ABSTRACT

Title of Document:	Behavioral Designs Patterns and Agile Software Development
	John McGahagan IV, Master, 2013
Directed By:	Associate Professor Mark Austin Institute for Systems Research and the Department of Civil and Environmental Engineering

This paper will explore two areas, agile software development and behavioral design patterns. It will explain the benefits of behavioral design patterns and provide rationale as to why agile teams should incorporate behavioral design patterns into their systems architecture.

Behavioral Designs Patterns and Agile Software Development

By

John McGahagan IV

Scholarly Paper submitted to the Faculty of the Graduate School of the University of Maryland, College Park, in partial fulfillment of the Requirements for the degree of
Master of Science in Systems Engineering 2013

Advisory Committee:
Associate Professor Mark Austin, Chair

Table of Contents

1. Motivation

My experience with agile software development began in January 2011 as a software engineer at a Fortune 500 consulting firm. The project to which I was assigned began embracing an agile mindset which included a change in our development processes. We began working in two week iterations, estimating using planning poker, and adopting several other agile practices. Since we were no longer defining all aspects of our design and system behavior upfront, the need to incorporate flexible design into our system emerged. I immediately began to see the potential that agile holds for software development teams, but also the importance of design that can change behavior.

Through my experiences I have discovered that there are a few agile methodologies and frameworks that are widely embraced. As software engineer, it occurred to me that there may be certain designs that can be agile as well, that is, can accommodate changes in behavior as design emerges. Behavioral software design patterns fit this category. They are often used in design to address common issues in software engineering and behaviors needed in software systems. This paper will highlight and reinforce the need for these behavioral design patterns especially for agile teams whose software systems are evolving in design and behavior. I assert that an understanding of behavioral design patterns will greatly aid agile software development teams. This is what ultimately led to the creation of this paper.

2. Introduction

Changing requirements for systems and new technologies emerging every year have prompted the need to create software that is agile. In the past 25 years, two toolsets have been developed to quicken and improve software development; software design patterns and agile practices. The latter is a set of practices and frameworks that aim to make software development better by allowing teams to adapt to change more easily. “Better” in this sense refers to the elimination of wasted effort, the building of higher quality software, and the ability to meet deadlines which are more likely to slip in waterfall type development environments. Both of these toolsets have the primary goal of making software development better and producing more robust software in a quicker fashion. Agile methodologies such as Scrum and XP exist to accommodate change on teams, but are there any software design patterns that are more adept to changes in the behavior of a software system? The answer is yes and they are called behavioral design patterns. Before we delve into the behavioral design patterns, let’s look at the history of agile and what that term really means.

3. History of Agile Software Development

In 2001, seventeen individuals met to discuss better ways of developing software. From this meeting, the *Agile Manifesto* was born.

“Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan”

The purpose of these four statements is to aid in the building of high quality software around high performing teams and doing so as quickly as possible. From the *Agile Manifesto* came several different agile frameworks and methods such as Scrum, Extreme Programming also known as XP, and Feature Driven Development, just to name a few. These methodologies, when followed with an agile mindset, facilitate the development of software that meets the business value of the end user.

Although *agile* commonly refers to a way of developing software, there is an importance on the ability to create software that a team and or end user considers agile. Responding to change is one of the four statements in the *Agile Manifesto*, thus it is important that agile teams ensure that the software they build has an architecture that is flexible and can change behavior and design. This paper will examine the commonly used behavioral design patterns and reinforce why agile teams should incorporate these patterns in their design.

4. Scope and Objectives

This paper begins with a review of some of the referenced sources in this paper and the sources consulted during the development of this paper. Following, is a summary of abstract classes and interfaces which play a large role in the design patterns discussed in this paper. From there, this paper will switch from the discussion of abstract classes and interfaces to a description of behavioral design patterns and rationale behind their use in agile development teams. A conclusion and mention of future work to be done in this field completes this paper.

5. Sources Review

My goal in performing the research that served as the basis for this paper was to become more acquainted with behavioral design patterns and agile software development. Behavioral design patterns provide solutions to common software engineering design issues. I believe that a more in-depth understanding of behavioral design patterns will aid developers and engineers on agile teams since they will better understand the direct benefits of incorporating behavioral design patterns into their design and because of this understanding will be encouraged to do so. Several sources were consulted to gain the knowledge required to do this. In particular, the books in my research consisted of literature relating to Agile Software Development, Lean Concepts, Data Structures and Design Patterns.

Of all of the sources referenced and consulted, Mike Cohn's *Succeeding with Agile: Software Development Using Scrum*, *User Stories Applied*, and *Agile Estimating and Planning* provided heavy influence on becoming familiar with agile software development. These sources provided an insight into the current ways of performing agile development and helped me gain an understanding of what it means to be agile. This understanding of agile helped form the basis of qualifying design patterns as agile.

- *Agile Estimating and Planning.* by Mike Cohn
- *Succeeding with Agile, Software Development Using Scrum.* by Mike Cohn
- *User Stories Applied.* by Mike Cohn

Although each book covers a slightly different aspect of agile software development, a common theme is the fourth tenant of the agile manifesto "Responding to Change Over Following a Plan." One could speculate that this is because Mike Cohn authored all three of these books, but instead of just showing a similar writing style, I believe Mr. Cohn is staying true to the agile mindset which drives all of the processes and practices. Cohn uses each book to explain practices, tools, and methods that allow teams to respond to change quickly and effectively.

Each of these books is complementary; however each can stand alone as a valuable resource on agile software development. *User Stories Applied* covers writing and accepting user stories, how the team should operate around user stories, and the product backlog of user stories.

