Non-Software Examples of Software Design Patterns

by
[Michael Duell](#)

Abstract

Software design patterns have roots in the architectural patterns of Christopher Alexander, and in the object movement. According to Alexander, patterns repeat themselves, since they are a generic solution to a given system of forces. The object movement looks to the real world for insights into modeling software relationships. With these dual roots, it seems reasonable that software design patterns should be repeated in real world objects. This paper presents a real world, non software instance of each design pattern from the book, *Design Patterns - Elements of Reusable Object-Oriented Software* [13]. The paper also discusses the implications of non-software examples on the communicative power of a pattern language, and on design pattern training.

1. Introduction

Within the software industry, a growing community of patterns proponents exists. The roots of the patterns movement are found in the writings of architect Christopher Alexander, who describes a pattern as a generic solution to a given system of forces in the world [1]. Alexander's patterns can be observed in everyday structures. Each pattern in *A Pattern Language*[2] includes a picture of an archetypal example of the pattern.

Since objects were the predominate world view at the time that patterns were embraced by the software world, patterns also have roots in the object movement [9]. Unfortunately examples of software design patterns are not as abundant as Alexandrian patterns, since they represent elegant designs, rather than the designs that people generate initially [13]. Access to elegant designs is often limited due to the proprietary nature of much of the software being developed today.

According to Alexander, real world *patterns always repeat themselves, because under a given set of circumstances, there are always certain fields of relationships which are most nearly well adapted to the forces which exist* [1]. In software, real world problems are either modeled entirely, or real world objects are transformed into hardware and software to produce real world results [5]. Since software design patterns have roots in both Alexandrian patterns, and in the object movement, it seems logical that software design patterns can be found in real world objects. This is not to say that software design patterns are necessarily models of the real world objects, but the relationships between objects that have been adapted to deal with certain forces can be observed both in the "real world" and in software objects. To test this hypothesis, a real world example was sought for each of the 23 Gang of Four Patterns [13]. The examples follow in sections 2 through 4.

2. Creational Patterns

Five creational patterns have been documented by the Gang of Four. Examples of these creational patterns can be found in manufacturing, fast food, biology and political institutions.

2.1 *Abstract Factory* Example

The purpose of the *Abstract Factory* is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the

manufacture of Japanese automobiles. The stamping equipment is an *Abstract Factory* which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes [16].
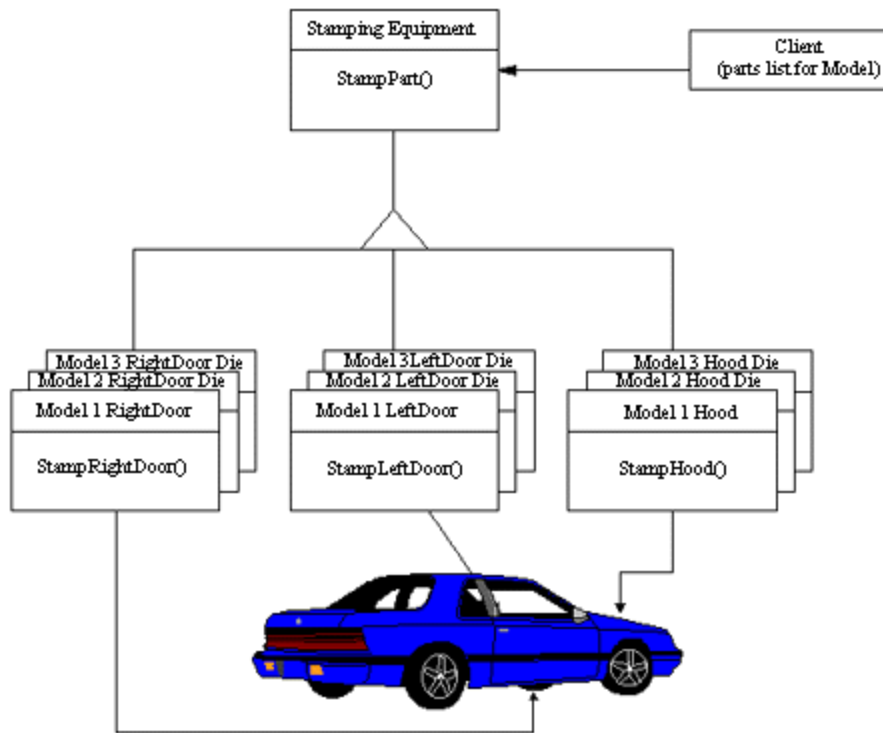

Figure 1: Stamping Example of the *Abstract Factory*

2.2 *Builder* Example

The *Builder* pattern separates the construction of a complex object from its representation, so that the same construction process can create different representation. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, coke, and toy car). Note that there can be variation in the contents of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.
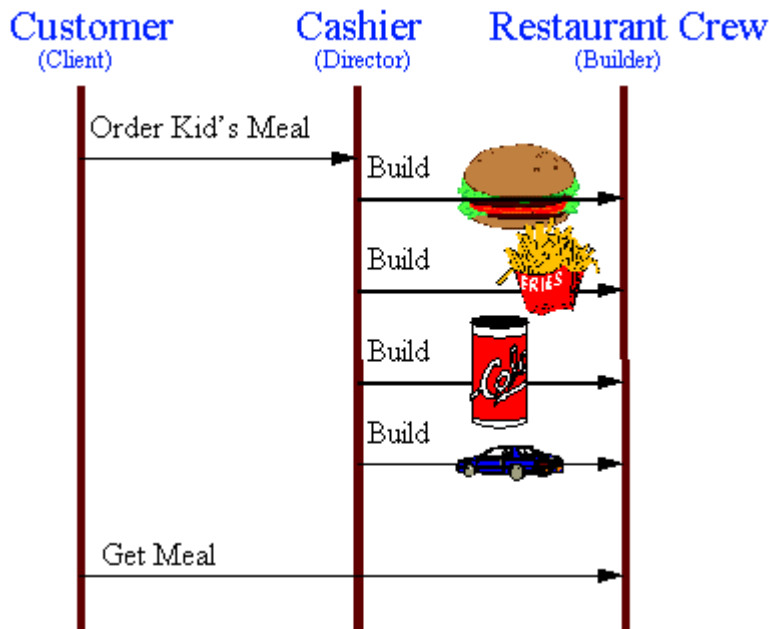
Figure 2: Object Interaction Diagram for the *Builder* using Kid's Meal Example

