EXEMPLARS: A Practical, Extensible Framework For Dynamic Text Generation

Michael White and Ted Caldwell
CoGenTex, Inc.
{mike,ted}@cogentex.com

Abstract

In this paper, we present EXEMPLARS, an object-oriented, rule-based framework designed to support practical, dynamic text generation, emphasizing its novel features compared to existing hybrid systems that mix template-style and more sophisticated techniques. These features include an extensible classification-based text planning mechanism, a definition language that is a superset of the Java language, and advanced support for HTML/SGML templates.

1 Introduction

In "NLG vs. Templates," Reiter [Reiter 95] points out that while template-based text generation tools and techniques suffer from many well-known drawbacks, they do nevertheless enjoy numerous practical advantages over most tools and techniques developed in the NLG community in many circumstances. These advantages include, among others, efficiency, simplified system architectures, full control over output, and much reduced demands on knowledge acquisition and representation. This leads Reiter to suggest that, from a practical perspective, one should use NLG techniques in hybrid systems that mix template-style and more sophisticated techniques; moreover, to facilitate adoption, NLG technologies should be developed so that they can be used without "getting in the way."

In line with this thinking, we have been developing EXEMPLARS, an object-oriented, rule-based framework for dynamic text generation, with an emphasis on ease-of-use, programmatic extensibility and run-time efficiency. *Exemplars* [Rambow et al. 98] are schema-like text planning rules that are so called because they are meant to capture an exemplary way of achieving a communicative goal in a given communicative context, as determined by the system designer. Each exemplar contains a specification of the designer's intended method for achieving the communicative goal. In the general case envisioned in Rambow et al., these specifications can be given at the level of intentional-rhetorical, conceptual, lexico-syntactic, or formatting/hypertext structures. The present framework currently supports specifications only at the level of formatting/hypertext structures — using any SGML-based representation, such as HTML — or RealPro abstract syntactic structures [Lavoie & Rambow 97]. A more complete range of specifications is instead supported in PRESENTOR [Lavoie & Rambow 98], a parallel implementation of the general approach with a complementary emphasis; while PRESENTOR emphasizes representation, we have instead emphasized extensibility and classification-based planning. In future work, we plan to merge the best of the two implementations.

In comparison to existing hybrid systems (e.g. [Reiter et al. 95]; [Milosavljevic et al. 96]; [Knott et al. 96]), we believe the present framework offers the following novel features:

- **Extensible classification-based text planning mechanism:** The text planner's rule selection mechanism involves a decision tree–style traversal of the exemplar specialization hierarchy, where the applicability conditions associated with each exemplar in the hierarchy are successively evaluated in order to find the most specific exemplar for the current context. Viewed in this way, the rule selection mechanism naturally forms the basis of an efficient, deterministic approach to text planning, where communicative actions are classified in context and then recursively executed, much as in [Reiter & Mellish 92]. In contrast to Reiter and Mellish's approach, however, we emphasize extensibility, supporting *inter alia* discourse-sensitive conditions.
- **Java-based definition language:** Exemplars are defined using a superset of Java, and then compiled down to pure Java. This approach makes it possible to (i) reuse basic Java constructs as well as Java's inheritance mechanism, exceptions, threads, etc., (ii) directly and efficiently integrate with other application objects, and (iii) take advantage of advanced Java-based system architectures.
- **Advanced HTML/SGML support:** With exemplars, the designer can bootstrap the authoring process using existing HTML or (normalized) SGML, then annotate the specification to produce dynamic content. Moreover, in contrast to other HTML template approaches (e.g. that provided with JavaSoft's Java Web Server [Sun 98]), we allow the designer to generate HTML in a truly hierarchical fashion.

