PROPAGATOR — A Family of Patterns

Peter H. Feiler
Software Engineering Institute[1]
Carnegie-Mellon University
Pittsburgh, PA 15232

Walter F. Tichy
University of Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

PROPAGATOR is a family of patterns for consistently updating objects in a dependency network. The propagator patterns are found in such diverse applications as MAKE, WWW, spreadsheets, GUIs, distributed file systems, distributed databases, workflow systems, and multilevel caches.

There are four main patterns: STRICT PROPAGATOR, STRICT PROPAGATOR WITH FAILURE, LAZY PROPAGATOR, and ADAPTIVE PROPAGATOR. In the strict propagation patterns, updates flow from the point of original change forward through the network. Dependent objects are immediately updated, unless failure occurs. In the lazy propagation pattern, an updated object merely sets its timestamp, without notifying any other object. Upon request, a dependent object makes itself current after requesting predecessor objects to bring themselves up-to-date. The adaptive propagator separates propagation of out-of-date markers from the actual update. It supports a combination of strict, lazy, and periodic update schemes.

All patterns support smart propagation for avoiding redundant work as well as concurrent updates.

Examples of these patterns are formulated in the Internet language Java.

[1] The Software Engineering Institute is sponsored by the U.S. Department of Defense.

1. PROPAGATOR

1.1 Intent

Define a network of dependent objects so that when one object changes state, the change propagates to all direct and indirect dependants.

1.2 Also Known As

CASCADED UPDATE, RECURSIVE OBSERVER

1.3 Motivation

PROPAGATOR is a generalization of OBSERVER [Gamma95]. Any object that is an observer may itself be the subject of several observers, resulting in a dependency network. An observer may be dependent on more than one subject, so the network can be a tree or a directed and potentially cyclic graph. Any object may be updated, and the change cascades through multiple levels of dependencies until pure observer objects are reached, a cycle is detected, or a fixed point in an update cycle is obtained.

The difference between OBSERVER and PROPAGATOR is as follows. With OBSERVER, subject and observer objects are neatly separated. With PROPAGATOR, the subject of one level is merged with the observer of the previous level and a dependency network of arbitrary depth is possible. Furthermore, PROPAGATOR gives rise to a range of differing update strategies.

When a system is partitioned into a network of dependent objects, consistency with respect to change becomes an issue. An update of any object must propagate to all dependants. In some cases it is critical that objects are always consistent, i.e., a change must be propagated immediately. In other cases temporary inconsistency is tolerable, provided it is detectable and consistency can be re-established on demand.

A dependent object may simply contain a duplicate of the predecessor object, e.g., cached data, or a representation in a different form, e.g., a presentational view of data, or the result of a computation based on one or more predecessors, e.g., a spreadsheet cell or a translation of source code into object code.

Updates can be handled in a number of ways:

- Updates may have to cascade immediately, e.g., users want to see immediate spreadsheet recalculation after a cell has changed.
- The update of an individual object may fail and prevent further propagation of the change, e.g., recompilation or computation may stop in mid-propagation.
- The update may only propagate on demand. For example, in MAKE – a program for organizing processing steps when compiling and linking software [Feldman79] – a dependent object such as an executable is requested to be up-to-date. As a result of the request the dependency network is traversed backwards to bring all relevant objects up-to-date.

- Updates may be performed periodically. For example, in a simulation system the displayed data may be updated once a second, while the simulation state changes more frequently. In the World Wide Web, cached pages at a site are dependent on their originals but are updated only when touched and of a certain age.
- It may be desirable to operate in a combination of update modes.

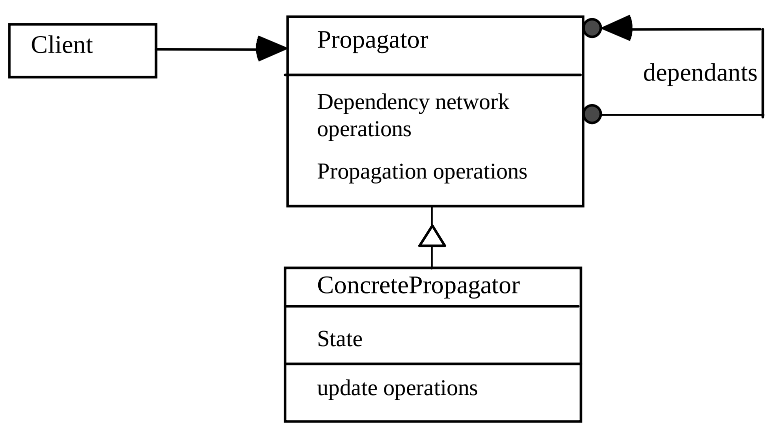
Other considerations to take into account are:

- At any given time, several different updates may propagate through the network concurrently.
- Based on the particular nature of the change to the original object, the state of the dependants may not be affected.
- The dependency structure itself may change during the life of the system.