2.3 *Factory Method* Example

The *Factory Method* defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes [15]. The class of toy (car, action figure, etc.) is determined by the mold.
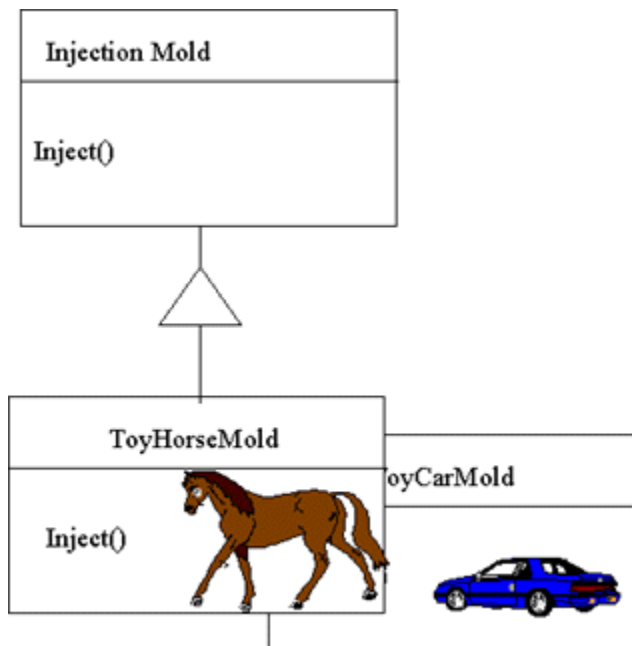

Figure 3: Object Diagram for *Factory Method* using Injection Mold Example

2.4 *Prototype* Example

The *Prototype* pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive, and does not participate in copying itself. The mitotic division of a cell, resulting in two identical cells, is an example of a prototype that plays an active role in copying itself and thus, demonstrates the *Prototype* pattern. When a cell

splits, two cells of identical genotype result. In other words, the cell clones itself.
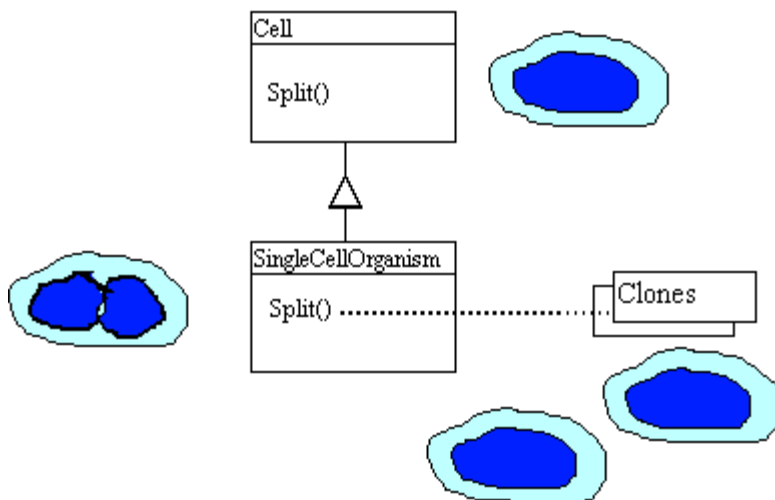

Figure 4: Object Diagram for *Prototype*using Cell Division Example

2.5 *Singleton* Example

The *Singleton* pattern ensures that a class has only one instance, and provides a global point of access to that instance. The *Singleton* pattern is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a *Singleton*. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.
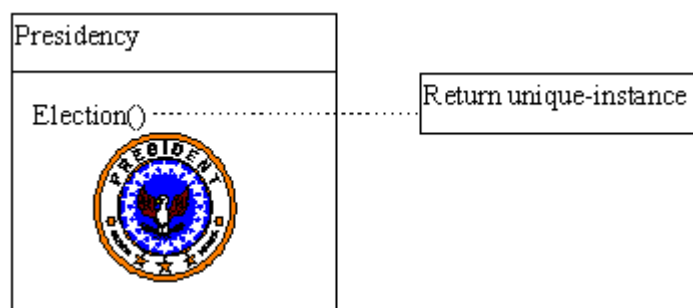

Figure 5: Object Diagram for *Singleton* using Presidency Example

3. Structural Patterns

Seven structural patterns have been documented by the Gang of Four. Examples of these patterns can be found in hand tools, residential wiring, mathematics, holiday tradition, catalog retail, and banking.