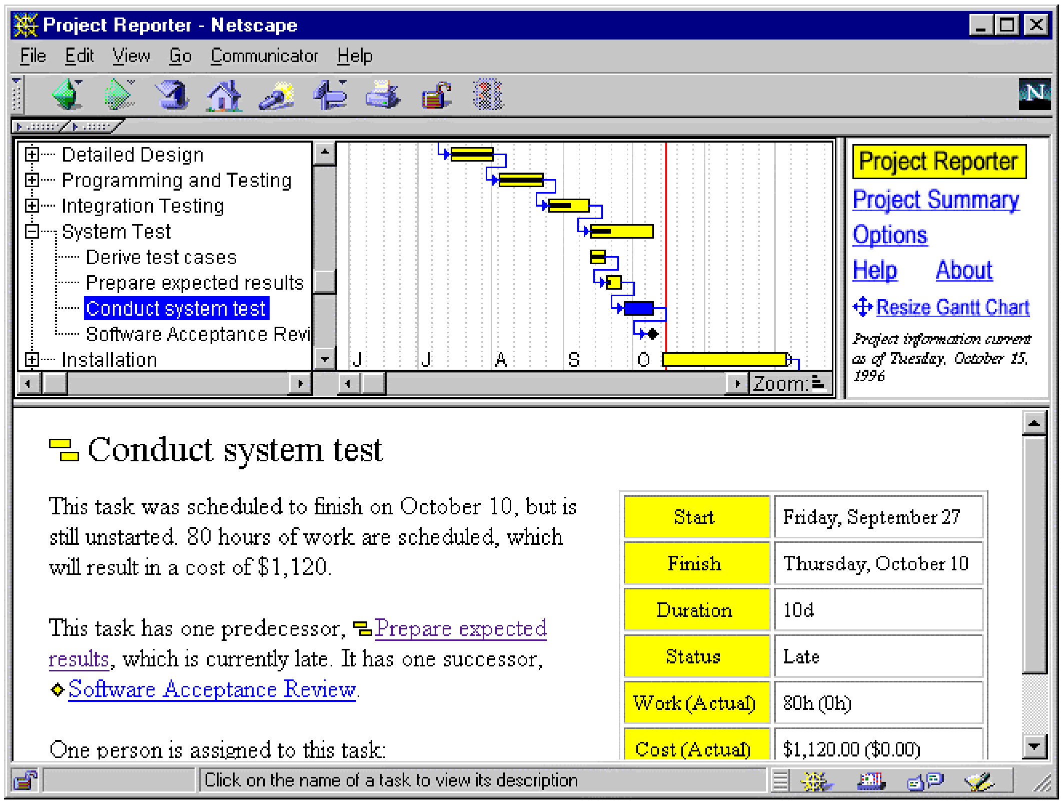
To date we have developed three systems[1] with the framework at CoGenTex — namely the Project Reporter [CoGenTex 97], CogentHelp [Caldwell & White 97] and EMMA systems [McCullough et al. 98] — and are currently engaged in using it to develop a natural language query tool for a large data warehousing company. The framework has benefited substantially from feedback received during its use with these projects.

The rest of this paper is organized as follows. In Section 2, we describe how the EXEMPLARS framework can be used to dynamically generate HTML using objects from an application object model. In Section 3, we focus on the role of specialization and extensibility in managing textual variation. In Section 4, we compare our classification-based approach to text planning to that of [Reiter & Mellish 92], as well as to systemic and schema-based approaches, plus HTML template approaches taken outside the NLG community. In Section 5, we conclude with a discussion of the types of generation systems for which we consider the framework to be appropriate.

[1] Project Reporter is currently in the pre-beta release stage of development; CogentHelp and EMMA are operational prototypes.

2 Dynamically Generating HTML

In this section we sketch how the EXEMPLARS framework can be used to dynamically generate HTML, using Project Reporter as an example. Project Reporter is an innovative web-based tool for monitoring the status of a project. Using information obtained from a project management database, Project Reporter automatically generates fluent natural-language reports describing task progress, staffing, labor expenditures, and costs for a project. It also displays project data in tables and in Gantt chart form,



Project information current as of Tuesday, October 15, 1996

Conduct system test

This task was scheduled to finish on October 10, but is still unstarted. 80 hours of work are scheduled, which will result in a cost of $1,120.

This task has one predecessor, Prepare expected results, which is currently late. It has one successor, ◆ Software Acceptance Review.

One person is assigned to this task:

Start	Friday, September 27
Finish	Thursday, October 10
Duration	10d
Status	Late
Work (Actual)	80h (0h)
Cost (Actual)	$1,120.00 ($0.00)

Figure 1: Project Reporter screenshot

providing a complete multimodal view of the project's status. See Figure 1 for a screenshot.

