

ConAg: A Reusable Framework for Developing “Conscious” Software Agents

Myles Bogner¹, Jonathan Maletic, and Stan Franklin²

Institute For Intelligent Systems
Department of Mathematical Sciences Division of Computer Science
The University of Memphis
Memphis, TN 38152
{bognerm,maleticj,franklins}@msci.memphis.edu

1. Abstract

ConAg is a reusable framework, written in Java, for creating “conscious” software agents. Its particular focus is on these agents’ “consciousness” mechanism. A “conscious” software agent is an cognitive agent that integrates numerous artificial intelligence mechanisms to implement Bernard Baars’ global workspace theory, a psychological theory of mind. This article gives overviews of “conscious” software agents and the need for a software reuse strategy for these agents’ development. It describes the architectural style of “conscious” software agents, from which ConAg is built. ConAg is then discussed, including the design patterns and components utilized in this framework.

2. An Introduction to “Conscious” Software Agents

For the past several years, the “Conscious” Software Research Group (CSRG)³ has been developing “conscious” software agents. From the onset, and continually more so as development progresses, it is clear that these agents are extremely complex and time-consuming to develop and implement. This is understandable in that this software is designed to be “smarter” software. “Conscious” software systems range in functionality, from academic seminar organizers (Bogner, Ramamurthy, and Franklin, 1999), to naturalistic tutors (Graesser, Franklin, & Wiemer-Hastings, 1998), to navy detailers responsible for naval personnel placement (Franklin, Kelemen, & McCauley, 1998), to personal travel agents. ConAg, the “Conscious” Agent Framework, is being developed to help facilitate these agents’ construction. ConAg closely follows current software reuse development methodologies. By doing so, it is hoped that a reusable framework is developed which expedites the building of these agents.

Autonomous agents (Franklin & Graesser, 1997) are systems situated in and part of an environment. They sense this environment and act on it over time, in pursuit of their own agenda, in such a way as to possibly influence what they sense at a later time. Autonomous agents are coupled to their environment. They include numerous animals such as humans and software agents such as some computer viruses and artificial life agents. Many of these agents are also cognitive agents (Franklin, Stan, 1997; Bogner, Ramamurthy & Franklin, 1999). Cognitive agents are autonomous agents equipped with cognitive features such as perception, concept formation, attention, learning, short and long-term memory, emotions, and consciousness. In this article, these cognitive features are used both in the folk-psychological and technical senses.

“Conscious” software agents (Bogner, Ramamurthy & Franklin, 1999; Ramamurthy, Bogner, & Franklin, 1998; McCauley & Franklin, 1998; Zhang, Franklin, & Dasgupta, 1998; Bogner, 1998) are cognitive agents that implement global workspace theory, a psychological theory of consciousness (Baars, 1988; Baars, 1997). Global workspace theory postulates that human cognition is implemented by a multitude of relatively small, special purpose processes, almost always unconscious. In this multi-agent system, coalitions of such processes, when aroused by novel and/or problematic situations, compete for access into a global workspace and, therefore, into consciousness. This limited capacity workspace serves to broadcast the coalition’s message to all the unconscious processors, in order to recruit other processors to join in handling and solving the current novel situation or problem. The theory describes additional elements of mind such as action selection, learning, and problem solving. “Conscious” software agents are very domain-specific entities and contain many cognitive features such as emotions (McCauley & Franklin, 1998), behaviors (Song, 1998), and learning (Ramamurthy, Bogner, & Franklin, 1998; Ramamurthy, Franklin, & Negatu, 1998, Bogner, Ramamurthy, & Franklin, 1999).

This article is intended to appeal to several audiences including the cognitive-modeling and software reuse communities. As such, this paper first discusses why it is important that ConAg relies heavily on software reuse. ConAg is then discussed under the auspices of software reuse methodology. The architectural styles of “conscious” software agents, which constrain ConAg, are described. The design patterns utilized by the framework are discussed. The framework itself is thoroughly presented, including its component-based structure. In addition, ConAg’s design patterns are presented.

3. Why Is A Focus On Reuse So Important?

Software reuse is the use of existing software to create new software instead of building the new system from scratch (Krueger, 1992). Reuse involves using both previously defined higher-level concepts such as ideas and knowledge and lower-level specific components such as objects in new situations. When creating “conscious” software agents, both high-level concepts and code are often reused. The software reuse process is commonly thought to involve three steps, each of which is needed for a “conscious” agent framework (Basili, Briand, & Melo, 1996; Prieto-Días & Freeman, 1987):

1. Accessing and choosing a reusable artifact.
2. Understanding and adapting the artifact to the application’s purpose.
3. Integrating the artifact into the product currently being developed.

In general, portions which can be reused include design structures, documentation such as manuals and specifications, and source code. ConAg does just this. Also, reuse involves both black-box techniques, utilizing a component as is, and white box techniques or code scavenging where existing components are modified to fit the needs of the new system. ConAg provides for both of these techniques. In general, adaptation is much more common than straight reuse as available components usually do not match the desired functionality. Later sections describe which portions of ConAg are most likely to need white-box reuse.

Software reuse is often not standard practice in software development organizations (Krueger, 1992). ConAg hopes to help change this for the CSRG. As is the case for the CSRG, reuse is difficult as abstractions for large and complex systems are typically complicated. It is often difficult for developers to learn these abstractions. Research has shown that a reuser's skill is important in determining their levels of reuse (Prieto-Días & Freeman, 1987). Many developers are not trained in reuse, and those with training are often not pressed to practice it. To make reuse attractive, an organizational-wide classification scheme is often needed. Also, the effort to use existing code must be less than the effort needed to write new code (Prieto-Días & Freeman, 1987). For example, without an adequate classification scheme, reuse becomes less attractive as it is difficult to distinguish between similar items. ConAg places great emphasis on classification in the hope of providing an appealing framework.