3.1 *Adapter* Example

The *Adapter* pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the *Adapter*. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.
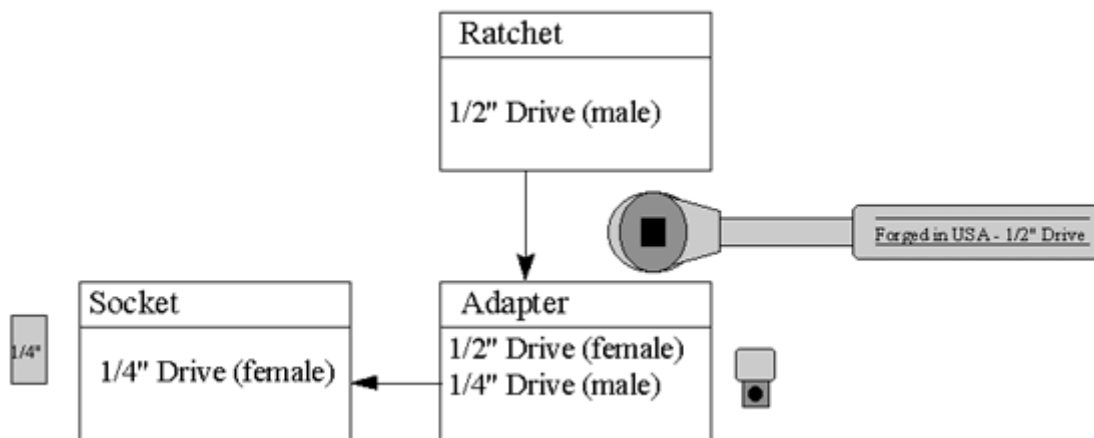
Figure 6: Object Diagram for *Adapter* using Socket Adapter Example

3.2 *Bridge* Example

The *Bridge* pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the *Bridge.* The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches.
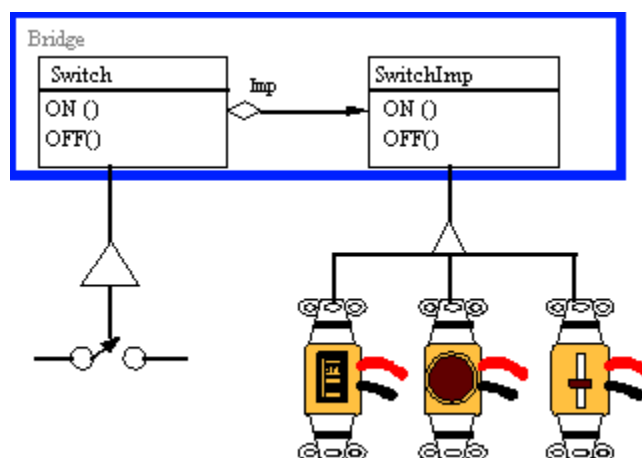

Figure 7: Object Diagram for *Bridge* using Electrical Switch Example

3.3 *Composite* Example

The *Composite* composes objects into tree structures, and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are *Composites.* An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.
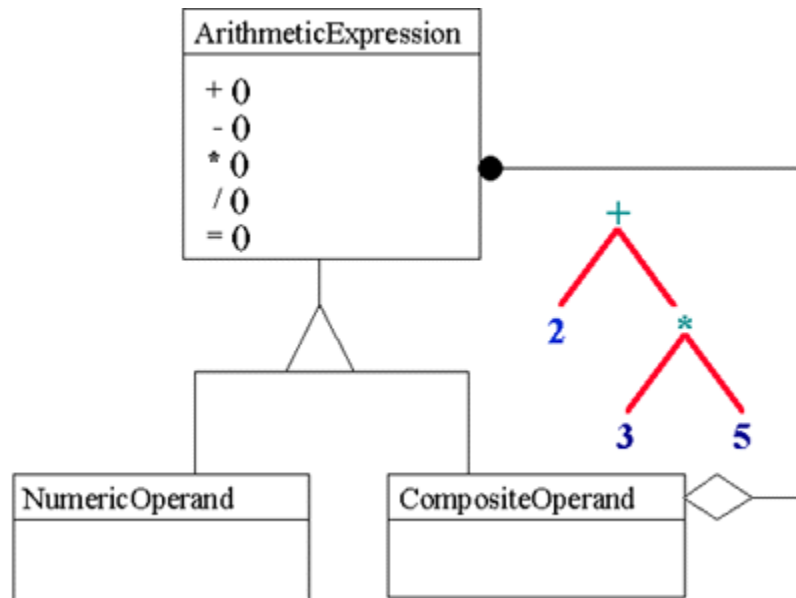
Figure 8: Object Diagram for *Composite* using Arithmetic Expression Example

3.4 *Decorator* Example

The *Decorator* attaches additional responsibilities to an object dynamically. Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.
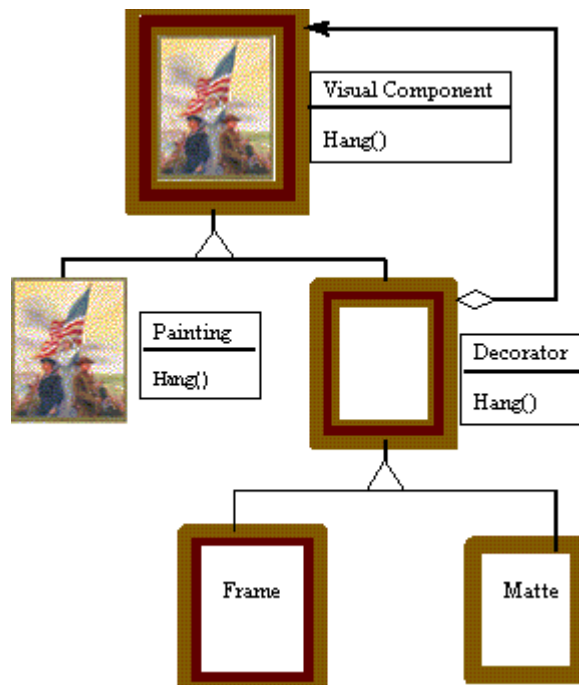