1.4 Applicability

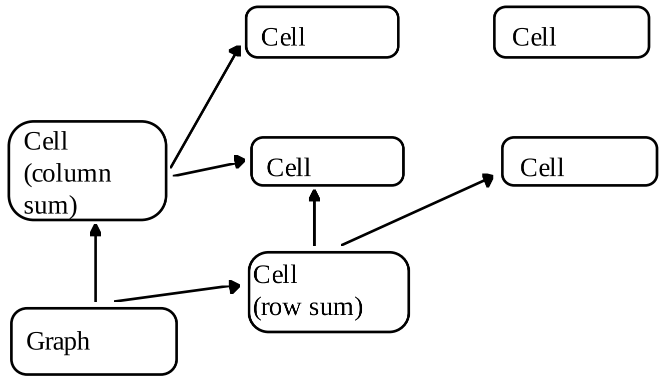
Use this pattern in the following situations:

- When there is a network of dependent objects that need to be kept consistent.
- When a change to one object requires changes to others, but the changed object cannot make any assumptions about the extent of the updates nor about the structure of the affected objects and therefore cannot perform the updates itself.
- When the dependency structure changes dynamically.



1.5 Structure

The key to the pattern is the class Propagator which represents any object that can be part of a dependency network. This class handles all dependency network related operations, while ConcretePropagator provides an update operation. All members of the PROPAGATOR family share this structure, but differ in the propagation mechanisms. The following diagram shows a typical dependency network from the spreadsheet domain. The ConcretePropagator classes are cells in a spreadsheet, some containing formulas, and a graph displaying the data.



1.6 Consequences

Arbitrary subclasses of Propagator can be combined in a single network. The dependency structure of the network can be varied dynamically by adding or deleting dependants.

1. *Abstract coupling between Propagators.* The abstract class Propagator focuses on maintaining the dependency graph and on cascading the change. It has no knowledge of the specifics of ConcretePropagator objects other than the existence of an update method. These objects may contain different data structures, belong to different layers of abstraction in a system, or be transformations of other objects. For example, the concrete objects may be source code, object code, or executables. Only the update method understands the mapping between an object and its predecessor objects. Abstract coupling allows new classes of objects to be added to the dependency network without changing the client.

2. *Network structure.* Since the dependency network is explicitly represented, it can be changed dynamically and extended at any level. However, care must be taken to maintain consistency when dependency relationships change.

3. *Cost of propagation.* The cost of change propagation is driven by the structure of the particular dependency network and the cost of update. A simple propagation protocol only passes a reference to potentially affected objects, leaving it to the update methods of these objects to discern what changed and which objects are actually affected. By default, an update method will have to assume the worst case and rederive the entire object state. However, this cost can be reduced by avoiding redundant updates and by identifying changed portions (see discussion of Smart Propagation in section 2.7).

4. *Update completion.* It may not be desirable or possible to complete the update for all objects impacted by a change. First, the cost of the actual update may be high enough that the client is not willing to pay its cost up front, but wait until an impacted object is accessed. Second, the update of a dependant may fail, preventing propagation from actually reaching all dependants. This leads to the need for maintaining state information regarding the validity of objects in a dependency network.

1.7 Implementation

From the discussion above it is evident that a range of requirements and constraints may be placed on the propagator concept, leading to a family of patterns:

STRICT PROPAGATOR: Immediate propagation of updates without failure.

STRICT PROPAGATOR WITH FAILURE: Immediate propagation of updates with ability to handle failures.

LAZY PROPAGATOR: Demand-driven propagation of change in that changed objects are only marked. Dependent objects assure that they are up-to-date on demand.

ADAPTIVE PROPAGATOR: Immediate propagation of out-of-date indicators and separate propagation for re-establishing consistency in three variants: forward propagation of updates, on-demand update through backward chaining, and periodic update.

Each of these patterns is discussed in detail in the following sections.

2. STRICT PROPAGATOR

2.1 Intent

Define a network of dependent objects so that when one object changes state, all directly and indirectly dependent objects are updated immediately.