While software techniques are involved and difficult to utilize, their benefits to ConAg's design, development, and use by others far outweighed these complexities. In general, reuse techniques have been gaining momentum as they have potential to significantly reduce development costs, maintenance costs, and unrealizable schedules. Software reuse has repeatedly been suggested as a means for successfully combating the software crisis: the problem of building large and reliable software systems at a reasonable cost (Krueger, 1992; Mellor & Johnson, 1997). Reusability is widely believed to be a key to improving software development quality (Biggerstaff & Richter, 1997). Reuse results in a completed systems' containing fewer total symbols with less time having been spent on the symbols' organization. Therefore, in a sense, reuse enhances software developers' capabilities, and most developers prefer to reuse than write code from scratch when given the option (Frakes & Fox, 1995). Through reuse, ConAg hopes to capitalize on many of these benefits.

According to Biggerstaff and Richter (1987), a good reuse system addresses four problems. Each of these issues is addressed by ConAg.

1. The ability for developers to be able to find necessary components, both exact and similar matching ones.
2. A means for easily understanding the components. This is particularly key when components need modification.
3. A method by which components can be modified in order to apply them to new domains.
4. A way to appropriately document newly composed components. This representation should illustrate the composed components both as independent entities as well as showing how they can be modified to fit new domains.

Software reuse involves four dimensions, each found within ConAg (Krueger, 1992).

1. Abstraction is the central feature of software reuse. Abstraction allows for a succinct description of an item, highlighting the important information while leaving out what is unimportant. A common example of an abstraction technique are object-oriented languages' class definitions. The provision of these languages for inheritance also

- allow for a reuse class hierarchy. These subtype hierarchies are helpful in finding reusable items.
2. Selection provides classification schemes for organization and the finding of reusable artifacts. Selection works well when the representation is clear on what the artifact does. To be effective, the classification schemes must allow developers to find components faster than write them.
 3. Specialization allows developers to modify general components to fit their specific needs.
 4. Integration allows developers to combine their specialized components into a new software system.

There are many techniques which when utilized help foster software reuse. Some of the main techniques are the use of architectural styles, design patterns, and objects (Mellor & Johnson, 1997). Each of these will be discussed at length below in conjunction with a description of how ConAg uses them.

4. ConAg's Architectural Style

Building on the findings that object-oriented development has been shown to significantly increase productivity, ConAg is developed in Java using object oriented methods. ConAg's architecture rests on a design philosophy that directs "conscious" software agents' development. It is worthy to study ConAg's design philosophy as research has illustrated that design reuse does have several advantages over simple code reuse (Johnson, 1997).

4.1. What Are Architectural Styles?

Design reuse is common as it can be applied to many contexts. In addition, as has been the case in the development of "conscious" software agents, the design process can be applied earlier in the development process, and, thereby having a larger impact on the project. Also, true to form with "conscious" software agent development, most design reuse is informal and happens with experienced developers (Johnson, 1997). Design reuse allows for open systems, and it allows the "Conscious" Software Research Group's developers to share a common vision.

Architectural styles are a form of design reuse. Architectural styles provide a collection of constraints, building-block design elements, and rules for composing a system (Monroe, Kompanek, Melton, & Galran, 1997). There are several benefits to architectural style usage. For example, routine solutions with well-understood properties can be reapplied to new problems with confidence, potentially leading to significant code reuse. In addition, architectural styles can be applied to a broad range of problems (Mellor & Johnson, 1997), such as the different domains for "conscious" software agents.

Each architectural style has its own notation, or specialized design language, (Mellor & Johnson, 1997; Monroe, Kompanek, Melton, & Galran, 1997) describing:

- The structural and semantic properties of systems falling within the style.
- A common vocabulary such as “blackboard system,” “client-server system,” and “database.” A semantic interpretation is also provided so that the composition design elements have well defined meanings.
- The patterns of interaction of systems built within the style. These design rules (constraints) determine which design element compositions are permitted. For example, all “conscious” software agents’ processors have access to a single blackboard.
- Analyses that can be performed on systems built in the style.

4.2. “Conscious” Software Agent’s Architectural Style

“Conscious” software agents are designed following the action selection paradigm, a design philosophy providing principles for cognitive agent architectures (Franklin, 1997; Franklin, 1995). The action selection paradigm states that minds are autonomous agents’ control structures. Minds’ task are to produce the next action. Minds should be viewed as continuous instead of boolean. Sensations, such as perception, are operated on by minds to create information for their own use. A multitude of disparate mechanisms enable minds, and there is little communication between them. Minds and action selection are limited to autonomous agents. Agents are situated in environments, and agents’ actions are selected in the service of drives. Prior-information (memories) are re-created to help produce actions. Cognitive functions such as categorizing, inferencing, planning, recalling, recognizing, and sensing all serve to help determine what to do next.

“Conscious” software agents also fall under Baars’ global workspace theory. Particularly important from the theory is that the system is comprised of numerous small processors, known as codelets (Hofstadter & Mitchell, 1994) in “conscious” software agents. Some of these processors form coalitions and compete for consciousness. When a coalition reaches consciousness, its information is broadcast to the entire system. Becoming conscious is sufficient for learning. Processors act under the auspices of contexts: conceptual contexts, cultural contexts, goal contexts, and perceptual contexts. Each context is a coalition of processors.