Figure 9: Object Diagram for *Decorator* using Framed Painting Example

3.5 *Facade* Example

The *Facade* defines a unified, higher level interface to a subsystem, that makes it easier to use. Consumers encounter a *Facade* when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a *Facade,* providing an interface to the order fulfillment department, the billing department, and the shipping department.
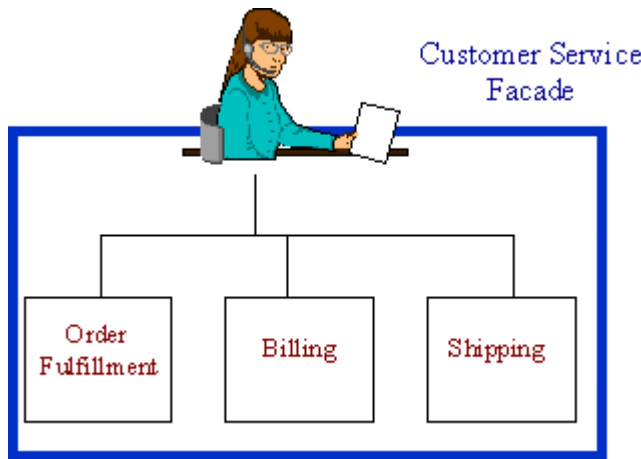
Figure 10: Object Diagram for *Facade* using Phone Order Example

3.6 *Flyweight* Example

The *Flyweight* uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a *Flyweight.* There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the hand set to make a call. All that matters to subscribers is that dial tone is provided, digits are received, and the call is completed.
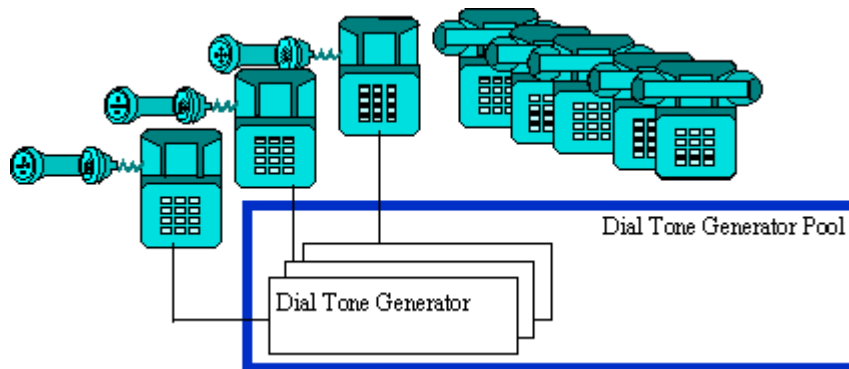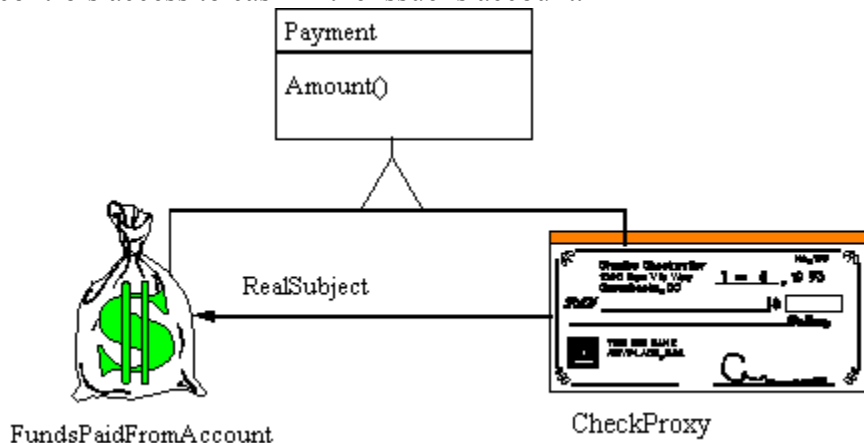

Figure 11: Dial Tone Generator Example of *Flyweight*

3.7 *Proxy* Example

The *Proxy* provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.

Figure 12: Object Diagram for *Proxy* using Bank Draft Example

4. Behavioral Patterns

Eleven behavioral patterns have been documented by the Gang of Four. Examples of these patterns can be found in coin sorting banks, restaurant orders, music, transportation, auto repair, vending machines, and home construction.