The main idea behind the EXEMPLARS framework is to enable the designer to determine the behavior of a dynamic (hyper-) text generation system by writing a set of object-oriented text planning rules and arranging them into a specialization hierarchy, where more specialized rules can augment or override the more general ones they specialize. By text planning rules, we mean rules that determine the content and form of the generated text. Each such rule has a condition and an action: the condition defines the applicability of the rule in terms of tests on the input application objects, the discourse context, and the user model, whereas the action defines what text to add to the current output and how to update the discourse context and user model.

For purposes of exposition, we have extracted a subset of Project Reporter's exemplars and simplified them to use a much-reduced application object model, part of which is shown in the UML diagram in Figure 2. As the diagram indicates, tasks and milestones in a project are represented using Task and Milestone classes, where their common features are abstracted into the base class TaskOrMilestone. To generate an HTML page for a task or milestone along the lines of the one shown in Figure 1, the generator invokes the top-level exemplar ShowTaskOrMilestone with the given task or milestone. This exemplar sets up the HTML page, adds a table with basic data, and then calls other exemplars, including the DescribeTaskOrMilestoneStatus exemplar, to add appropriate text for the task or milestone. The call from ShowTaskOrMilestone to DescribeTaskOrMilestoneStatus is reflected in the UML diagram in Figure 3 by the dependency arrow between the two. The other two dependency arrows in the diagram show that

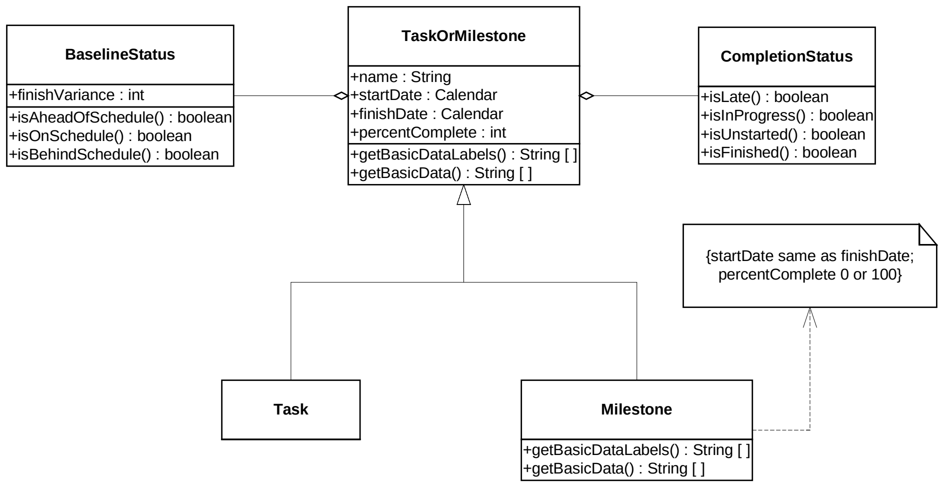

BaselineStatus
+finishVariance : int
+isAheadOfSchedule() : boolean
+isOnSchedule() : boolean
+isBehindSchedule() : boolean
TaskOrMilestone
+name : String
+startDate : Calendar
+finishDate : Calendar
+percentComplete : int
+getBasicDataLabels() : String [ ]
+getBasicData() : String [ ]
CompletionStatus
+isLate() : boolean
+isInProgress() : boolean
+isUnstarted() : boolean
+isFinished() : boolean
{startDate same as finishDate; percentComplete 0 or 100}
Task
Milestone
+getBasicDataLabels() : String [ ]
+getBasicData() : String [ ]

Figure 2: Simplified project model with tasks and milestones

DescribeTaskOrMilestoneStatus in turn makes use of AddBaselineStatusModifier and IdentifyDate.

A call to DescribeTaskOrMilestoneStatus produces text such as *This task started last Saturday, June 13, and is scheduled to finish July 9, three days ahead of the baseline schedule. It is currently 10% complete.* Exemplar calls are mediated by the text planner component of the framework, which automatically selects the most specific applicable exemplar to actually apply in the current context. For example, the above text would be produced by the DescribeTaskStatus exemplar, whose source is shown in Figure 4; this exemplar is chosen when the given task or milestone is in fact a task, and is currently in progress.