Pandemonium theory (Jackson, 1987) is also key to these agents’ design. Pandemonium theory’s components interact like people in a sports arena. Both the fans and players are known as demons. Demons can cause external actions, they can act on other internal demons, and they are involved in perception. The vast majority of demons are the audience in the stands. There are a small number of demons on the playing field. These demons are attempting to excite the fans. Audience members respond in varying degrees to these attempts to excite them, with the more excited fans yelling louder. The loudest fan goes down on the playing field and joins the players, perhaps causing one of the players to return to the stands. The louder fans are those who are most closely linked to the players. There are initial links in the system. Links are created and strengthened by the amount of time demons spend together on the playing field and by the system’s overall motivational level at the time.

Figure 1 illustrates “conscious” software agents’ high level architectural style, comprised of many cognitive features from the action selection paradigm.

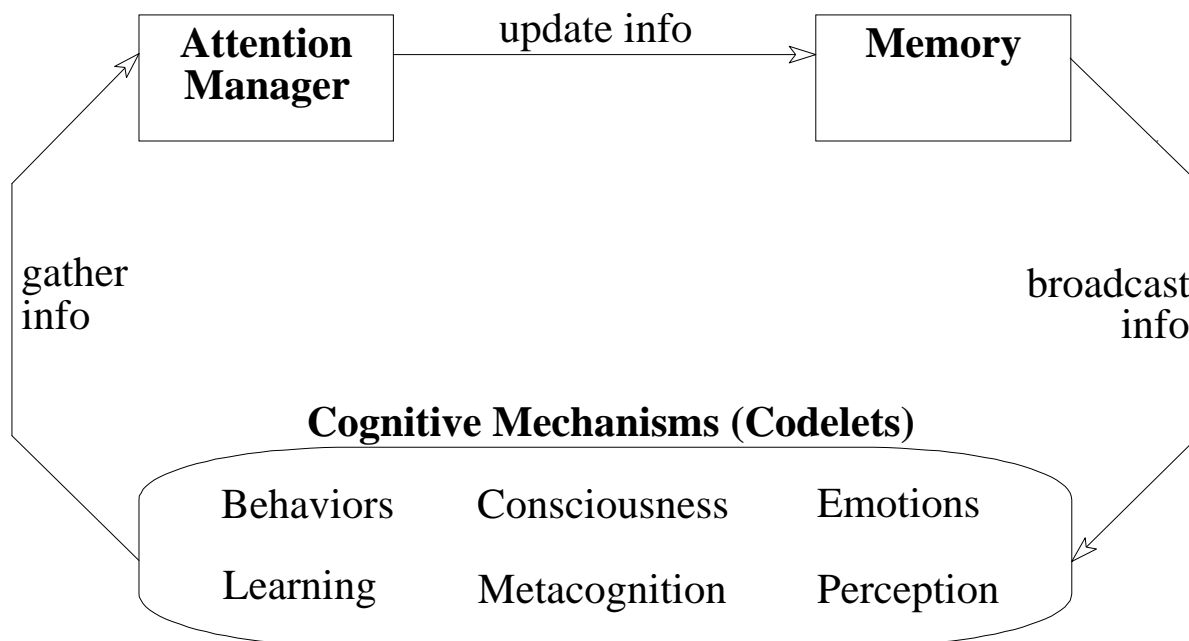


Figure 1: Architectural Style For "Conscious" Software Agents

By extrapolating this architectural style at a lower level, Figure 2 illustrates the particular architecture that is used in ConAg. ConAg is focused primarily on the circled items.

Many of the cognitive mechanisms are in reality driven by small single-task codelets corresponding to global workspace theory's processors and pandemonium theory's demons. Specifically, these codelets comprise emotions (McCauley and Franklin, 1998), behaviors (Song, 1998; Maes 1990), metacognition (Zhang and Franklin, 1998), and perception (Ramamurthy, Bogner, & Franklin, 1998; Bogner, Ramamurthy, & Franklin, 1999; Zhang, Franklin, Olde, Wan, & Graesser, 1998). Emotion codelets are dispersed throughout "conscious" software agents, looking for situations which will influence the systems' overall emotional state. Systems' emotional states are a composite of several emotions, such as happiness, sadness, anger, and fear. Behaviors serve to perform the systems' major actions. For example, for agents which communicate via email, a behavior might be to compose a reply to an email. Drives are built-in to "conscious" software agents, and they operate in parallel. Drives activate behaviors, and behaviors work to fulfill them. Perception varies depending on the domain; it can range from receiving voice in tutoring systems to natural-language email messages in department seminar organizers. The Focus is the location where perceptual information is created for the agents' own use. Here this perceptual information is associated with emotions and memories. "Conscious" software agents contain numerous memories, including associative (Kanerva, 1988), case-based (Kolodner, 1993), short-term memory associated with what has become "conscious," and numerous working memories. Metacognition keeps track of agents' internal conditions. If necessary, it can influence the behaviors, perception, "consciousness," and learning. For example, metacognition can make the agent more goal-oriented or opportunistic, and cause voluntary attention by influencing the chances that a coalition of codelets will make it to "consciousness." Learning takes many forms in these agents such as the ability to learn new behaviors. The primary responsibility of "consciousness" codelets are to bring novel or conflicting information to "consciousness" (Bogner, Ramamurthy, & Franklin, 1999). This