4.1 *Chain of Responsibility* Example

The *Chain of Responsibility* pattern avoids coupling the sender of a request to the receiver, by giving more than one object a chance to handle the request. Mechanical coin sorting banks use the *Chain of Responsibility*. Rather than having a separate slot for each coin denomination coupled with receptacle for the denomination, a single slot is used. When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank.
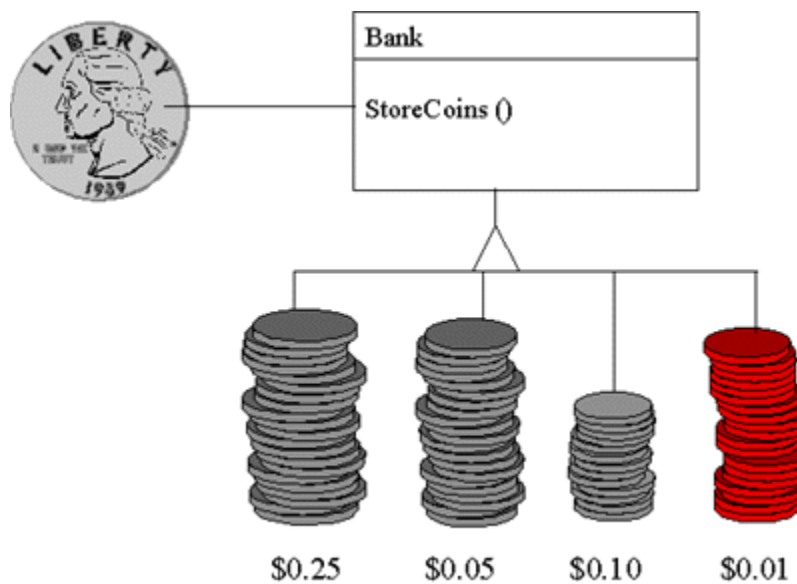

Figure 13: Object Diagram for *Chain of Responsibility* using Coin Sorting Example

4.2 *Command* Example

The *Command* pattern allows requests to be encapsulated as objects, thereby allowing clients to be paramaterized with different requests. The "check" at a diner is an example of a *Command* pattern. The waiter or waitress takes an order, or command from a customer, and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by different diners is not dependent on the menu, and therefore they can support commands to cook many different items.
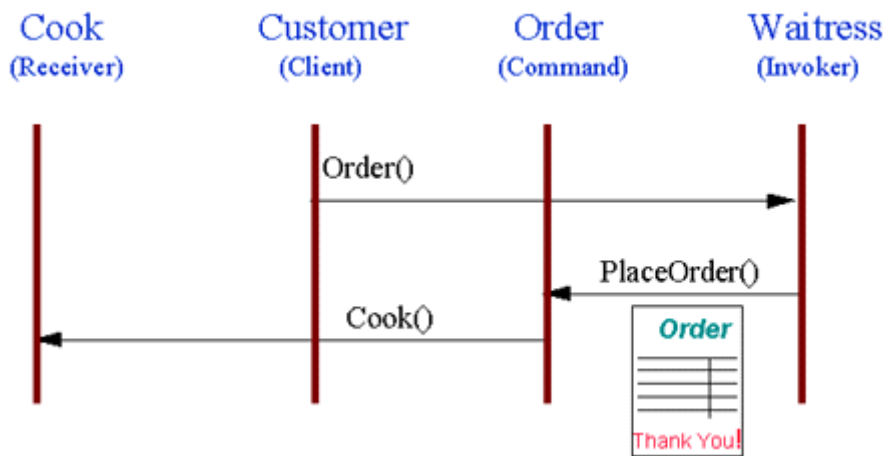
Figure 14: Object Interaction Diagram for *Command* using Diner Example

4.3 *Interpreter* Example

The *Interpreter* pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of *Interpreters.* The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music [14]. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.
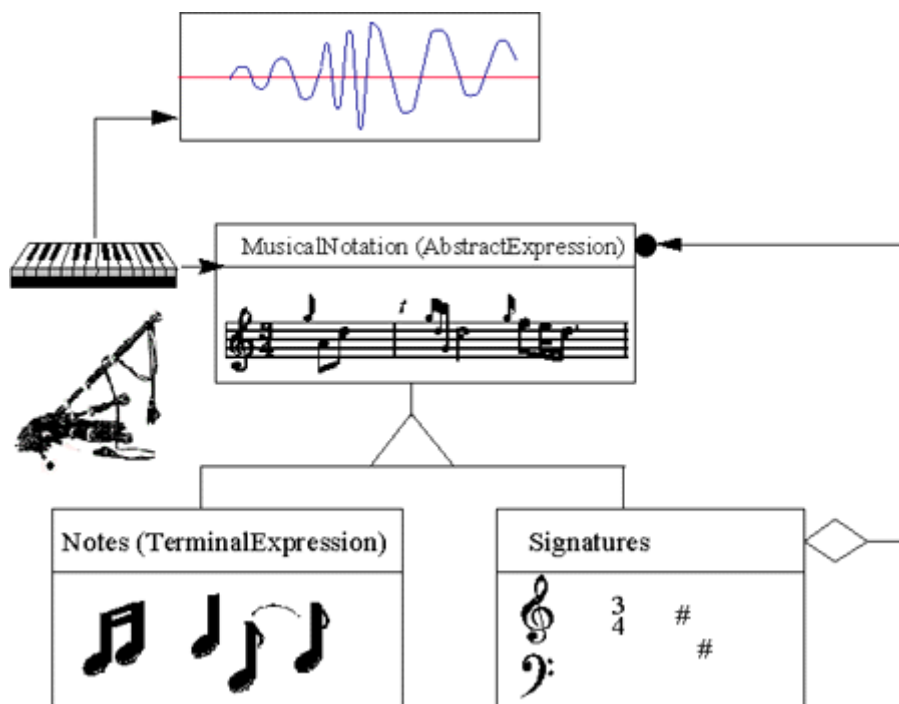