2.2 Also Known As

FORWARD | IMMEDIATE | EAGER PROPAGATOR.

2.3 Motivation

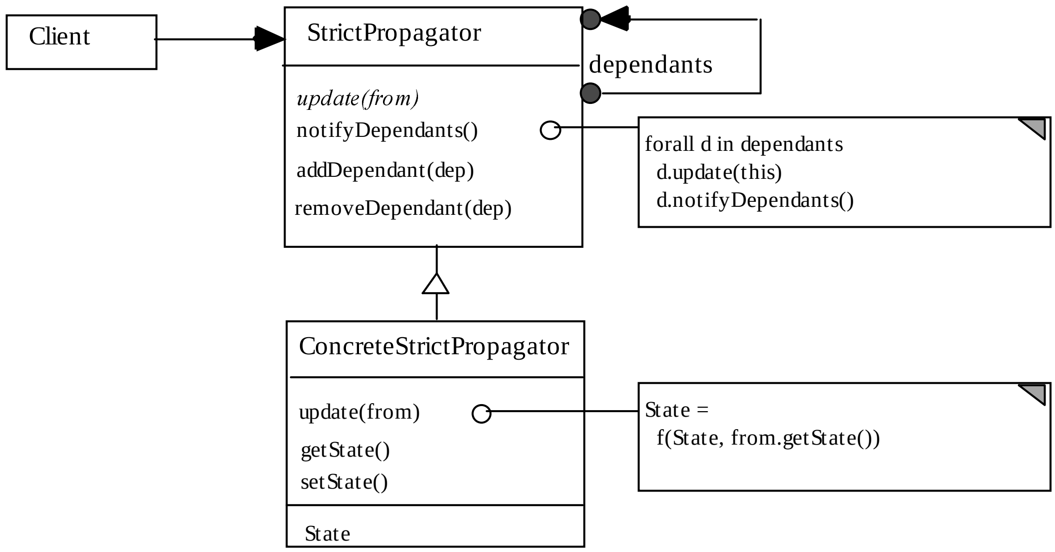
There is a class of applications that requires changes to be propagated immediately and the propagation processes to complete successfully. These assumptions lead to a simple realization of PROPAGATOR. This pattern is also effective in situations where immediate update is not required, but the delay in establishing consistency of an object on demand outweighs the cost of redundant updates.

This pattern is quite close to OBSERVER [Gamma95] in that it combines the methods of the subject and object classes. However, its notification method is recursive.

2.4 Structure

Some simplifying assumption have been made for this pattern:

- Updates always complete; no indication of success/failure is given.
- For the purpose of propagation, an object needs only keep track of its dependants. The changed predecessor is passed with the update method and is thus accessible to the dependant. This parameter allows the dependant to identify which of its predecessors has changed.



2.5 Participants

StrictPropagator

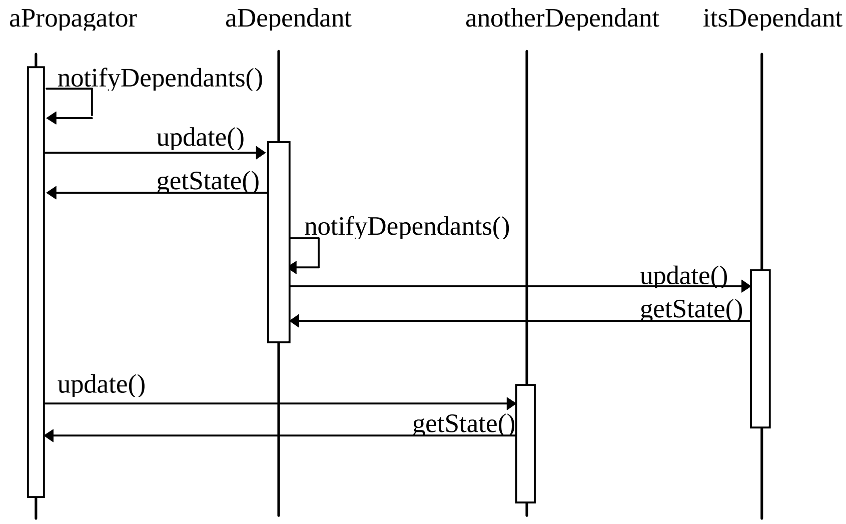
- knows its direct dependants. The direct dependants may have dependants of their own, and so on.
- provides a notification interface, i.e., a method for informing dependants to update themselves and their recursive dependants.
- provides an updating interface, i.e., a method for a dependent object to update itself given the changes of an object.
- provides a template method ([Gamma95]) for propagating the notifications and updates through the network recursively.
- provides methods for adding and removing direct dependants.

