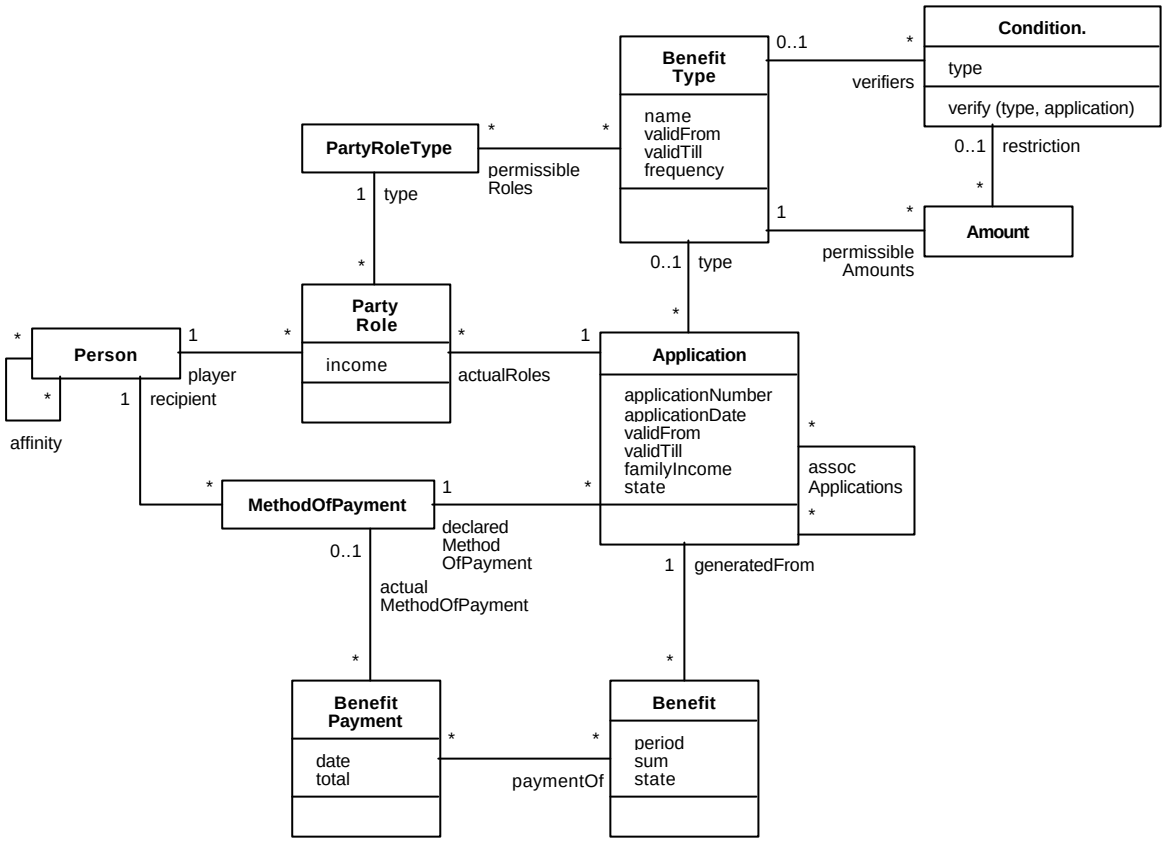

Figure 5 Application in the state social support domain

Related Patterns

The Recurring fulfilments pattern uses Fowler's Accountability pattern ([Fowler 1997] p. 25) as a subpattern, i.e., Obligation corresponds to Accountability and Obligation Type corresponds to Accountability Type. As we allow an n-ary relationship between an obligation and its parties, we changed the Accountability pattern in the style of the Contract Roles pattern (from [Hay 1996] p. 107).

Martin Fowler proposed another 'recurring' pattern, namely Recurring Events [Fowler 1996]. The purpose of his pattern is different than ours. That pattern focuses on the issue of many different events recurring on different times without addressing any (financial) obligations. To the contrary, our aim is prescriptions of financial obligations and not the time when their generation is scheduled. All kinds of prescriptions are usually generated together and regularly (e.g. each day or once a week). If there is a need, however, these patterns can be used together (combined).

The Recurring fulfilments pattern addresses the temporal issue as well. Both Obligation and Prescription are associated with a validity period of their values. As they are explicit, each of them is a specific case of the Temporal Association entity of the Temporal Association pattern [Carlson+ 1999].

The Recurring fulfilments pattern arose from the insurance business. Wolfgang Keller may have been the first who provided some valuable patterns for this domain [Keller 1998]. His patterns are of different kinds, however. First, they are dedicated to insurance business while our pattern is more general. Second, his patterns are usually more coarse-grained (see e.g. the Insurance

Value Chain) while our pattern is pretty fine-grained. The main difference is in the topic of interest. Keller addresses the knowledge level, the issue of a product definition (patterns: Product Tree, Object Event Indemnity, etc.) and policies (Policy as Product Instance) in particular, while our aim is the operational level, namely money income.

Acknowledgement

I would like to thank my shepherd Ed Fernandez for his comments, suggestions and all the help he provided during the revision of this paper. I am also grateful to members of the writers' workshop at PLoP, especially Kyle Brown, Joseph Yoder, Ali Arsanjani and Kent Beck, for their valuable suggestions for improvements of this pattern.[8]

References

[Booch+ 1998] Booch, G., I. Jacobson, and J. Rumbaugh. Unified Modeling Language User Guide. Addison-Wesley, 1998.