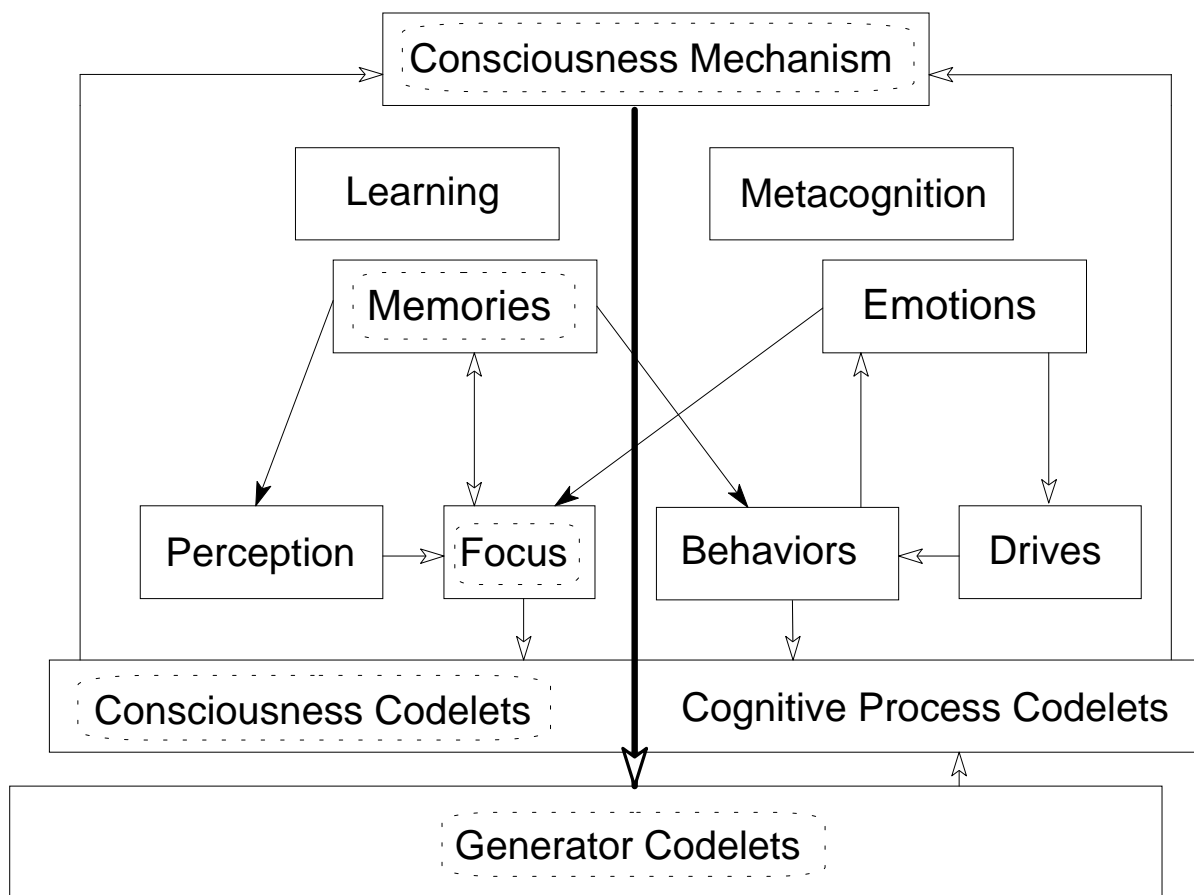


Figure 2: An Architecture For “Conscious” Software Agents

includes new perceptual information. It also includes conflicts between what is perceived and what is remembered, and conflicts in the potential communication output of the agents’.

All codelets have activation levels corresponding to how important they perceive their action to currently be. These activation levels are also directly associated with the higher level concept the codelet serves, such as a behavior currently being executed. Codelets also contain associations with other codelets, corresponding to the links of pandemonium theory’s demons. All codelets which are actively performing their tasks join the playing field, also inspired from pandemonium theory. The playing field is the first portion of “consciousness” “attention mechanism. The attention mechanism contains a way to form coalitions of codelets. Specifically, a coalition manager works to group codelets into coalitions based on their associations to other codelets. A coalition must be selected for “consciousness” from among the formed coalitions. It also contains a spotlight controller that chooses the next coalition for the spotlight of “consciousness” based on coalitions’ activation level. Once the “conscious” coalition has been selected, the attention mechanism’s broadcast manger sends out the coalition’s information. This information is also placed in the module’s short-term memory as it is known that approximately seven recently “conscious” items remain in short-term memory. All codelets in the system are able to receive the broadcast. In some cases, there must be multiple copies of the same kind of codelets based on what is “conscious.” Generator codelets, each corresponding to a specific kind of codelet, solve these cases, by receiving the broadcast and instantiating copies of themselves with the correct information.

“Conscious” software agents are extremely domain-specific entities. Following the action selection paradigm, what an agents perceives, its drives and corresponding behaviors, etc. are coupled to its environment. A relatively domain-independent portion is the “consciousness” mechanism. This is ConAg’s main focus.

5. The “Conscious” Agent Framework

5.1. What Are Frameworks?

Frameworks are often not well understood and are misused outside the object-oriented community (Johnson, 1997). Frameworks are reusable designs of all or part of systems. They are commonly represented by a set of abstract classes and the way these classes’ instances interact. A framework’s purpose is to provide an application skeleton that can be customized by developers. Framework’s are powerful as they can significantly reduce the amount of effort necessary to develop customized applications, thereby saving organizations time and money.

Frameworks are a form of design reuse as they express reusable designs. They are at a lower level than architectural styles as they are more concrete. In fact, frameworks are actual programs, and, therefore, users of frameworks are often tied to a programming language. Because of this, frameworks are more closely tied to their domain than architectural styles. Also, successful frameworks must be consist throughout more so than architectural styles. Since frameworks are programs, they are often easier for programmers to learn and apply than architectural styles. This occurs partially because only a compiler is needed, not special design notation or software tools often utilized when creating architectural styles.

When using frameworks, developers often think they are just using an object-oriented language’s class library. However, frameworks are different than class libraries as frameworks reuse high-level design. With frameworks, there is more to learn before classes can be reused. For example, a set of classes must typically be learned at once, and classes are not reused in isolation. A framework can usually be distinguished from a class library if there are dependencies among components and developers learning the library comment on its complexity. Because of this complexity, frameworks require quality documentation. Even with the difficulty which comes in learning a framework, expert developers normally prefer frameworks over special-purpose languages as they are easier to extend.