ConcreteStrictPropagator

- stores the state of an object.
- sends a notification to dependants when its state changes.
- implements the update interface.

2.6 Collaborations

- An instance of ConcreteStrictPropagator notifies its direct dependants whenever a change occurs that could cause dependants to become inconsistent.
- After notification, a dependant may query the notifying object for information and update itself accordingly. The dependant then notifies its own dependants, which in turn query for information and then notify their dependants, and so on.



2.7 Implementation

1. *Cycles in the dependency graph.* If the graph of dependencies contains cycles, notification and update can recurs forever. If it is infeasible to ensure that the dependency graph will remain acyclic, then a graph marking procedure should be used to terminate recursion. Alternatively, one can use smart propagation (see below) when it can be assured that repeated updates of an object within a cycle will not actually change its state and therefore terminate recursion.

2. *Smart propagation.* On update, each node can observe whether its content actually changes due to the change in the originating object. If it does not, then further notification of dependent objects is unnecessary. A variety of techniques are available, see [Adams94]. The simplest is *cutoff propagation*: An object simply performs its update and by comparison with its previous state (saved, for instance, by using MEMENTO [Gamma95]) determines whether it actually changed. If not, propagation can be cut off. If information about the nature of change is available, e.g., in form of a *delta*, then the dependent object may be able to decide a priori whether it is impacted by the particular part of the predecessor object state. A delta effectively encodes the difference between the old and new state. It can be viewed as a script that will reproduce the new state from the old. If the changes were small, the delta should be small.

3. *Compression and Deltas.* The object state may be quite large and must be accessible to dependent objects for update. In a distributed system, the amount of data transfer may become a bottleneck. Therefore, the use of compression techniques may be desirable. The sender can forward a compressed form of its state and let the receiver decompress it. A particularly effective compression technique involves sending a delta (see above). Alternatively, the protocol can be extended so the receiver can ask for exactly what it needs.

4. *Bundling updates from several predecessors (multiple predecessor paths).* In case of graph structured dependency networks, a dependent object may have several

predecessors that themselves are updated in response to a single changed object further back in the dependency structure. Using the simple propagation scheme, this object would be notified and updated several times. A more sophisticated algorithm informs every object at most once and ensures that all predecessor objects are up-to-date. This algorithm might use a topological sort of the dependency graph and propagate changes in the order given by the sort.

5. *Maintaining the dependency graph.* The simplest way for storing the dependency graph is to let each StrictPropagator maintain a list of dependants. Alternatively, the graph structure can be handled by a graph manager, for example using a hash table. This approach may save some space at the expense of speed. It also makes topological sorting for update bundling (see above) faster.

6. *Dynamically changing the dependency graph.* When a new edge is added to the dependency graph, the dependent object must usually be updated and its dependants notified. When an edge is removed, the same action may be necessary. However, during the initial construction of the network, the updates should be suppressed, because they may cause highly redundant propagations and even failures if there are uninitialized objects. Instead, the entire dependency graph should be built and its roots initialized before the update method is called on all roots.

7. *Depth-first vs. breadth-first propagation.* Depth-first propagation first notifies and updates a direct dependant and all its dependants before notifying any other direct dependant. Breadth-first propagation updates all direct dependants before recursing. The choice depends on the nature of the dependency network and whether elimination of redundant work is desired. For example, in networks with multiple predecessor paths (see above) breadth first propagation has the effect of a topological sort (without elimination of duplicates, which can be addressed by smart propagation or marking).

8. *Other aspects.* Most of the implementation hints about OBSERVER in [Gamma95] apply to STRICTPROPAGATOR as well.