Figure 15: Object Diagram for *Interpreter* using Music Example

4.4 *Iterator* Example

The *Iterator* provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next
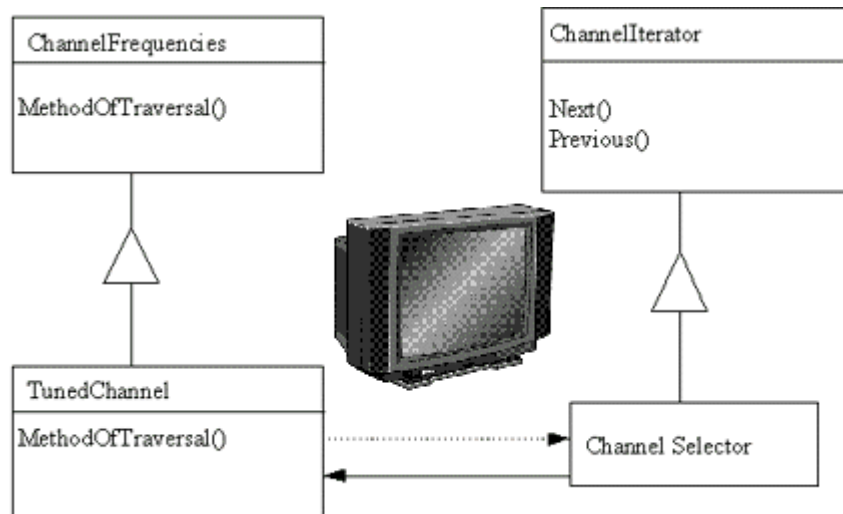
channel, without knowing its number.

Figure 16: Object Diagram for *Iterator* using Channel Selector Example

4.5 *Mediator* Example

The *Mediator* defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the *Mediator*, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.
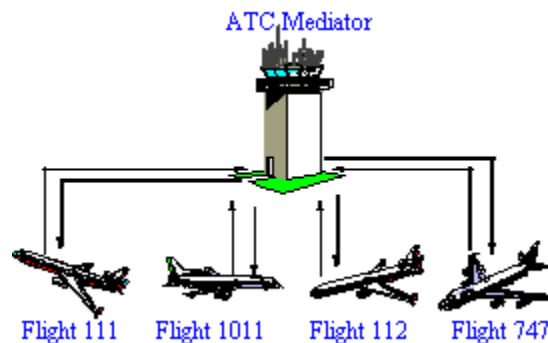

Figure 17: ATC Tower Example of *Mediator*

4.6 *Memento* Example

The *Memento* captures and externalizes an object's internal state, so the object can be restored to that state later. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled, and the other side serves as a *Memento* of how the brake parts fit together [8]. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the *Memento*.
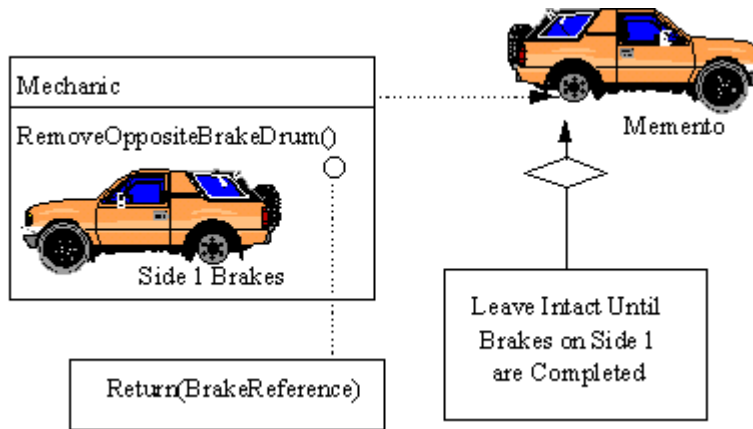
Figure 18: Object Diagram for *Memento* using Brake Example

4.7 *Observer* Example

The *Observer* defines a one to many relationship, so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid.
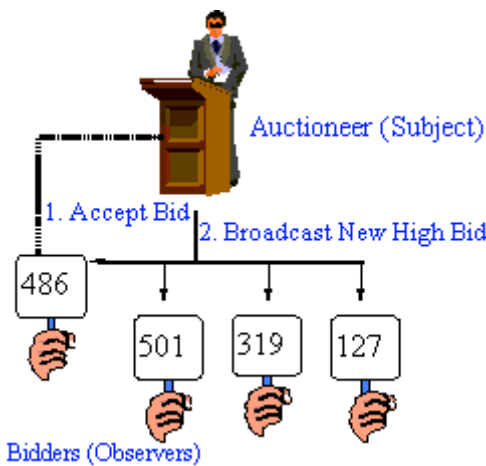

Figure 19: Auction Example of *Observer*

4.8 *State* Example

The *State* pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.
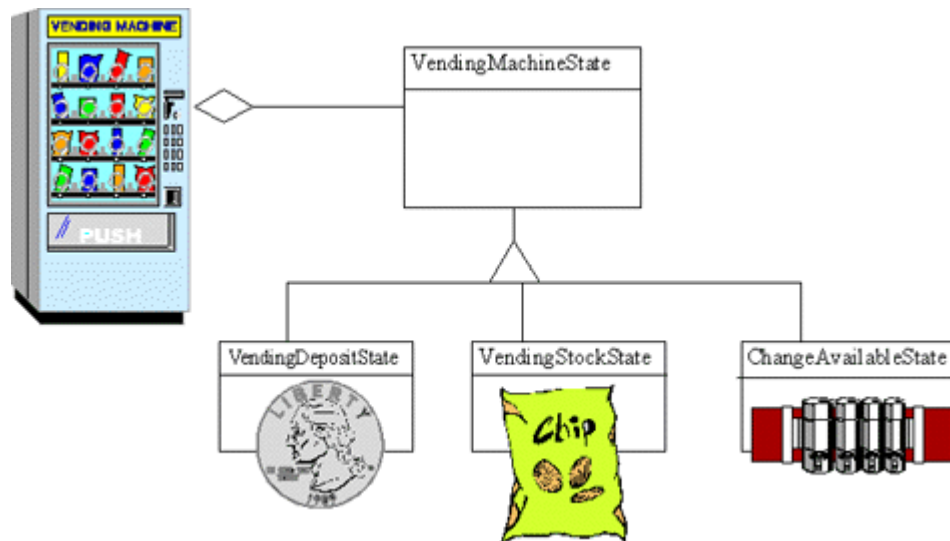
Figure 20: Object Diagram for *State* using Vending Machine Example