To find the most specific applicable exemplar, the text planner traverses the specialization hierarchy top-down and left-to-right,[2] evaluating applicability as it goes; the traversal is similar to that of a decision

[2] Left-to-right order is determined by the order in which exemplar definitions appear in the source file.

tree, insofar as applicability conditions are implicitly conjoined, except that the search may end at a non-leaf node in the hierarchy. To determine whether a particular exemplar is applicable, the text planner first checks the type constraints on the exemplar's arguments, then evaluates any explicitly defined conditions. Once the most specific applicable exemplar is found, its exclusion conditions (if any) are checked, to see if it is optional and should be skipped in the given context; if not, its action is invoked at this point. In the case of the exemplars shown in Figure 3, the exemplars under DescribeTaskOrMilestoneStatus define their applicability conditions using the methods of the task or milestone's associated CompletionStatus object; the exemplar AddBaselineStatusModifier is optional, and defines its exclusion conditions using

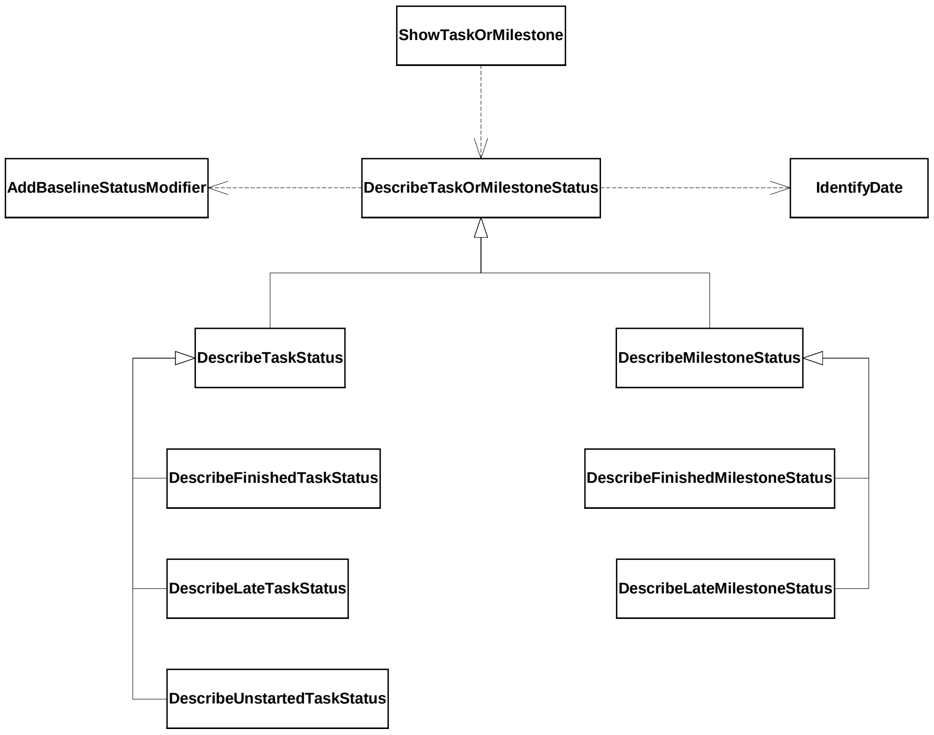
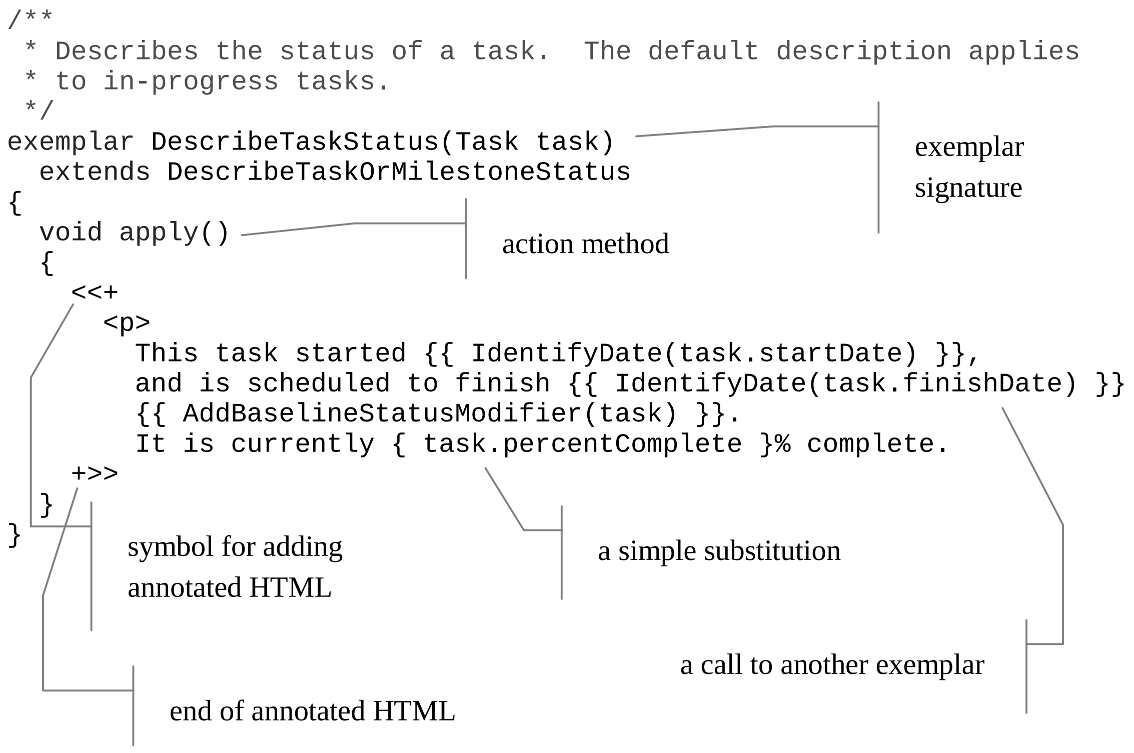