2.8 Sample Code and Usage

The abstract class StrictPropagator can be defined in Java as follows. This implementation allows arbitrary, directed, acyclic graphs, where each StrictPropagator maintains a dynamic vector of its direct dependants. It is threadsafe, since all methods are synchronized. Thus, multiple updates can be in progress simultaneously. However, parallelism is limited since an object is locked until all dependants have been updated. The update operation has a parameter for telling it which predecessor object initiated the update notification. The notifyDependants operation performs a depth-first notification coupled with update. The operations for adding and removing dependants do not initiate propagation of update.

```
public abstract
class StrictPropagator {
    private Vector dependants;
    StrictPropagator(int initialCapacity) {
        dependants = new Vector(initialCapacity);
    }

    abstract synchronized
    void update(StrictPropagator from);

    public synchronized
    void notifyDependants(){
        Enumeration e = dependants.elements();
        StrictPropagator d;
        while (e.hasMoreElements()) {
            d = (StrictPropagator)e.nextElement();
            d.update(this);
            d.notifyDependants();
        }
    }

    public synchronized
    void addDependant(StrictPropagator d) {
        dependants.addElement(d);
    }

    public synchronized
    void removeDependant(StrictPropagator d) {
        dependants.removeElement(d);
    }
}
```

The class HeightProp has an extremely simple state, a single integer initialized to 0. The update method simply adds one to the state it reads from its predecessor.

```
public class HeightProp extends StrictPropagator {
    int counter;                        // this is the state.
    HeightProp(int initialCapacity) {
        super(initialCapacity);
        counter = 0;
    }

    public synchronized
    void update(StrictPropagator from) {
        counter = ((HeightProp)from).counter+1;
    }
}
```

Suppose instances of this class are linked in a tree structure and the counters are all initialized with zero. Then the following call assigns to each object its distance from root.

```
root.notifyDependants()
```

2.9 Known Uses

Examples of this pattern can be found in GUI systems that manage objects whose display properties depend on each other, e.g., objects nested graphically inside one another thus influencing sizing and clipping. Other examples can be found in spreadsheet applications that support automatic recalculation.

3. STRICT PROPAGATOR WITH FAILURE

3.1 Intent

Support a network of dependent objects so that when one object changes, dependent objects are updated, while tolerating update failure.

3.2 Also Known As

OPTMISTIC PROPAGATOR

3.3 Motivation

Some classes of applications cannot guarantee successful update of individual objects. Thus, the propagation must mark objects dependent on a failed update as invalid. One example is division by zero in a spreadsheet cell that is needed elsewhere.

3.4 Consequences

This pattern continues to localize all information relative to update consistency in the abstract StrictPropagator and methods associated with it. These methods are not impacted by the particulars of the objects in the network.

3.5 Implementation

We extend StrictPropagator with a valid marker. In this simple case, the valid marker needs only two values: true and false. True reflects that the last update has completed successfully, false that the update failed, i.e., that the object's state is corrupted or non-existent. The update method is changed to return a Boolean value indicating successful or unsuccessful completion.

Dependent objects must also be marked as invalid. An additional method, invalidateDependants, accomplishes this task. It visits dependent objects in the same order as notifyDependants.

The following state transitions are possible. They reflect a form of smart propagation of the invalid mark. They are built into notifyDependants and invalidateDependants, as appropriate.

Dependant Predecessor	Valid = T	Valid = F
Valid = F	Valid = F invalidateDependants()	Valid = F \\ no further propagation

Valid = T	Valid = update();
	if(Valid)
	notifyDependants();
	else
	invalidateDependants();
		Valid = update();
		if(Valid)
		notifyDependants();
		else
		\\ no further propagation

This simple technique must be extended when updates can reach an object via several paths. In this case, one propagation may mark an object as invalid, while the next propagation, coming in from a different edge, might promote it incorrectly to valid. To guard against this case, each invalid object might record from which predecessor the invalidation came.

3.6 Known Uses

This pattern is found in applications that derive object state from predecessor objects, but where the transformation might fail. An example are spreadsheet applications that handle arithmetic overflow. If a cell value is larger than the computational range or invalid (e.g., division by zero), propagation of the calculation is terminated and an appropriate symbol for the invalid value is displayed. This pattern can also be found in system building tools that automatically cause recompilation to be triggered upon edits – as found in interactive programming environments such as LOIPE [Feiler81].

4. LAZY PROPAGATOR

4.1 Intent

Support a network of dependent objects such that when objects change, other objects in the network can determine whether they are affected by the changes and bring themselves up-to-date.