5.2. What Are Components?

Components are actual working code portions and are designed for reuse. Ideally, components should be easy to learn. Often, with black-box reuse, developers do not need to learn how components are implemented. Components are simply connected to create a new system. By using existing components, more reliable systems which are created and, therefore, easier to maintain. As components increase in generality, the payoff for use in narrow focused domains diminishes (Biggerstaff & Richter, 1987). On the flipside, with component growth, the payoff

when reusing the component increases more than linearly due to the complexity costs. Larger components, however, often become more specific which increases the costs when modification is required.

5.3. Frameworks and Components

Frameworks are intertwined with components, and they are cooperating technologies. Frameworks make it easier to develop new components. For example, frameworks provide a standard way for components to do data exchange, error handling, and invoke operations on other components. In other words, frameworks allow components to make assumptions about their environment, making component integration easier. Frameworks provide specifications and templates for new components and allow new components to be built out of smaller components. Frameworks can actually be viewed as components in the sense that applications might use several components, and vendors sell them as products. As a whole, frameworks are more customizable than components and have more complex interfaces, again highlighting the difficulty of learning a framework.

5.4. ConAg

As a framework, ConAg serves four primary goals:

1. To fit within the boundaries of the architectural styles for “conscious” software agents.
2. To provide a drop-in implementation for the domain-independent portions of these agents’ “consciousness” mechanism.
3. To provide working, easily customizable, and properly documented domain-specific portions of the “consciousness” mechanism, such as “consciousness” codelets which look for a specific conflict.
4. To provide quality stubs for the cognitive mechanisms such as emotions and behaviors found in “conscious” software agents. These will illustrate how these modules should work with the “consciousness” module and provide a starting point for these portions’ development.

5.4.1. ConAg and Java

ConAg’s source code is 100% pure Java, and utilizes Java’s AWT and Beans frameworks. All possible classes are Java Beans, helping contribute to both black-box reuse and, when necessary, easy modification necessary for white-box reuse. Each source file has detailed header comments, and almost every line of code is commented, with all comments catered for Javadoc use. In other words, explicit care has been taken to ensure ConAg follows good coding practice.

ConAg has a detailed package structure allowing for its components to be easily found and identified. The next several figures and subsections describe these packages at a high level. At the root is the “conscious” software agent package, shown in figure 3. Within this package is ConAg, as well as the other cognitive mechanisms such as emotion and perception. ConAg’s

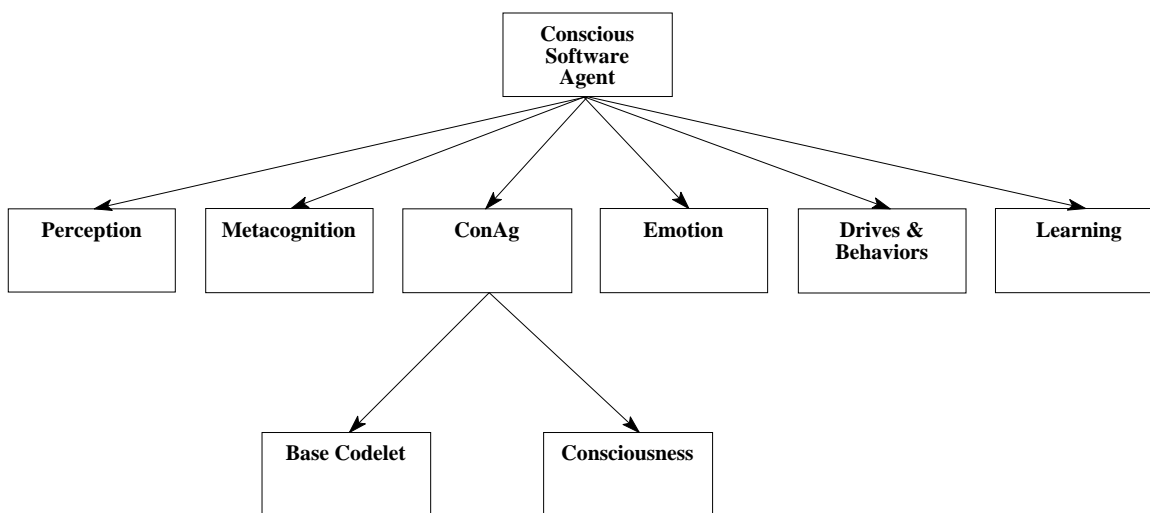


Figure 3: “Conscious” Software Agent Package

package branches off into two main groups, the base codelet and “consciousness” packages. Each will be described below.

5.5. *ConAg’s Domain Independent Portions*

ConAg can be viewed partially as a “generic” framework similar to frameworks for building graphical user interfaces. It implements portions of the “consciousness” mechanism intended to work across domains. In other words, many portions of the framework can be dropped in an agent being developed such as a travel agent or an intelligent tutor.

5.5.1. Codelet Definitions

Within figure 3’s base codelet package, ConAg includes a definition for what a minimum codelet is. All codelets in “conscious” software agents must inherit this base codelet, customizing it for specific use. This is needed as codelets must contain certain properties to be appropriately recognized and handled by the “consciousness” mechanism’s attention portion. This codelet definition includes items such the way codelets’ associations to other codelets are represented. The manner in which codelets’ activation levels are represented and by which they are increased and decreased. The structure each codelet should carry in order for its information to be successfully broadcast. The way codelets receive the “conscious” broadcast and check short-term memory. A method for codelets to join and leave the playing field. And so on.