ShowTaskOrMilestone
AddBaselineStatusModifier
DescribeTaskOrMilestoneStatus
IdentifyDate
DescribeTaskStatus
DescribeMilestoneStatus
DescribeFinishedTaskStatus
DescribeFinishedMilestoneStatus
DescribeLateTaskStatus
DescribeLateMilestoneStatus
DescribeUnstartedTaskStatus

Figure 3: Exemplars for showing the status of a task or milestone

the methods of the task or milestone's associated BaselineStatus object.

As mentioned in the introduction, exemplars are defined using a superset of Java, and then compiled down to pure Java. A source file normally contains a set of related exemplars, each of which is compiled into a separate Java class files (though all in the same package). The compilation process consists of (i) translating the signature into a set of constructors and methods that implement it; (ii) translating any statements to add annotated HTML, including any embedded simple substitutions or exemplar calls, such as those shown in Figure 4; and (iii) passing through any Java code in-between. While the DescribeTaskStatus exemplar in Figure 4 happens to not contain additional Java code, exemplars often contain loops, local variables, try-catch blocks, auxiliary methods, and so on.

From the perspective of everyday object-oriented design, the way in which exemplars are treated as first-class objects is quite similar to the way methods are promoted to objects in certain design patterns, such as the Command pattern in [Gamma et al. 95]. While the Command pattern promotes methods to objects so that they can be tracked and possibly undone, the primary reason for doing so with exemplars is so that they can become essentially self-specializing. Additionally, the way in which type constraints are used in specialization resembles the Visitor pattern, whose primary purpose is to group closely related methods for various classes in one place, rather than having them scattered over many classes. Exemplars are much more flexible than visitors, however, since they support arbitrary applicability conditions plus type-



```
/**
 * Describes the status of a task.  The default description applies
 * to in-progress tasks.
 */
exemplar DescribeTaskStatus(Task task)
  extends DescribeTaskOrMilestoneStatus
{
  void apply()
  {
    <<+
      <p>
        This task started {{ IdentifyDate(task.startDate) }},
        and is scheduled to finish {{ IdentifyDate(task.finishDate) }}
        {{ AddBaselineStatusModifier(task) }}.
        It is currently { task.percentComplete }% complete.
    +>>
  }
}
```

Figure 4: Exemplar describing the status of a task

based selection on multiple arguments.