4.2 Also Known As

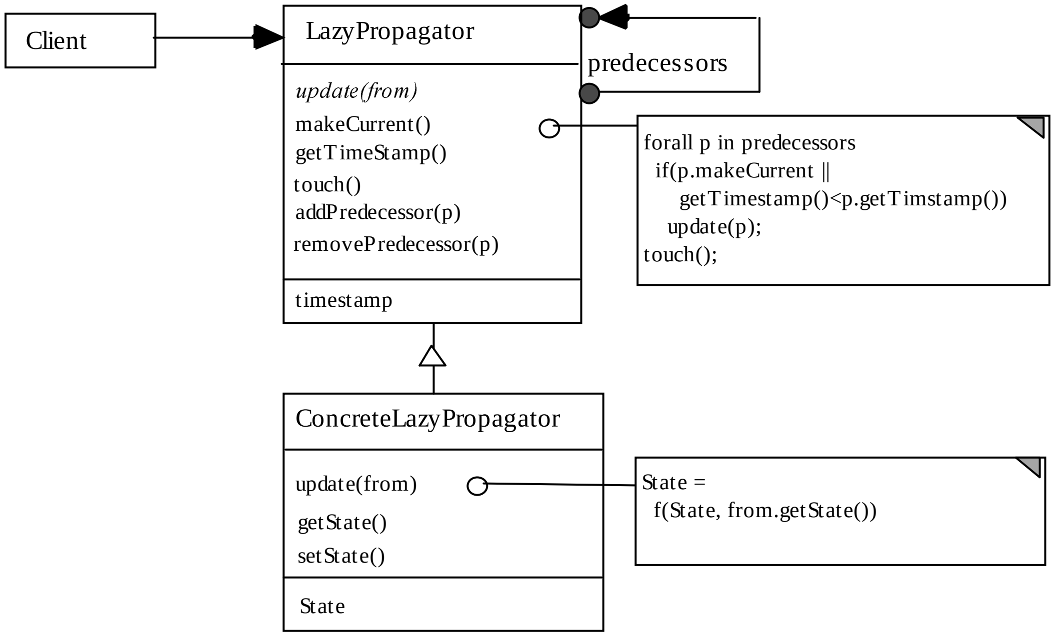
BACKWARD PROPAGATOR, UPDATE ON DEMAND.

4.3 Motivation

In some applications propagation of change can be expensive due to the cost of the update operation or the extent of the dependency network. If the application requires simultaneous access to only a small portion of the objects, e.g., only a subset of the objects are visible in a window, then only those objects need to be kept up to date. This may be accomplished at a lower cost by tracing back through the dependency network from the relevant target objects to bring them up to date. In some cases an explicit event requests an object to be brought up to date, such as an executable of a program managed by MAKE.

4.4 Structure

In this pattern, all objects have a timestamp reflecting the last modification time of an object's state. Whenever an object changes, it is marked with a new timestamp that is higher than the previously issued timestamp. When an up-to-date object is requested, it checks whether it depends (directly or indirectly) on any object with a newer timestamp than itself and updates itself (and its out-of-date antecedents) in that case.



4.5 Participants

LazyPropagator

- knows its direct predecessors. The direct predecessors may have predecessors of their own, and so on.
- provides an updating interface, i.e., how an object is changed when another changes.
- provides template methods for checking whether an object is up-to-date and for initiating updates if it is not.
- contains a timestamp to represent the last modification time.
- provides methods for adding and removing direct predecessors.

ConcreteLazyPropagator

- stores the state of interest.
- implements the updating interface.

TimestampManager

- runs a clock or records the highest timestamp given.
- provides a new timestamp value when a timestamp is updated.

4.6 Collaborations

An instance of ConcreteLazyPropagator compares the timestamps of its direct predecessors with its own. If its own timestamp is older than the timestamps of its direct predecessors, it is out of date. The predecessors use the same procedure recursively to determine whether they are out of date.

After an object finds it is out of date, it may query its predecessors for information and update itself accordingly. This update happens only after the predecessors have updated themselves in case they were out of date also.

4.7 Consequences

The consequences of LazyPropagator are analogous to those of StrictPropagator. Arbitrary subclasses of LazyPropagator can be combined in a single network. The dependency structure of the network can be varied dynamically. Predecessors can be added or deleted without modifying others.

4.8 Implementation

The implementation hints of StrictPropagator apply analogously. We list only those that require more discussion.

Smart propagation. Smart propagation backwards is a little trickier than with forward propagation. Suppose an object notices that an update did not change its content, so its dependants need not be updated (unless these objects depend on other changed objects). The timestamps of the unchanged object must be touched to reflect its currency, but downstream updates should not be triggered. This could be handled by passing back an indicator that says that the timestamp was reset, but the object was not changed. The receiving object can then suppress its update and touch its timestamp.