As seen in figure 4, ConAg also includes a definition for a base “consciousness” codelet. This definition highlights that “consciousness” codelets carry information, whether it be a conflict or novel information. In addition, it contains the method by which “consciousness” codelets rapidly increase their activation levels in their attempt to gain “consciousness.” Further specificity for “consciousness” codelets is included, and discussed in the Domain Dependent Portions section below.

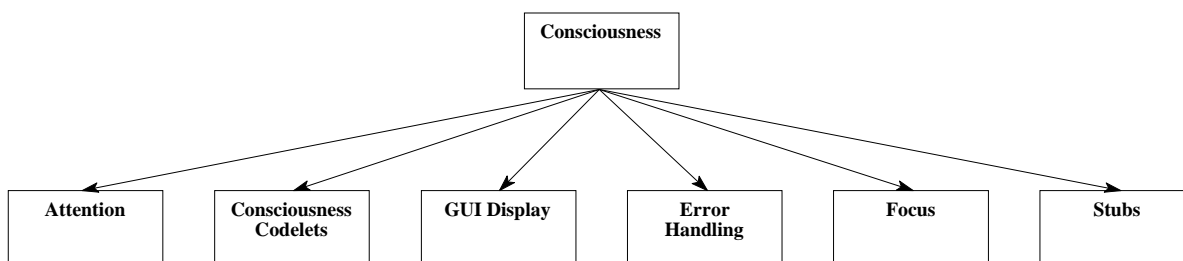


Figure 4: “Consciousness” Package

5.5.2. Attention

Figure 4 also shows where attention is located. The attention package includes a pandemonium-like playing field. It includes a coalition manager for forming coalitions. It has spotlight controller for selecting the “conscious” coalition. It contains a broadcast manager for sending out the broadcast. Based on pandemonium theory’s concept demons, it contains a concept manager for learning new concepts. It also has a short-term memory. While domain independent, each of these can be modified to the developer’s satisfaction. For example, a new algorithm for forming coalitions can be created based on the one included in the framework.

5.5.3. Graphical User Interface

ConAg’s graphical user interface, written using Java’s AWT, serves two roles. First, if desired, it allows the “consciousness” module to be started independently of the other modules. While yet unproven, this may provide a testing ground to compare how these agents run entirely “unconsciously” versus “consciously.” Second, the GUI provides a window into the inner workings of the system. Figure 5 illustrates the GUI package structure. Notice that it is a close mirror to the “consciousness” package shown in figure 4. The differences to the “consciousness” package include a components package, where display components used throughout the display module reside. All of the GUI is domain-independent except for the “consciousness” codelet and focus viewing mechanisms as what is being viewed is domain-dependent. More importantly, ConAg does not depend on a GUI to run; the package could be completely removed. This provides a means for a user-interface based on a different toolkit to be provided. The stubs package provides an insight to what is occurring inside ConAg’s provided stubs such as emotion. However, with the assumption that the actual cognitive modules will utilize the provided stubs and/or be implemented as Java beans, it is straight-forward to apply these viewers to the true modules.

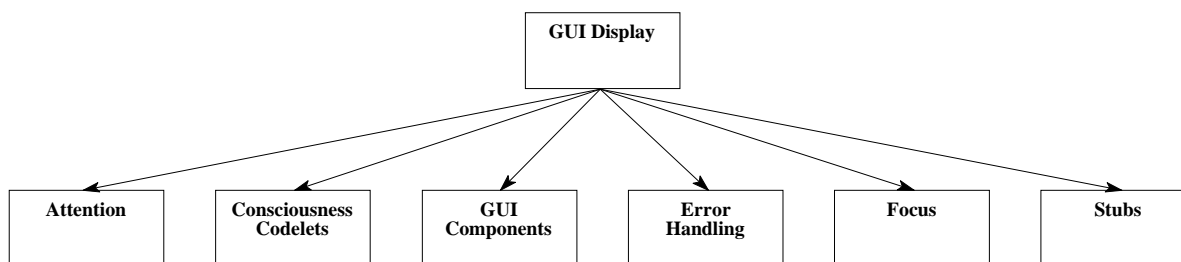


Figure 5: Display Package

5.5.4. Error Handling

As seen in figure 4, ConAg provides a common mechanism for handling errors throughout the framework. The provided GUI contains its own error handling mechanism, using identical techniques as the framework's main one. This technique allows the developer a single point in which to code for handling additional errors while also providing consistent debugging methods throughout the framework.

5.5.5. Other Cognitive Module Stubs

ConAg's stubs for the other cognitive modules such as metacognition do run and provide basic functionality. Each stub is in its own package, as illustrated in Figure 6. Unlike the fully implemented cognitive modules, care has been taken to try and make these modules as domain-independent as possible. ConAg's GUI for the stubs takes the same approach.

5.6. Domain Dependant Portions

ConAg's packages for "consciousness" codelets and the Focus are domain-dependent, the exception begin the base "consciousness" codelet components described in section 5.5.1. As seen in figure 7, ConAg provides "consciousness" codelets to detect novel and conflicting information for both perceptual input and conflict detection for the system's output. "Consciousness" codelets are domain specific. ConAg provides "consciousness" codelets for use when the agent communicates via email. These provide a basis for white-box reuse in order to apply these components to new domains. As "consciousness" codelets are Java beans, often times code changes to the graphical user interface are not needed "consciousness" codelets are applied to new domains.