3 Specialization and Extensibility

To further illustrate the role of specialization and extensibility in managing textual variation, we will now examine how dates are referred to in a context-sensitive way in Project Reporter. The IdentifyDate exemplar referenced in Figure 4, along with its specializations, form a separate reusable package. These exemplars are shown in the annotated UML diagram in Figure 5.

The IdentifyDate exemplars yield descriptions that are sensitive both to the current date and the last date mentioned. If the given date is the same as the last one mentioned, the phrase *the same day* is used, as in

This task is scheduled to start next Thursday, June 25, and to finish the same day. If the given date is not the same as the last one mentioned, the phrase used depends on how close it is to the current date, as Figure 5 shows.

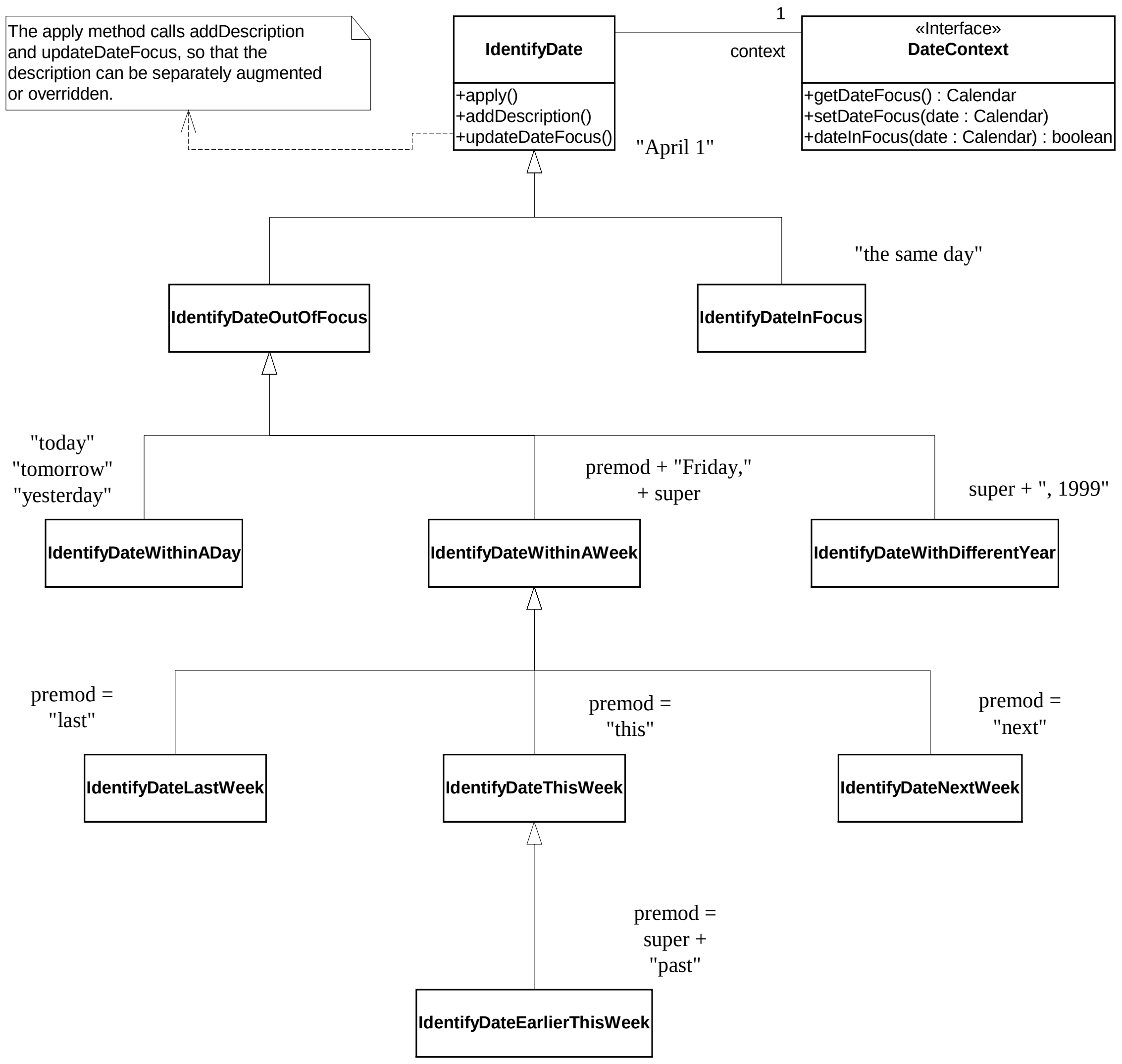


Figure 5: Exemplars for contextually identifying dates

Several aspects of the diagram merit further explanation. First, the context object associated with each instance of IdentifyDate is required to implement the DateContext interface, which provides the indicated methods for tracking the last date mentioned. Second, the IdentifyDate exemplar breaks down its apply method (its action) into two methods, addDescription and updateDateFocus, so that the description can be separately specialized in descendant exemplars. Third, it should be emphasized that the ability to extend the framework in this way follows from the treatment of exemplars as first-class objects plus the definition language allowing for arbitrary Java code. Finally, note that while some exemplars, such as IdentifyDateInFocus and IdentifyDateWithinADay, simply override the description of their parent exemplar (a.k.a. their ‘super’, in Java terminology), most of these exemplars instead augment the less specialized phrase with additional modifiers.

4 Related Work

Following [Reiter & Mellish 92], we view the process of selecting the most specific exemplar to be one of rule classification. In their approach, the process begins by forming a rule instance with appropriate fillers, which is then classified in the taxonomy of rules. Once the rule instance has been classified, any relevant (possibly inherited) attributes are read off, and its particular action is invoked. Our approach is very much the same, though the terms are slightly different: first an exemplar instance is constructed and initialized with the given arguments; this instance is then classified in the exemplar specialization hierarchy (making use of more specialized but otherwise equivalent exemplar instances); once the most specific applicable exemplar has been found, its particular action is likewise invoked, unless its (possibly inherited) exclusion conditions are true.

The primary way in which our approach differs from that of Reiter and Mellish is in the choice of classification procedure: while we employ a simple decision tree–style traversal of a tree-structured specialization hierarchy, they make use of a more sophisticated classification algorithm in a potentially more complex taxonomy. The particular classification algorithm they employ is the one built-in to I1, the knowledge representation system used in their IDAS system. While this algorithm potentially offers more in the way of automated reasoning, in our view it suffers from the inability to easily make use of dynamically determined constraints, such as those pertaining to the addressee or the discourse context.