Timestamps. Some applications are able to utilize information already maintained with the objects in the dependency network and do not need to maintain a separate timestamp. For example, MAKE takes advantage of the modification date/time maintained for each file in order to determine whether a dependant is up-to-date. The cost of this approach is that the dependency network has to be repeatedly searched backwards to determine the validity of an object.

4.9 Sample Code and Usage

The abstract class LazyPropagator is defined (in Java) as follows. This implementation supports arbitrary, directed, acyclic graphs. Different from the strict propagators above, the links go in the opposite direction: Each LazyPropagator maintains a vector of its direct *predecessors.*

The lazy propagator needs to manage timestamps. The timestamp class below uses an increasing sequence of integers, starting with a value of 0. The current value of a timestamp can be read with getTimestamp and set to a new value with touchTimestamp. The timestamp manager is a SINGLETON with a static variable Clock that is incremented on every call to touchTimestamp.

```
public class Timestamp {
    static private int Clock = 0;
    // only one copy of Clock; starts with 0

    private int TimestampValue;

    Timestamp() {
        touchTimestamp();
    }

    public synchronized
    void touchTimestamp() {
        Clock++;
        TimestampValue = Clock;
    }

    public synchronized
    int getTimestamp() {
        return TimestampValue;
    }
}
```

The implementation of LazyPropagator is threadsafe and multiple updates can be in progress simultaneously. The interface of the update operation is identical with that for StrictPropagator. The specific code example assumes that the update always completes successfully. In case of update with failure, the update method returning a success indicator should be used, and the makeCurrent method should abort if an update fails.

The touch method sets the timestamp of an object to a new value. Touch must be called when an object is changed. MakeCurrent guarantees that an object is up to date by calling the method update where necessary. It recurses before it compares the timestamps of its direct predecessors. It touches an object only after the changes of *all* predecessors have been incorporated into the object. The example code passes the predecessor as parameter to the update method. In case of multiple predecessors, update has to be called for each changed predecessor to incorporate its changes. Only then can the timestamp be updated.

```
public abstract
class LazyPropagator {
    private Vector predecessors;
    private Timestamp ts;

    LazyPropagator(int initialCapacity) {
        predecessors = new Vector(initialCapacity);
        ts = new Timestamp();
    }


    public void touch() {     // sets timestamp to new value.
        ts.touchTimestamp();
    }

    public synchronized       // provides timestamp readout.
    int getTimestamp(){
        return ts.getTimestamp();
    }
```

```
    abstract void update(LazyPropagator from);

    // Makes itself current if it is out of date with
    // respect to any of its antecedents.
    // returns true if it had to make itself current, or else false.
    public synchronized
    Boolean makeCurrent() {
        Boolean madecurrent = false;
        Enumeration e = predecessors.elements();
        LazyPropagator p;
        while (e.hasMoreElements()) {
            p = (LazyPropagator)e.nextElement();
            if (p.makeCurrent() ||
                (getTimestamp() < p.getTimestamp())) {
                update(p);
                madecurrent = true;
            }
        }
        if (madecurrent) touch();
        return madecurrent;
    }

    public synchronized
    void addPredecessor(LazyPropagator p) {
        predecessors.addElement(p);
    }

    public synchronized
    void removePredecessor(LazyPropagator p) {
        predecessors.removeElement(p);
    }
}
```

Let’s use HeightProp from the previous section as an example of a concrete LazyPropagator, with the identical update method. If the dependency structure of instances forms a tree with all states initialized to zero, calling makeCurrent on all leaves would assign every node its height. Of course, if there are nodes with several predecessors, then the height computed depends on the order in which the predecessors are visited.

4.10 Known Uses

One well-known example is the use of this pattern in MAKE, a system build tool. Other examples include interactive environments with on demand compilation, i.e., compilation of program units the first time they are used. Some of the Web browsers like, e.g., Netscape maintain a cache of retrieved pages. They will request an update for those pages only after they exceeded a certain age - typically by removing them from the cache and forcing retrieval at the next page access.

5. ADAPTIVE PROPAGATOR

5.1 Intent

Support a network of dependent objects so that when objects change, dependent objects are updated in a flexible manner, adapting to the performance and responsiveness needs of the user.

5.2 Also Known As

ADAPTIVE UPDATE

5.3 Motivation

It should be clear that the update and lazy propagator classes can be merged. The only requirement is that dependency links can be traversed in both forward and backward directions and that addition and deletion of links are done consistently in both directions. Thus, it would be possible to run backward and forward propagations through the net, even simultaneously. Furthermore, we can exploit the combination to get performance advantages either alone would not afford.