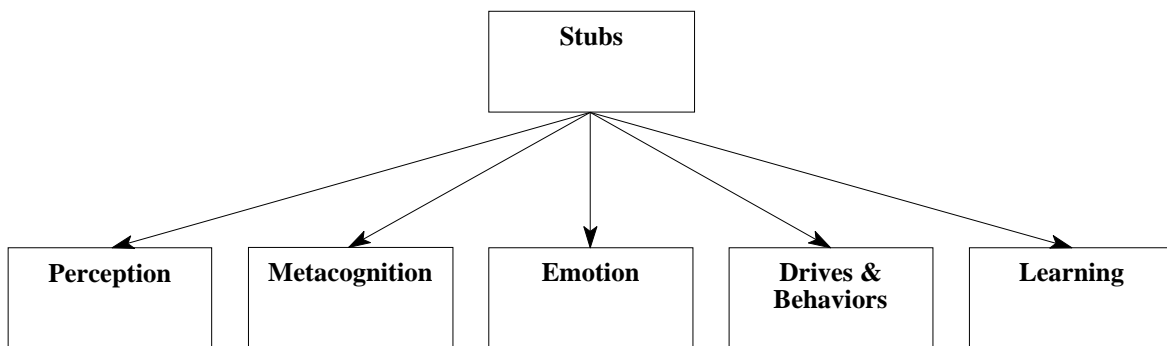


Figure 6: Stubs Package

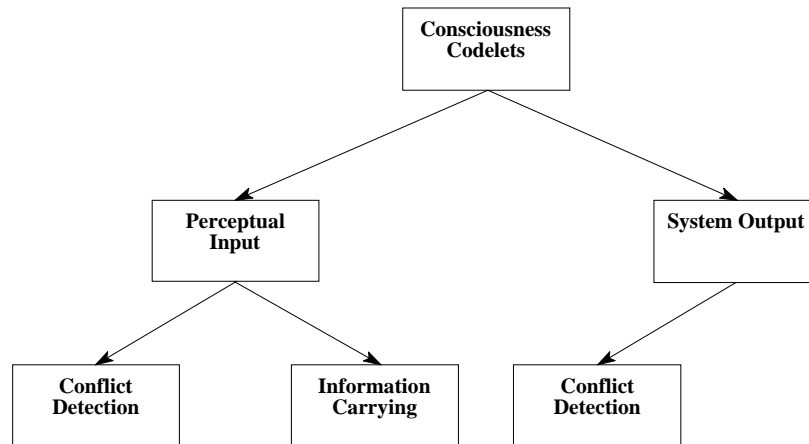


Figure 7: “Consciousness” Codelet Package

As discussed in section 4, the Focus is the location where perceptual information is created for the agents’ own use. This perceptual information is associated with agents’ memories and emotions, and “consciousness” codelets bring this new and potentially conflicting information to “consciousness.” Perceptual information is domain specific as are systems’ memories about their behaviors and what has perceived. Therefore, the Focus, while an integral part “conscious” software agents, is domain-dependent. Even so, ConAg provides common methods for a Focus’ use across domains.

5.7. What Are Design Patterns?

Frameworks are composed of micro-architectural elements called design patterns (Johnson, 1997). Design patterns describe solutions to recurring problems and a context for which the solution works. They include the solution’s costs and benefits. Design patterns represent the common idioms found repeatedly in software designs and makes them codified, explicit, and applicable to similar problems (Monroe, Kompanek, Melton, & Garlan, 1997). Patterns emphasize documentation and literary style rather than code generation or tools (Mellor & Johnson, 1997). Design patterns are useful as a documentation tool for classification of design fragments, making it easier for a development team to add new members (Cline, 1996). Design patterns provide a standard vocabulary for developers. They communicate information between designers, programmers, and maintenance programmers at a significantly higher level than individual classes. They provide a list of items to look for in a design review. Maintenance programmers are less likely to break existing code when they understand and work to preserve the integrity of design patterns during maintenance changes. Patterns are particularly useful for building robust designs in situations where the trade-offs are well understood. When specifying and reusing design patterns, there are three fundamental requirements to be followed (Kompanek, Melton, & Garlan, 1997). First, the design domain must be well understood. Second, the patterns must support the encapsulation of design elements. Finally, the design patterns must be responsible for a collection of well-known and proven design idioms.

5.8. ConAg's Design Patterns

Design patterns are heavily utilized throughout ConAg. Illustrated here is ConAg's use of several patterns, all described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1995). The abstract factory pattern provides an interface for creating families of related or dependent objects without their concrete classes needing specification. Abstract factory patterns are used throughout ConAg. Examples are seen in the base codelet and base "consciousness" codelet definitions.

The singleton pattern ensures that there is only one instance of a class and that it is accessible globally. ConAg relies on this pattern for each of the components that start up its different modules. For example, the attention startup component provides single access to the attention components, namely the playing field and broadcast. "Consciousness" codelet startup provides access to all of the "consciousness" codelets. Focus startup provides a single point of access to the perceived information and the memories and emotions associated with it.

Used throughout ConAg is the facade pattern, defining a higher-level unified interface to a subsystem, making these subsystems easier to use. In ConAg, active codelets join the playing field. The playing field's structure is hidden from them; there is simply a common way to join and exit the field. Completed perceptual information is set in the focus for use by the entire system; the actual process of perceiving is hidden. Codelets receive the broadcast information; hidden from them is how this information is collected and arranged for broadcast.

The strategy pattern defines a family of algorithms, encapsulating each one and making them interchangeable. This allows algorithms to vary without directly affecting those who utilize it. Throughout ConAg's attention package, great care has been taken to follow this pattern. For example, a different algorithm for forming coalitions can be used in the coalition manager; the same holds true for the spotlight controller's choosing a "conscious" coalition. A method for gathering the information to be broadcast and actual manner it is broadcast is also interchangeable. The same holds true for the representation of short-term memory.

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and automatically updated. A prime example of this in ConAg occurs with the "conscious" broadcast, where one broadcast is received by all codelets in the system. In addition, when the Focus receives a new percept, one announcement of this fact is sent out to all of a system's perceptual "consciousness" codelets.