As Reiter and Mellish point out, their classification-based approach to planning closely resembles systemic approaches (e.g. [Vander Linden & Martin 95]), especially insofar as both are deterministic choice makers (in contrast to unification-based systems). In a sense, our approach is even more closely related to systemic ones, at least those that allow arbitrary Lisp code in the choosers that determine the features used in systemic network traversal. However, what Reiter and Mellish fail to point out is the rather different flavor of classification vs. systemic network traversal: in a classification-based approach, no action is taken until the most specific rule is found; in contrast, in the systemic approach, action (in the form of executing realization statements) is performed as the network is traversed. As we saw in the preceding section, it is possible for a more specific rule to augment the action of its parent in the hierarchy, yielding much the same behavior as with systemic network traversal; nevertheless, it should be emphasized that this is not required in a classification-based approach.

Turning now to schema-based approaches such as that of [Lester & Porter 97], beyond the obvious differences of representation and the absence of classification, one way in which our approach differs is that we have explicitly embraced a powerful object-oriented programming language, rather than simply embedding a handful of procedural constructs. Since schemas are interpreted rather than reasoned about as formal objects (as in AI planning approaches), we suggest that the added flexibility of building upon an advanced programming language more than offsets any loss in declarativity.

Beyond flexibility, the Java basis of EXEMPLARS provides numerous further practical benefits. Perhaps foremost of these is that with just-in-time compilers, the compiled Java code supports the performance demands of interactive web applications. An important factor in this picture is the ability to directly integrate with application objects, rather than integrating indirectly via some interpretive scheme; direct integration also offers better static checking than is usually possible otherwise. Another key practical benefit of the approach is that it becomes possible to take advantage of advanced Java-based system architectures, such as the Java Servlet API [Sun 98].[3]

Finally, looking outside the NLG community, it is worth observing that while we have seen the emergence of numerous template-style HTML generation frameworks — e.g., the page compilation facility included with the Java Web Server — none include anything like our extensible classification-based planning mechanism, or even the ability to generate HTML in a truly hierarchical fashion. Typically, these frameworks embed special tags and code into HTML, rather than the other way around; while limiting in many respects, note that the mainstream approach does promise a more streamlined authoring process.