[Carlson+ 1999] Carlson, A., S. Estepp, and M. Fowler. Temporal Patterns. In: Pattern Languages of Program Design 4. Addison-Wesley, 1999.

[Hay 1996] Hay, D. Data Model Patterns: Conventions of Thought. New-York: Dorset House, 1996.

[Fowler 1996] Fowler, M. Recurring events, PLoP'96.

[Fowler 1997] Fowler, M. Analysis Patterns: Reusable Object Models, Reading, MA: Addison-Wesley, 1997.

[Gamma+ 1995] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995.

[Health 1996] Information System of the General Health Insurance Company. 1996 (in Slovak).

[Keller 1998] Keller, W. Some Patterns for Insurance Systems. PLoP'98. Also at: http://ourworld.compuserve.com/homepages/WofgangWKeller/

[Koop 1999] Software system for the Kooperativa insurance company. 1999 (in Slovak).

[Phare 1999] Information System of the State Social Support. Phare # 951801, 1999.

[Sesera+ 2000a] Sesera, L., A. Micovsky, J. Cerven, J. Architecture of software systems: Analysis Data Patterns. Textbook. Slovak Technical University, 2000 (in Slovak) (in print).

[Sesera 2000b] Sesera, L. Analysis Patterns. (Invited talk.) In: SOFSEM'2000. Lecture Notes in Computer Science series, Springer-Verlag, 2000 (in preparation).

[8] Their main suggestion was to break down this 'pattern' into a pattern language that I consider a challenge for my future work.

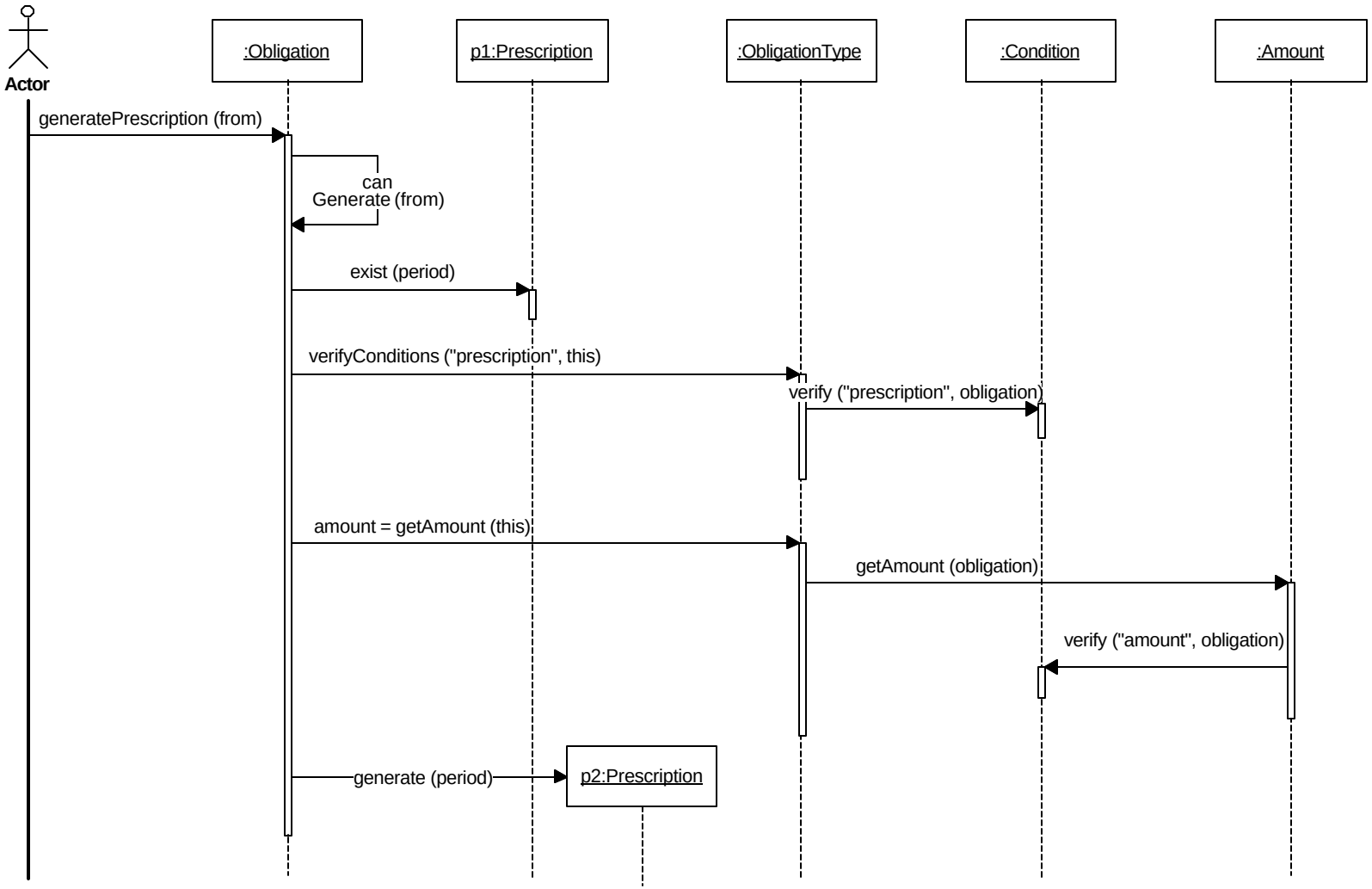

Figure 3 Sequence diagram for generation of prescriptions