4.9 *Strategy* Example

A *Strategy* defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a *Strategy.* Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the *Strategy* based on tradeoffs between cost, convenience, and time.
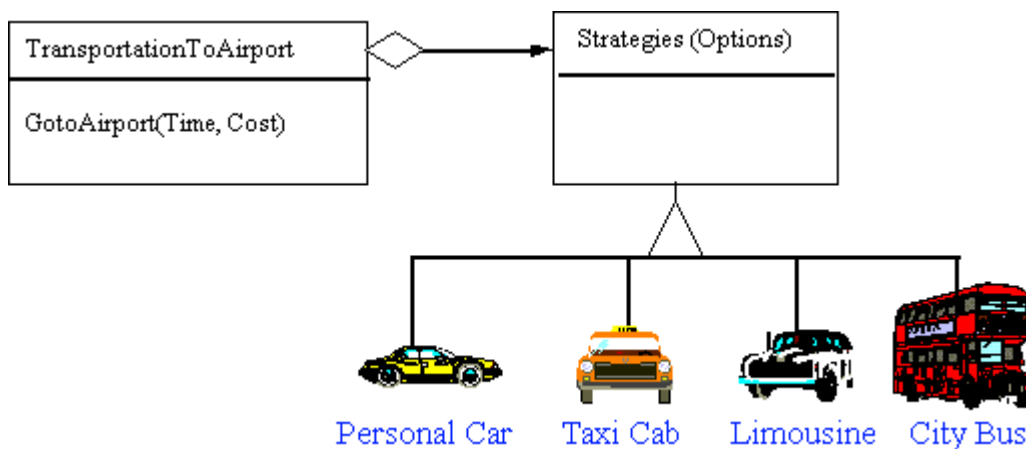

Figure 21: Object Diagram for *Strategy* using Airport Transportation Example

4.10 *Template Method* Example

The *Template Method* defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the *Template Method* when developing a new subdivision. A typical subdivision consists of a limited number of floor plans, with different variations available for each floor plan. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the latter stages of construction to produce a wider variety of models.
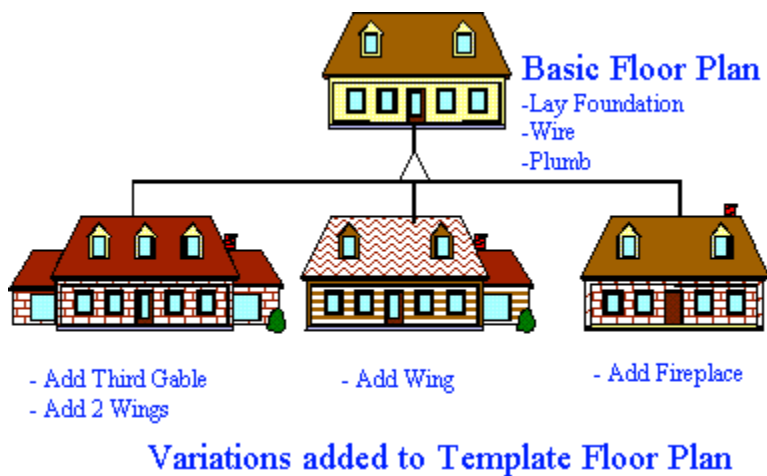
Figure 22: Basic Floor Plan Example of *Template Method*

4.11 *Visitor* Example

The *Visitor* pattern represents an operation to be performed on the elements of an object structure, without changing the classes on which it operates. This pattern can be observed in the operation of a taxi company. When a person calls a taxi company he or she becomes part of the company's list of customers. The company then dispatches a cab to the customer (accepting a visitor). Upon entering the taxi, or *Visitor*, the customer is no longer in control of his or her own transportation, the taxi (driver) is.
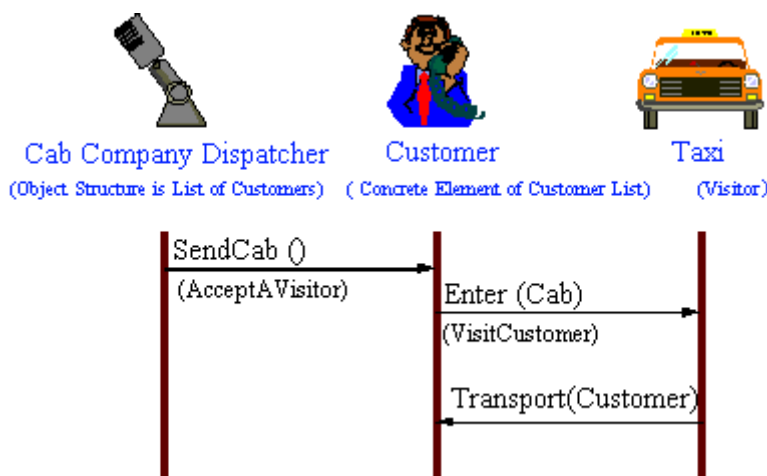
Figure 23: Object Interaction Diagram for *Visitor* using Taxi Cab Example

5. Implications

Non-software examples of each of the software design patterns cataloged by the Gang of Four have been shown to exist. One might now wonder about the practical implications of these examples. These non-software examples are useful in increasing the communicative power of the pattern language and as aids to learning the patterns.