Agile Estimating and Planning provides insight into estimating the size of user stories and tasks on agile teams and also covers planning for releases and iterations. *Succeeding with Agile: Software Development Using Scrum* discusses Scrum, the most popular agile framework, and how it can be applied to teams and organizations. Scrum encompasses many, if not all of the practices discussed in the other two Mike Cohn books. *User Stories* and *Agile Estimating* are more in-depth books that cover implementing the practices while *Scrum* covers the general idea of Scrum, the roles in Scrum, and other topics essential to a high level understanding of Scrum.

All three of these books speak from Mr. Cohn's the perspective and experience. The practices and techniques discussed have been implemented in several organizations and development teams (Cohn, 2010). Thus, they have been shown to work. Having experience on the management side of software development, I can see value in the practices. This gives the books not only credibility, but it allows the reader to relate and see the value in the techniques. The situations discussed, such as schedule overrun (Moløkken-Østvold and Jørgensen, 2005), are very real and are common in development teams and organizations. Although I trust Mr. Cohn's experience and have seen his techniques work in development teams, Mr. Cohn sticks to the prescribed implementation of various practices. In other words, I did not find any instances in his works where he suggests modifying a particular agile practice or framework, except regarding distributed teams where face to face communication is not always possible. From my experience I have learned that "If you believe there is one way to do agile, you are not agile." My best guess would be that Mr. Cohn has been in situations where some of the practices needed to be tweaked, however it would be beneficial to the readers of his books to provide an example of this.

Cohn's *User Stories Applied*, *Agile Planning and Estimating*, and *Succeeding with Agile: Software Development Using Scrum*, embrace the agile mindset and manifesto through implementation, suggested practices, and frameworks. Each book provides a better understanding of how to think with agility, what it means to be agile, and what sort of characteristics are important to agile development teams.

6. The Agile Approach to Design

A common misconception when individuals think of agile is that agile teams do not design. This misconception comes from the fact that agile teams do not follow the typical waterfall development cycle shown below in Figure 1. That is, agile teams do not define all of the requirements and design upfront. Agile teams do design, however the question that agile teams are concerned with is "when" the team should design instead of "if" the team should design. Agile teams follow a more iterative approach to design and system delivery by designing whenever changes need to be made as a result of changes in priorities or refactoring. As a result, design in agile teams is often referred to as "emergent." Emergent design in software incorporates three disciplines; using the thought process of design patterns to create application architectures that are resilient and flexible, limiting the implementation of design patterns to only those features that are current, and writing automated acceptance and unit tests before writing code to improve the thought process and to create a test harness. (Shalloway, 2010) In other words, the architecture should be flexible enough to be changed should a new priority emerge.

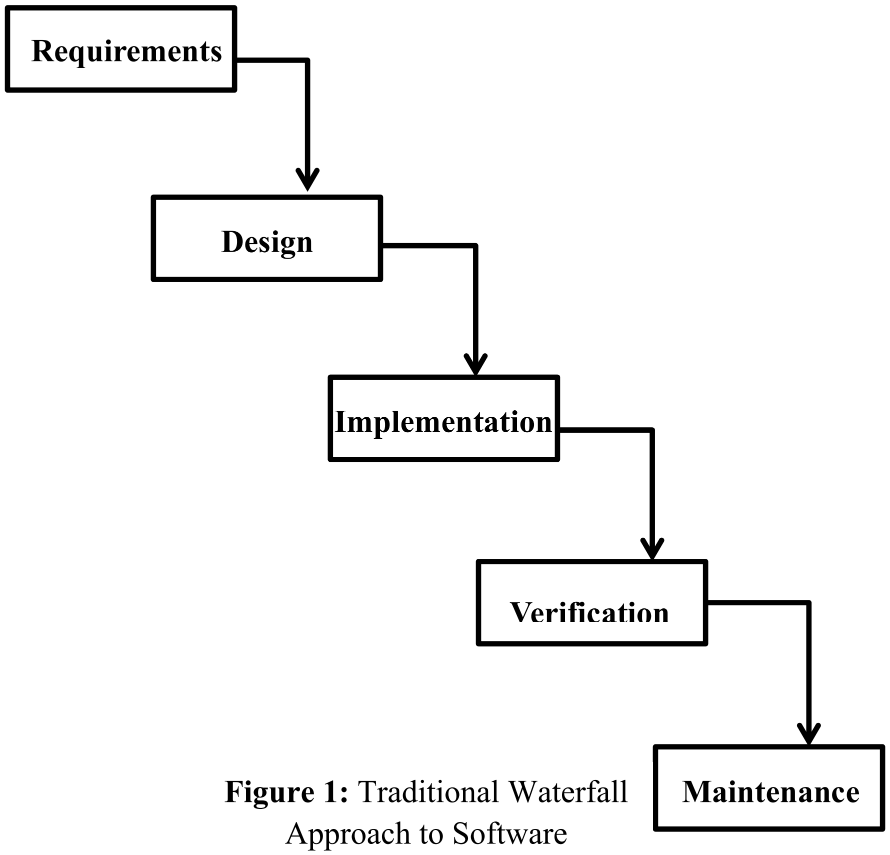


Figure 1: Traditional Waterfall Approach to Software

During each sprint or iteration, agile teams select which user story(ies) they are going to complete. Stories only represent a business need or requirement and do not mention design. It is expected that during the fixed time period known as the sprint that the team will design for the new functionality and incorporate the design into the current design. Sometimes it is not clear as to which technical direction to take so agile teams perform a spike during the beginning of the sprint to aid in design.