A disadvantage of STRICT PROPAGATOR is the snowballing effect: A single change can cause a large number of updates. There can be waves of updates before the changed objects have ever been used. LAZY PROPAGATOR avoids this type of busy work, but trades it for another drawback: In order to be certain that an object is current, it must traverse the entire network backwards to the roots and check timestamps. This can be costly if the objects requested are far from the roots.

ADAPTIVE PROPAGATOR takes a middle ground in that it immediately forward propagates only the invalid marker and separately propagates updates optimistically, periodically, or on demand. These update modes may coexist, i.e., the user may switch between them.

Adaptive propagation avoids both of the above drawbacks as follows: Whenever an object changes, a forward propagation phase marks the affected objects as invalid, but does not actually update them. If there are several forward propagation phases before an update, then the propagation of invalid markers will stop quickly, because it will run into already marked objects. Thus we avoid unnecessary updates as well as repeated tracing of links.

Optimistic update propagation starts with the changed object and causes those objects that are affected by the change to be updated – at a pace slower than the propagation of the invalid marker and possibly concurrently with other activities on objects in the dependency network.

In the on-demand update mode, backward propagation does not start until an object is actually requested. However, the backward propagation only needs to follow links backward as far as the invalid markers are set, and may therefore not have to trace back all the way to the roots. If invalid markers are retracted in the order in which forward propagation sets them, then we can even let forward and backward propagation run simultaneously without causing inconsistency of the markers.

5.4 Participants

The abstract AdaptivePropagator differs from StrictPropagator by maintaining both dependant and predecessor lists, maintaining an invalid marker, and a combination of schemes for actually updating dependent objects affected by change.

5.5 Implementation

Update operations on individual objects may take a considerable amount of time. Therefore, it may be desirable to increase the degree of concurrency by permitting invalidations to occur in parallel with update operations on objects affected by the propagation. This can be handled by continuing to perform state changes in an atomic manner, and by introducing additional state values for reflecting the fact that update is in progress and that an invalidation has been received while an update was in progress. This results in the following state transition table.

state event	Invalid	UpdateInProgress	UpdateInvalid
start update	UpdateInProgress	N/A	N/A
invalidate	Invalid	UpdateInvalid	UpdateInvalid
update complete	N/A	Valid	Invalid
update failed	N/A	Invalid	Invalid

5.6 Known Uses

Spreadsheets are examples of multiple level dependency structures that operate in immediate update propagation mode, but allow this mode to be disabled for demand or periodic update.

Marvel [Kaiser88] is a program development environment that supports a combination of forward and backward triggering of automated tasks, such as recompilation and system building.

Other examples include distributed file systems and databases with and without replication, single- and multi-level caching (in both processors and file systems). Cache protocols typically invalidate the cache value at the time of modification, but delay cache update until the value is requested from the cache. Some distributed systems anticipate requests for replicated objects, e.g., based on recent access patterns or through look-ahead techniques, and will optimistically update some objects, while other objects are not retrieved until requested.

Simulation environments are examples of systems that typically use periodic update propagation. The simulation itself may run at a certain clock frequency – updating various state variables in the simulation environment, while the displayed objects are refreshed at a frequency that is driven by the ergonomics of human computer interaction.

6. Related Patterns from [Gamma95]

COMPOSITE can represent object dependencies in tree or graph structures.

ITERATOR can walk the graph of dependent objects.

VISITOR can update the various objects in a uniform way.

MEMENTO can save the state of an object before update, to see whether the update changes the state and should therefore propagate further.

TEMPLATE METHOD provides a skeleton for update propagation.

7. Bibliography

[Gamma95] E. Gamma et. al. Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley Professional Computing Series. 1995.

[Feiler81] P. H. Feiler and R. Medina-Mora, An Incremental Programming Environment, IEEE Transactions on Software Engineering, **7**(5), Sep. 1981, 472-482.

[Feldman79] S. I. Feldman, Make — A Program for Maintaining Computer Programs, Software — Practice and Experience, **9**(3), Mar. 1979, 255-265.

[Kaiser88] G. E. Kaiser, P. H. Feiler, and S. S. Popovich, Intelligent Assistance for Software Development and Maintenance, IEEE Software, **5**(3), May 1988, 40-49.

[Adams94] R. Adams, W. Tichy, and A. Weinert, The Cost of Selective Rccompilation and Environment Processing, ACM Transactions on Software Engineering and Methodology, **4**(1), Jan. 1994, 3-28.