5.1 Increasing the Communicative Power of the Pattern Language

Alexander had hoped that true patterns would enter a common language that all could share [2]. Within the software design community, patterns are seen as a way to develop a set of languages to streamline communication between colleagues [4,17]. Patterns are expected to provide a vocabulary for discussing structures larger than modules, procedures, or objects [10].

One crucial element of a language is the mental imagery associated with symbols of the language. In a

language, a given configuration of symbols has meaning only if one can grasp its content, which involves mentally representing it [7]. The importance of mental imagery to pattern languages was not lost on Alexander, who stated that a language was not morphologically complete until the types of buildings that it generates could be visualized concretely [1]. In software design, Richle and Züllighoven recognized the importance of concrete examples in guiding our perception of the application domain [18].

If software design patterns are to become a common language among programmers, shared meaning is essential. If design decisions are communicated, but not understood, designers are forced to make missing assumptions to complete the job [19]. Commonplace examples facilitate understanding, because in order to understand anything, people must find the closest item in memory to which it relates [20]. The projects at AG Communication Systems that make extensive use of patterns often use non-software examples to illustrate the relationships at work in the patterns. The examples help provide a common understanding between designers. By establishing common understanding early in the design process, communication between designers is facilitated throughout the project life cycle.

5.2 Non-Software Examples as Aids to Learning Patterns

Students require examples whenever new concepts are presented. This was evident in the evaluation of a course in patterns offered at AG Communication Systems and has also been documented by others [12]. When learning something new, the student naturally tends to exploit prior knowledge in an effort to understand the new concepts [6]. For this reason, many examples should be included when students are first exposed to software design patterns [12]. Specific examples should be ones with which the student is acquainted, but not expert [3]. Providing acquaintance examples does not increase the new material that must be learned. At the same time, choosing an example outside of a student's expertise keeps the student from getting so involved in the example that the point of the new material is missed. Since patterns must ultimately reside in one's own mind [11], using examples that are common to a large cross section of people, training material can build on examples already committed to memory.

6. Conclusion

The repetition of software design patterns in non-software examples is evidence that patterns are not limited to a specific domain. Instances of these patterns in everyday objects can benefit software designers, even though the examples are not expressed in a programming language. The examples presented in this paper are intended to be ones that are familiar to a large cross section of people (although some may be culturally biased towards North Americans). By drawing on common experience, such examples facilitate understanding of specific design patterns, and thus improve communication and serve as an aid to learning the patterns.

7. Acknowledgments

The Author gratefully acknowledges Brandon Goldfedder, of the Dalmatian Group, Linda Rising, of AG Communication Systems, Alistair Cockburn, of Humans and Technology, and Ralph Johnson of the University of Illinois, Urbana for their useful comments on this paper.

8. References

1. Alexander, C. *The Timeless Way of Building.* Oxford University Press, 1979.
2. Alexander, C., et al. *A Pattern Language.* Oxford University Press, 1977.
3. Anthony, D. "Patterns for Classroom Education", in *Pattern Languages of Program Design II*, Addison-Wesley, 1996.
4. Berczuk, S. "Finding solutions through pattern languages", *Computer,* Vol. 27, No. 12. December, 1994.

5. Booch, G. "Object Oriented Design" in *Tutorial on Software Design Techniques,* pp. 420-437, IEEE Computer Society, 1984.
6. Carroll, J. *"The Nurnberg funnel: Designing minimalist instruction for practical computer skill",* MIT Press, 1990.
7. Chierchia, G. and McConnel-Ginnet, S. *Meaning and Grammar: An Introduction to Semantics.* MIT Press, 1990.
8. Chilton, *Chilton's Auto Repair Manual,* Chilton Book Company, 1985.
9. Coplien, J. "Broadening beyond objects to patterns and to other paradigms", *Position statement for the ACM Workshop on Strategic Directions in Computing Research*, MIT, June 14-15, 1996.
10. Coplien, J. "Idioms, Patterns, and Other Architectural Literature", *IEEE Software*, November, 1996.
11. Cunningham W., Johnson, R., Introduction to *Pattern Languages of Program Design,* Addison-Wesley, 1995.
12. DeBruler, D. "A Generative Pattern Language for Distributed Processing", in *Pattern Languages of Program Design*, Addison-Wesley, 1995.
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
14. Leonhardt, C. *Discovering Music Together 7,* California State Department of Education, 1967.
15. President and Fellows of Harvard College, *Good Time Toy Company*, Publishing Division, Harvard Business School, 1986.
16. Hill, C.W.L. "Toyota: The Evolution of Toyota's Production System" in *Cases in Strategic Management,* Houghton Mifflin, 1993.
17. OOPSLA '95 "Patterns: Cult to Culture?", Panel Discussion in the *Addendum to the Proceedings*. ACM Press, 1996.
18. Richle, D, Züllighoven, H. "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor", in *Pattern Languages of Program Design*, Addison-Wesley, 1995.
19. Ross, D., and Schoman Jr., K. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, Vol. SE3, No 1., January, 1977.
20. Schank, R. *"Tell Me a Story: A New Look at Real and Artificial Memory",* Charles Scribner's Sons, 1990.

1-888-888-AGCS
www.agcs.com

© 1998 AG Communication Systems