Changes in design occur on three specific cases in agile development – a new user story (agile form of requirements) needs to be developed, feedback from a demo requests that system behavior change, or from code refactoring which is the changing of code without changing behavior. Agile teams are continuously evaluating their design and redesigning and these three events are always taking place. Thus, it is important to ensure that all design is able to change without too much rework.

7. Abstract Classes and Interfaces

The behavioral design patterns discussed in this paper make use of two abstract data types within object oriented programming; abstract classes and interfaces. The defining features of each are described below.

Abstract Classes (Wolfgang and Koffman, 2005)

- Allowed to have instance variables and methods
- Allowed to define default values for variables and default implementation for methods
- Cannot be instantiated, only can come into play when an object extending the abstract class is instantiated

Abstract classes provide an abstract view of a real-world entity or concept. They are an ideal mechanism when you want to create something for objects that are closely related in hierarchy, yet distinct. For example, two classes that share some core functionality and store properties and methods common to all extensions of the abstract class, but also have unique instances or methods. The abstract class serves as a good base to start the design. A good example of a possible abstract class and two subclasses which inherit from the abstract class would be a ball as the abstract class, a football as a subclass and a basketball as the other subclass.

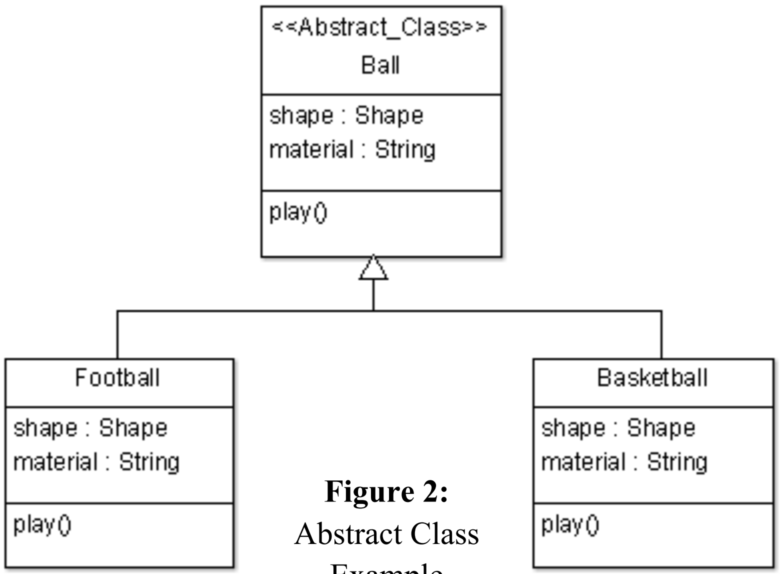


Figure 2: Abstract Class Example

Both subclasses are balls and share some functionality; however a football and a basketball are quite different in many regards such as the way you play with them, their shape, and their material. The difference here could be how you define a method play() which is defined in the abstract class ball but is overridden in the child class football and basketball. Abstract classes help enable design agility in that they can be used to easily extend functionality of a generic class or category and they also limit other developers from instantiating illogical classes. For example, let's consider another example in a software application for a pet store. In the design of this application there exists an abstract class "Dog" that has a few instance variables and the methods Bark() and WagTail(). If a developer wanted to create the class Retriever, which is inherently a Dog and will need to extend the Dog class, the developer cannot possibly create a Retriever that cannot Bark() or WagTail(). The only way of doing so would be to not extend from the Dog class, which would not make it through any code reviews by other members of the team. This is an example how an abstract class can prevent developers from creating any illogical classes.

Interfaces (Wolfgang and Koffman, 2005)

- Define method prototypes
- Must be implemented by another class $^{\text{see 5th bullet}}$
- Cannot have instance variables
- Cannot define default functionality for methods
- Each class implementing the interface must provide implementation for the methods defined in the interface

Interfaces are the mechanism by which components describe what they do (or provide in terms of functionality and/or services). Unlike abstract classes, interface abstractions are appropriate for collections of objects that provide common functionality, but are otherwise unrelated. A software interface defines a set of methods without providing an implementation for them. An interface does not have a constructor – therefore, it cannot be instantiated as a concrete object. Any concrete class that implements the interface must provide implementations for all of the methods listed in the interface. Interfaces are used to create loosely coupled software, to create pluggable software, to allow objects to interact with ease, to hide implementation details and to maintain uniformity in design. All of goals of interfaces support building software that is easy to manage (self-explanatory code, little duplication), well organized with an easy to follow hierarchical structure.

8. Software Design Patterns

Software design patterns provide solutions to common problems introduced in software engineering and design. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibility (Gamma. et al, 1995). Design patterns have been created through experience with object oriented design that have been proven to address common design scenarios.

There are three main types of design patterns.

- **Behavioral** - Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. These patterns describe not just patterns of objects or classes but also the patterns of communication between them. Behavioral patterns characterize complex control flow that's difficult to follow at run time (Gamma. et al, 1995). Behavioral patterns use inheritance and composition to distribute behavior between classes as well as encapsulation to delegate behavior to objects. The behavioral design patterns we will evaluate are discussed in the next section.
- **Structural** - Structural patterns are concerned with how classes and objects are composed to form larger structures. Inheritance is used to compose interfaces and implementations (Gamma et al, 1995). These patterns are typically useful for making a group of classes work together. Structural patterns are also useful for determining ways to compose objects and create new functionality.
- **Creational** - Creational design patterns abstract the instantiation process. They make a system independent of how its object are created, composed, and represented (Gamma. et al, 1995). These patterns use inheritance to change the instantiated class. As systems grow there becomes a greater emphasis on allowing the variation of objects and hardcoding pre-determined properties of the objects becomes less desirable. There are two recurring themes in the creational patterns. First, all of the creational patterns encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of the objects are created and assembled (Gamma. et al, 1995). Creational patterns provide a mechanism to control and manage this creation and reduce the complexity.