5 Conclusion

In this paper, we have presented EXEMPLARS, an object-oriented, rule-based Java framework designed to support practical, dynamic text generation, emphasizing its novel features compared to existing hybrid systems. To date the framework has been used with success in three projects at CoGenTex and in one outside consulting effort. Based on this experience, we suggest the framework is well suited to monolingual generation systems of moderate to high complexity, especially those of a highly application-specific nature and with significant performance demands.

Acknowledgements

We thank Ehud Reiter, Regina Barzilay, Tanya Korelsky, Richard Kittredge, Benoit Lavoie, Daryl McCullough, David Post, and Owen Rambow for feedback and advice. Kudos also to the JavaCC folks for making the exemplars compiler possible. This work has been partially supported by Rome Laboratory contract F30602-94-C-0124 and DARPA contract F30602-96-C-0220.

[3] *Servlets* are the server-side analogues of applets, insofar as they are specialized mini-servers that can be dynamically loaded into a web server.

References

[Caldwell & White 97] Caldwell, D. E., and M. White. 1997. CogentHelp: A tool for authoring dynamically generated help for Java GUIs. In *Proceedings of the 15th Annual International Conference on Computer Documentation (SIGDOC '97)*, pp. 17–22, Salt Lake City, UT.

[CoGenTex 97] CoGenTex, Inc. 1997. Text Generation Technology for Advanced Software Engineering Environments. Final Report, Rome Laboratory Contract No. F30602-92-C-0163.

[Gamma et al. 95] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

[Knott et al. 96] Knott, A., C. Mellish, J. Oberlander, and M. O'Donnell. 1996. Sources of Flexibility in Dynamic Hypertext Generation. In *Proceedings of the Eighth International Natural Language Generation Workshop (INLG '96),* pp. 151–160, Herstmonceux Castle, Sussex, UK.

[Lavoie & Rambow 97] Lavoie, B., and O. Rambow. 1997. A Fast and Portable Realizer for Text Generation Systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP '97),* pp. 265–268, Washington, D.C.

[Lavoie & Rambow 98] Lavoie, B., and O. Rambow. 1998. A Framework for Customizable Generation of Multi-Modal Presentations. To appear in *Proceedings of the 36th Meeting of the Association for Computational Linguistics (ACL '98),* Montréal, Canada.

[Lester & Porter 97] Lester, J. C. and B. W. Porter. 1997. Developing and Empirically Evaluating Robust Explanation Generators: The KNIGHT Experiments. . *Computational Linguistics*, vol. 23, no. 1, pages 65–100.

[McCullough et al. 98] McCullough, D., T. Korelsky, and M. White. 1998. Information Management for Release-based Software Evolution Using EMMA. To appear in the *Proceedings of the Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE '98),* San Francisco Bay, CA.

[Milosavljevic et al. 96] Milosavljevic, M., A. Tulloch, and R. Dale. 1996. Text generation in a dynamic hypertext environment. In *Proceedings of the 19th Australasian Computer Science Conference*, pp. 229–238, Melbourne, Australia.

[Rambow et al. 98] Rambow, O., D. E. Caldwell, B. Lavoie, D. McCullough, M. White. 1998. Text Planning: communicative intentions and the conventionality of linguistic communication. In preparation.

[Reiter & Mellish 92] Reiter, E., and C. Mellish. 1992. Using classification to generate text. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL '93)*, pp. 265–272, Newark, Delaware.

[Reiter 95] Reiter, E. 1995. NLG vs. templates. In *Proceedings of the Fifth European Workshop on Natural Language Generation (EWNLG '95),* Leiden, The Netherlands.

[Sun 98] Sun Microsystems, Inc. 1998. Java Web Server.
http://jeeves.javasoft.com/products/webserver/index.html

[Vander Linden & Martin 95] Vander Linden, K., and J. H. Martin. 1995. Expressing Rhetorical Relations in Instructional Text: A Case Study of the Purpose Relation. *Computational Linguistics*, vol. 21, no. 1, pp. 29–58.