The memento pattern provides a way to capture and externalize an object's internal state, without violating encapsulation, so that the object can be restored to this same state later. "Conscious" software agents often have a self-preservation mechanism. For agents with this mechanism, the base codelet class has the option to utilize Java's serialization techniques. Serialization achieves the memento pattern, and since all codelets inherit the base codelet component, all codelets' states can be captured and restored. This applies to the system's short-term memory as well.

6. Current Progress

ConAg is a reusable framework for implementing "consciousness" in cognitive agents. Its uniqueness comes from both its cognitive modeling goals and its strict adherence to the principles of software reuse. ConAg utilizes architectural styles, design patterns, and

components to create a framework which enhances developers capabilities. Currently, the majority of ConAg is implemented, and testing and refinement is ongoing. In addition, it is currently being utilized to implement several “conscious” software agents. Integration with other cognitive modules is so far proving successful.

Acknowledgments

1. Myles Bogner is supported in part by NSF grant SBR-9720314.
2. Stan Franklin is supported in part by NSF grant SBR-9720314 and by ONR grant N00014-98-1-0332.
3. The “Conscious” Software Research Group currently includes Art Graesser, Uma Ramamurthy, Lee McCauley, Aregahegn Negatu, Zhaohua Zhang, Ashraf Anwar, and Arpad Kelemen.

References

- Baars, Bernard. (1988). *A cognitive theory of consciousness*. New York: Cambridge University Press.
- Baars, Bernard. (1997). *In the theater of consciousness*. New York: Oxford University Press.
- Basili, V., Briand, L., & Melo, W. (1996). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39 (10), 104-116.
- Biggerstaff, T. & Richter, C. (1987). Reusability framework, assessment, and directions. *IEEE Software*, 4 (2), 41-49.
- Bogner, Myles. (1998). *Creating a "conscious" agent*. Memphis: Master's thesis, The University of Memphis.
- Bogner, Myles, Ramamurthy, Uma, & Franklin, Stan. (1999). "Consciousness" and conceptual learning in a socially situated agent. Dautenhahn, Kerstin (Ed.). *Human Cognition and Social Agent Technology*. Amsterdam: John Benjamins Publishing Company.
- Cline, Marshall P. (1996). The pros and cons of adopting and applying design patterns in the real world. *Communications of the ACM*, 39 (10), 47-49.
- Etzkorn, Letha H. & Davis, Carl G. Automatically identifying reusable OO legacy code. *IEEE Computer*, Oct., 66-71.
- Frakes, William B. & Fox, Christopher, J. (1995). Sixteen questions about software reuse. *Communications of the ACM*, 38 (6), 75-87.
- Franklin, Stan. (1995). *Artificial minds*. Cambridge, MA: The MIT Press.
- Franklin, Stan. (1997). Autonomous agents as embodied ai. *Cybernetics and Systems*, 28 (6), 499-517.
- Franklin, Stan, Kelemen, Arpad, and McCauley, Lee. (1998). IDA: A Cognitive Agent Architecture. *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, 2646-2651.
- Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. (1995). *Design Patterns*. Reading, MA: Addison-Wesley.
- Graesser, Arthur, Franklin, Stan, & Wiemer-Hastings, Peter. (1998). Simulating smooth tutorial dialog with pedagogical value. *Proceedings of the American Association for Artificial Intelligence*. Menlo Park, CA: AAAI Press, 163-167.
- Hofstadter, Douglas & Mitchell, Melanie. (1994). The copycat project: A model of mental fluidity and analogy-making. Holyoak, K. & Barden, J. (Eds.). *Advances In Connectionist and Neural Computation Theory*, 2. Norwood, NJ: Ablex.
- Jackson, John. (1987). Idea for a mind. *SIGGART Newsletter*, 101, 23-26.
- Johnson, Ralph. (1997). Frameworks = (components + patterns). *Communications of the ACM*, 40 (10), 39-42.
- Kanerva, Pentti. (1988). *Sparse distributed memory*. Cambridge, MA: The MIT Press.
- Kaspersen, Donna. (1994). For reuse, process and product both count. *IEEE Software* 11 (5), 12.
- Kolodner, Janet. (1993). *Case-based reasoning*. Morgan Kaufmann Publishers.
- Krueger, Charles. (1992). Software reuse. *ACM Computing Surveys*, 24 (2), 131-183.
- Maes, Pattie. (1990). *How to do the right thing*. Connection Science.

- McCauley, Thomas L. & Franklin, Stan. (1998). An architecture for emotion. AAAI Fall Symposium "Emotional and Intelligent: The Tangled Knot of Cognition."
- Mellor, Stephen & Johnson, Ralph. (1997). Why explore object methods, patterns, and architectures. IEEE Software, Jan., 27-30.
- Monroe, Robert T., Kompanek, Andrew, Melton, Ralph, & Garlan, David. (1997). Architectural styles, design patterns, and objects. IEEE Software. Jan., 43-52.
- Prieto-Días, Rubén, & Freeman, Peter. (1987). Classifying software for reusability. IEEE Software, Jan., 6-16.
- Ramamurthy, Uma, Bogner, Myles, & Franklin, Stan. (1998). "Conscious" learning in an adaptive software agent. Proceedings of The Second Asia Pacific Conference on Simulated Evolution and Learning (SEAL 98). Canberra, Australia.
- Zhang, Zhaohua, Franklin, Stan, & Dasgupta, Dipankar. (1998). Metacognition in software agents using classifier systems. Proceedings of AAAI 98, 82-88.
- Zhang, Zhaohua, Franklin, Stan, Olde, Brent, Want, Yun, & Graesser, Arthur. (1998). Natural language sensing for autonomous agents. Proceedings of the IEEE Joint Symposia on Intelligence and Systems. Rockville, Maryland, 374-81.