This paper will set aside structural and creational design patterns and focus on behavioral design patterns. As mentioned earlier, a primary reason for agile teams to change software is based on customer feedback. Customers do not have knowledge into the architecture of the product such as the management of object creation (creational design patterns) and the composition of structures (structural design patterns). Customers are only concerned with how the product behaves and thus how they interact with it. Of the three design patterns, customers will see and provide feedback on characteristics of the software defined by its behavior, hence behavioral design patterns.

9. What are Behavioral Design Patterns?

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects. These patterns describe not just patterns of objects or classes but also the patterns of communication between them. Behavioral patterns characterize complex control flow that's difficult to follow at run time (Gamma. et al, 1995). Behavioral patterns use inheritance to distribute behavior between classes as well as encapsulation to delegate behavior to objects. There are two main types of behavior design patterns – behavioral object patterns and behavioral class patterns.

Class patterns use inheritance to distribute behavior between classes (Gamma. et al, 1995). This is exemplified by the Template Method design pattern shown later in this paper. Object patterns use object composition rather than inheritance to distribute behavior (Gamma. et al 1995). However, with object patterns, the separate objects need to know of the other objects and managing this becomes an issue as it increases coupling. Some object patterns are concerned about encapsulating object behavior inside of an object. An example which of this is the Iterator design pattern which determines the way aggregated objects are traversed.

9.1 The Chain of Responsibility Design Pattern

Description: The Chain of Responsibility design pattern was intended to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request (Gamma. et al, 1995). As illustrated in Figure 3, the Chain of Responsibility has several participants – a Handler, and Concrete Handler, and a Client. The Handler defines an interface to handle request. The ConcreteHandler handles the requests it is responsible for and can access its successor, and occasionally forwards the request if it cannot to someone who can process the request. Each object has an implicit receiver which handles each request. Each receiver either handles the request or forwards it to a class that can handle the request. The Client initiates the request to a ConcreteHandler on the chain of communication. Each object on the chain shares an interface for handling requests and determining the successor on the chain.

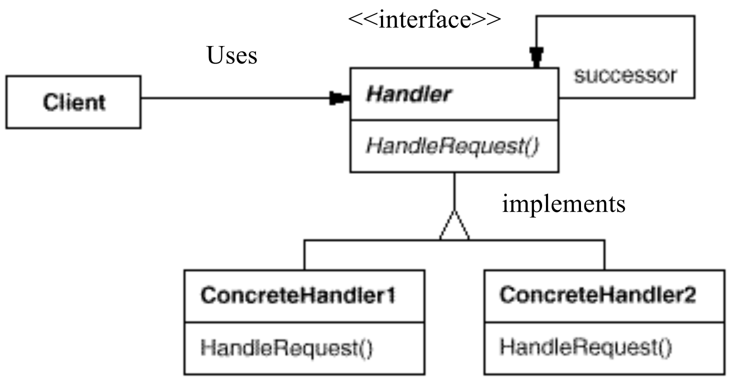


Figure 3: Relationship among classes in the Chain of Responsibility design pattern (Gamma, Erich et al 1995).

Benefits:

- Reduces Coupling – Objects do not need to know who handles the request and only that the request will be handled. Objects also do not need to know about the chain's structure.
- Ability to assign responsibility to objects – In the Chain of Responsibility, a developer or engineer can distribute the responsibility or action among various objects in any fashion that they choose. Responsibilities for handling a request can be changed at run time

meaning that based on the message received; the object can send the request or result of the request to the appropriate chain.

Example Use Case:

The Chain of Responsibility design pattern is used often in user interface development. When a user clicks on a button or widget, a corresponding method or function is called to handle the request. Typically, this will hand off the request to another function and so on until the correct data is returned to the widget or display. Once data is returned from some method in the chain (it is not necessary that the button or widget have knowledge of the entire chain) the widget is updated and the user sees the result.

9.2 The Iterator Design Pattern

Description: The motivation for the Iterator design patterns was to be able to iterate through an aggregate without exposing the underlying details of the aggregate. The Iterator defines various ways of traversing the aggregate. The Iterator design pattern defines four participants – the Iterator, the ConcreteIterator, the Aggregate, and the ConcreteAggregate. The Iterator is an interface for accessing and traversing elements. The ConreteIterator actually implements the iterator interface and keeps track of the current position of traversal in the aggregate. The Aggregate is an interface that is defined for creating an Iterator object. The ConcreteAgrregate implements the Iterator creation interface and returns the correct instance of the ConcreteIterator. Typically, an iterator of a data structure class will contain the methods Next() and First(). Figure 4 below shows an example of the Iterator design pattern.

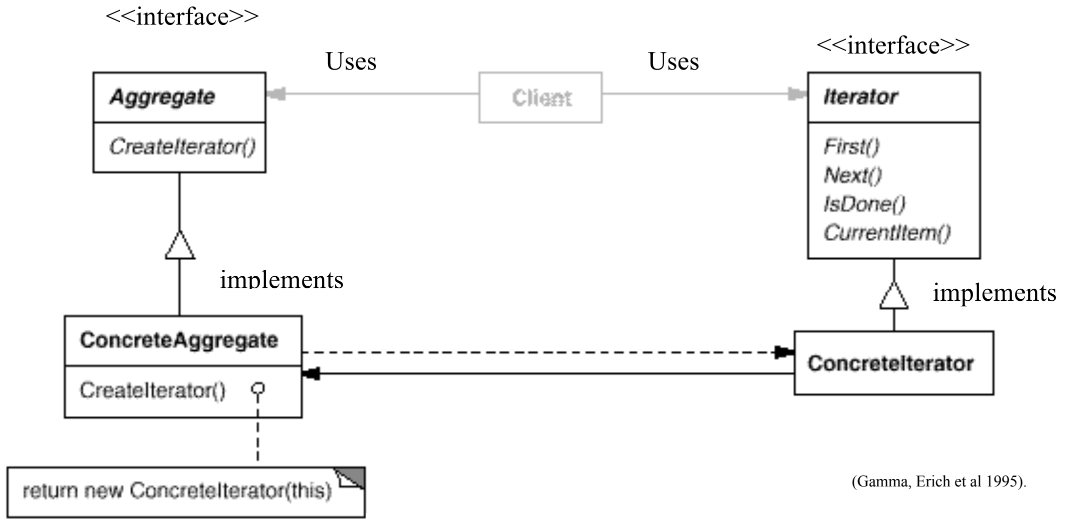


Figure 4: Relationship among classes in the Iterator design pattern

Benefits:

- Supports traversals of various aggregates – Different aggregates have different traversal algorithms. For example, a tree is traversed differently than a list. Different iterators for handling these types of traversals can be defined and used.
- Supports various traversals of aggregates – Iterators allow the developer to change the traversal style by changing the type of iterator.

- Using more than one traversal on an aggregate - An iterator keeps track of its own state in the traversal and this allows for more than one traversal to occur at the same time.

Example Use Case:

The Iterator design pattern is implemented in various data structures and can be implemented for data structures with difficult traversals (such as a tree) or custom data structures created by a developer.

9.3 The Observer Design Pattern

Description: The Observer design pattern was created to address the need of changing various objects' states based on the state of another common object. This pattern defines a one-to-many dependency between objects. When the one object changes, the many are notified and they change as well. This pattern is useful when a change to one object requires changes to a number of objects and the one object does not know how many other objects need to change. This pattern is also useful when one object needs to notify other objects and does not need to know of the other objects. There are four participants in the observer design pattern – The Subject, the Observer, the ConcreteSubject, and the ConcreteObserver. The Subject is an interface that knows who its observers (an Observer object) are and provides an interface for attaching and detaching Observer objects. The Observer is an abstract class that should define an interface to be notified with changes in the Subject. The ConcreteSubject implements the Subject interface and stores the value that is of interest to the Observer objects and sends a notification to the Observer objects when this state changes. The ConcreteObserver implements the Observer interface and keeps a reference to the ConcreteSubject object and stores the value of interest with the subject. Figure 5 shows the structure of the Observer design pattern.

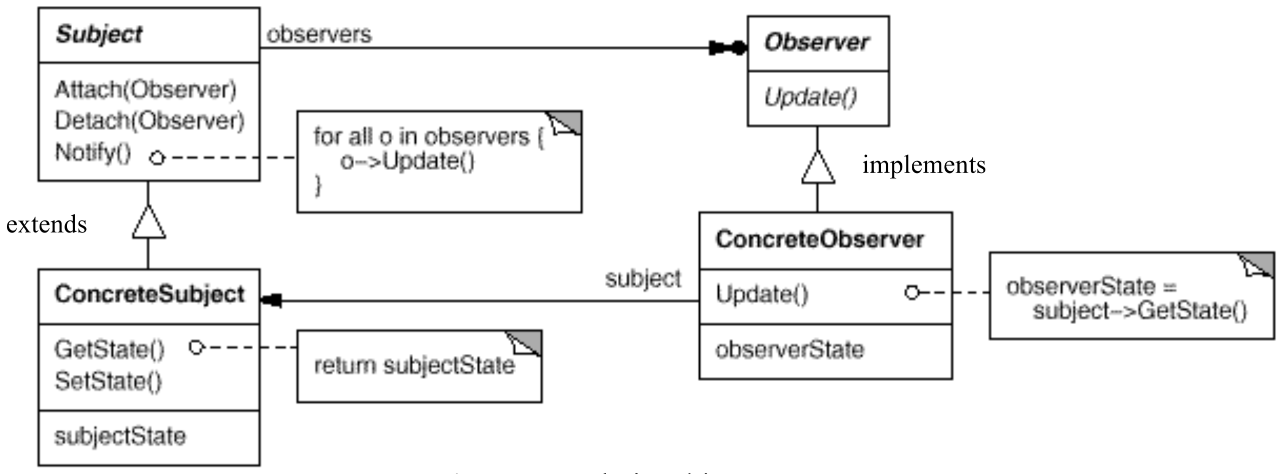


Figure 5: Relationship among classes in the Observer design pattern

(Gamma, Erich et al 1995).

Benefits:

- Decoupling of the Subject and Observer – The subject has no knowledge of the individual observers and only knows that it has observers that implement the Observer class.
- Broadcast messages and communication – This design pattern allows for the communication to a broad number of receivers. The Subject does not need to know of the details of its observers but yet can communicate with several very simply.

Example Use Case:

The Observer design pattern is ideal for any system where changes to a specific component need to be communicated to various components. For example, consider an HVAC system where the temperature can be set using various widgets. A change to one of the widgets would impact a central controller which would need to notify all of the widgets of the change so the appropriate heating or cooling action can be taken.

9.4 The Strategy Design Pattern

Description: The Strategy design pattern was created to enable execution of various algorithms at run time. This algorithm is used when various related classes differ only in their behavior or when you need variations of an algorithm. This pattern can also get rid of overuse of conditional statements used within one algorithm in that the algorithm can be split into different classes. The Strategy design pattern has three participants – A Strategy, the ConcreteStrategy, and the Context. The Strategy is an interface common to all variations of the algorithm. The ConcreteStrategy is the implementation of the algorithm and implements the Strategy Interface. The Context keeps a reference to a Strategy object and has a ConcreteStrategy object.

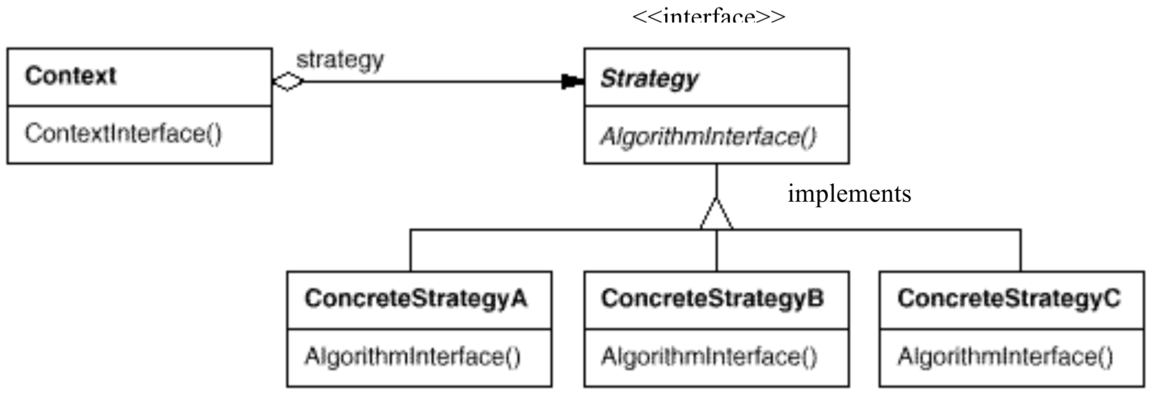


Figure 6: Relationship among classes in the Strategy design pattern (Gamma, Erich et al 1995).

Benefits:

- Allows for the creation of related algorithms – Based on the variations of algorithms, hierarchies of algorithms can form and can eliminate having too many unique strategies.
- Reduces code complexity by removing conditionals – Conditional statements are common in algorithms however if the algorithm becomes laden with too many conditionals, we can break the algorithm into strategies.
- Variation of implementations – The strategies can implement the same behavior but in different ways. This could become useful given resource constraints of the system at a particular time.

Example Use Case:

This design pattern can be used for sorting algorithms on lists and tables. For example, consider a software system that has some sort of internal data structure that can be sorted in various ways. Instead of using conditionals to determine which type of sort to use, the system can be broken down into different strategies which each represent a different version of the sort algorithm.

9.5 The Template Method Design Pattern

Description: The Template Method design pattern was intended to allow the developer to change the steps in an algorithm by deferring pieces of the algorithm to subclasses where the implementation is different. This pattern is used to implement distinct parts of an algorithm once then leave it to the subclasses to define the exact behavior. There are two participants in the Template Method pattern – the AbstractClass and the ConcreteClass. The AbstractClass defines the skeleton of the algorithm and abstract methods that represent steps in the algorithm that will be defined in the concrete subclasses. The ConcreteClass implements the methods or operations needed to carry out the specific steps of the algorithm. The structure of the Template Method Design Patten is shown below in Figure 7.

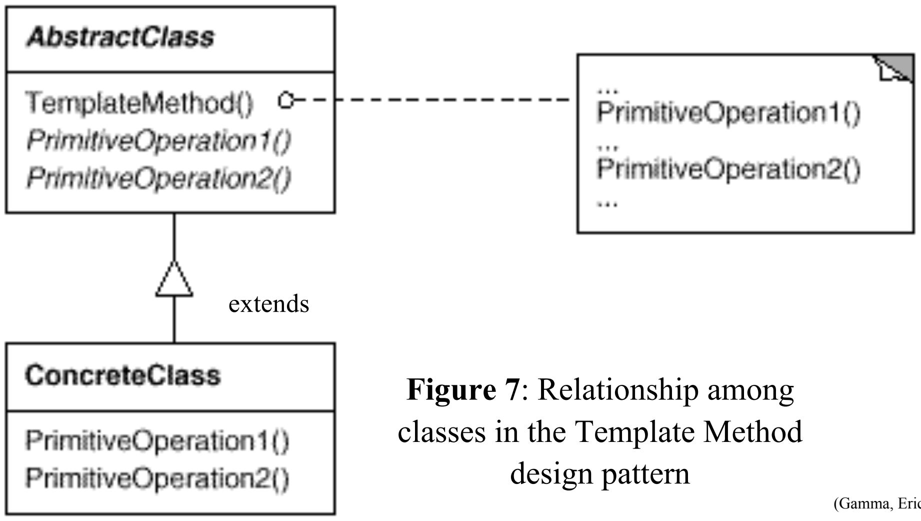


Figure 7: Relationship among classes in the Template Method design pattern

(Gamma, Erich et al 1995).

Benefits:

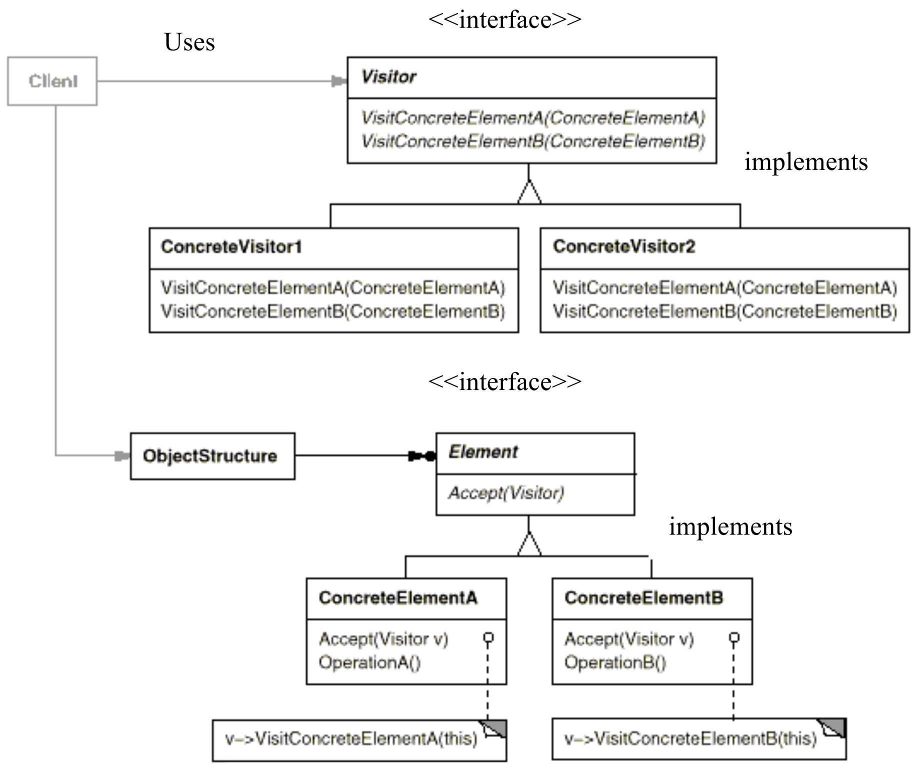
- Fundamental for code reuse which is crucial in software development.

Example Use Case:

Consider an algorithm that compares two objects. Assume this algorithm performs a “less than” calculation on two objects as part of its compare. If the object is simply an Integer, just compare the Integers, however if the object contains various fields, a custom compare will need to be implemented which requires a more involved compare which may involved a different “less than” calculation. These two variations of the primitive “less than” operation can be in their own classes.

9.6 The Visitor Design Pattern

Description: The Visitor design pattern allows one to define a new operation on an object without changing the classes of that object. The Visitor design pattern defines five participants as shown in Figure 8 – The Visitor, the ConcreteVisitor, the Element, the ConcreteElement, and the ObjectStructure. The Visitor declares a Visit operation for each class of the ConcreteElement in the Object Structure (Gamma. et al, 1995). The operation has a name and signature that identifies the class that sends the Visit request to the Visitor. This allows the visitor to know the concrete class of the element that is being visited. The visitor can then access the element directly through this particular interface. The ConcreteVisitor implements the Visitor interface. Each operation in the ConcreteVisitor is defined. This class also provides the context of the algorithm and the state. The Element is an interface that defines an Accept operation that takes a visitor for an argument. The ConcreteElement implements the Element interface and defined the accept operation that takes a visitor as an argument. The ObjectStructure is a composite or an aggregate that has various Elements. It also provides a high level interface to allow the visitor to visit its elements.



<<interface>>
Uses
Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
implements
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
<<interface>>
ObjectStructure
Element
Accept(Visitor)
implements
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v–>VisitConcreteElementA(this)
v–>VisitConcreteElementB(this)

Figure 8: Relationship among classes in the Visitor design pattern

(Gamma, Erich et al 1995).

Benefits:

- Allows adding of new operations – The Visitor allows for the addition of new behavior to an object. To add additional behavior, all that one needs to do is add a new visitor.
- Related behavior is grouped together – In this design pattern, related behavior is localized in a visitor.

Example Use Case:

The Visitor design pattern can be used in many cases however it is particularly useful for handling unanticipated use cases. A well-known example of this is with an abstract syntax tree (see glossary) which is used often in compilers. Depending on the type of node, which represents a construct, visited in the tree, the compiler will need to perform a different operation. By implementing the Visitor design pattern, this outcome can be achieved.

10. Why Agile Teams Should Use Behavioral Design Patterns

So why should agile teams incorporate these design patterns into their architecture? Agile teams create software with “emergent” design. As a reminder, software built by an agile team is not completely defined upfront. The design evolves as new needs i.e. user stories are written. Agile teams also demo software at the end of every iteration or sprint. The feedback gained from these demos is often incorporated into the software systems. However, it is common that the user may request the software to behave differently. Thus agile teams need design that can change behavior without too much rework. The following is a summary of the benefits of using the behavioral design patterns and why these patterns are especially important for agile teams.

- **Distributing Behavior Amongst Objects** - Distributing behavior amongst objects modularizes the behavior of the system or part of the system. This is useful to agile teams in that if behavior needs to be changed, changes only need to be made to the part of the code where the behavior lies as opposed to across the system.
- **Extending Object Behavior** - Since agile teams use frequent customer feedback to guide design decisions, it is very likely that a customer will ask for an extension of current functionality. For example, if a software system has a sort by name feature for a table, the customer may also want to sort by some numerical field. Having the ability to extend object behavior becomes very useful in situations such as this and thus is another reason why agile teams should incorporate behavioral designs patterns in software design.
- **Hiding Access to Internals of a Class** – Hiding the internals of a class is a good practice, especially for agile teams. Since code is collectively owned (Beck, 2005), meaning that the team owns the code and not just one individual, it is very possible that another team member will change code they did not originally write. Therefore it is very important to make sure the internals of certain classes are hidden but have points of access, such as iterators, clearly defined when needed.
- **Reducing Code Complexity** - Maintaining complicated code is not easy. It is especially not easy for agile teams to are constantly refactoring, testing, and re-designing their code. Hence the less complicated the code, the better for the agile team in that maintaining the code base through these frequent changes becomes less of an issue.

- **Reducing Code Redundancy** – Redundant code comes from a lack of separating useful functionality into modules within software design. When there is a lack of useful modules, it becomes difficult to add to the software architecture. Agile teams often add functionality to software systems based on new customer needs and feedback from demonstrations and redundant code would make this occurring activity cumbersome. Thus agile teams would benefit from using behavioral design patterns as code redundancy is reduced.
- **Reducing Coupling** - Coupling is the degree to which two modules within a software system rely on each other. High coupling implies that changes made to one module need to be made to another as well. In addition to changes to system architecture and behavior, agile teams refactor code very often, that is, they make it more efficient in performance and design without changing overall behavioral. By having a low degree of coupling within the software system, agile teams need to spend less time making sure modules are in sync as they are not coupled.

11. Conclusions and Future Work

Agile software development requires that teams continuously reassess and change aspects of their design. If this is not done carefully, the software system will become unwieldy and difficult to maintain and improve. However, there is a solution in that behavioral design patterns can be incorporated into the design of agile development teams. This will allow the system design to emerge with requirements, user interaction the system, and refactoring. This paper has reinforced and elucidated the direct benefits of using behavioral design patterns in agile software development teams. By using these patterns, agile teams can best prepare their software design to change.

The concepts in this paper can be extended to include other design patterns not covered in this paper such as the creational and the structural design patterns. It would be useful to call to attention how using creational and structural design patterns within agile development teams can positively impact the design. It may also be a worthwhile endeavor to gather and record statistical data to determine if agile development teams are actually using the behavioral design patterns referenced in this paper and attempt to quantify the benefits in terms of a metric such as time.

12. Glossary

Abstract Class – A class in object oriented programming that is meant to be extended by a subclass. An abstract class defines methods and instances of a class.

Abstract Syntax Tree – Tree representation of the syntactic structure of source code. Each node in the tree represents a construct occurring in the source code such as “if”, “while”, and “return”.

Aggregate – A data structure in object oriented programming consisting of many objects. Examples are lists and trees.

Class Diagram – A standard UML diagram shows the relationships between different classes in a system.

Class – A class represents an object in object oriented programming.

Constructor – A method contained in every object in object oriented programming. This method is responsible for the creation of the object.

Estimation – A tactic used in agile software development to determine the size of a user story or a task. Estimates for user stories are typically in story points while estimates for tasks are normally in ideal time.

Extreme Programming (XP) - Extreme Programming is a discipline of software development that is based on four values; simplicity, communication, feedback, and courage. Extreme programming implements several agile practices such as iteration planning and pair programming.

Ideal Time – The time it takes to complete an activity with no interruptions.

Interface – A class in object oriented programming that provides method definitions. An interface can be implemented by other classes which means that the implementing class must define all methods in the interface.

Iterations and Sprints – The timeboxed period in which agile teams perform work. Agile teams typically work in 2 to 4 week time periods.

Lean - Lean Software Development is an extension of Lean Manufacturing that focuses on translating Lean Manufacturing principles into the software development environment.

Planning Poker – An activity used to perform estimation. During this activity, each member of a team has a group of cards that represent estimates.

Release – Designates a version of a software system. In agile development, a release represents a system that encompasses several user stories which have a similar or common theme. A release usually makes up several iterations or sprints.

Scrum – An agile software development methodology. In Scrum, teams work in short iterations or sprints that range from 2 weeks to 4 week. There are three primary roles in Scrum; the Product Owner, the Scrum Master, and the Development Team.

Spike – A short experiment to determine the team's direction going forward. An example spike could be a design, analysis of technologies, etc.

Story point – Unit to measure the size of a user story. Size is a combination of time, complexity, risk, and testing requirements.

Timeboxing – A timebox allocates a fixed time period for a particular activity. Iterations and sprints are timeboxed. If the activity is not finished by the expiration of the timebox, that activity is stopped and may be resumed in another iteration or sprint.

UML – (**U**nified **M**odeling **L**anguage) is an object-oriented design language. There are several types of diagrams in UML which all show various views of a system.

User stories – The agile form of requirements, typically written by a user of a software system or a Product Owner. Used to capture the business need behind a software feature.

13. Bibliography

1. Cohn, Mike. *Agile Estimating and Planning.* Pearson Education Inc, 2006
2. Cohn, Mike. *Succeeding with Agile, Software Development Using Scrum.* Pearson Education Inc, 2010.
3. Cohn, Mike. *User Stories Applied.* Pearson Education Inc, 2004
4. Shalloway, Alan and Guy Beaver and Trott, James R. *Lean-Agile Software Development* Pearson Education Inc, 2010
5. Beck, Kent et al *Extreme Programming Explained*. Pearson Education Inc, 2005
6. Shore, James and Shane Warden. *The Art of Agile Development*. O'Reilly Media Inc, 2008.
7. Wolfgang, Paul and Elliot Koffman. *Objects, Abstractions, Data Structures and Design Using Java.* John Wiley and Sons Inc, 2005.
8. Moløkken-Østvold; Kjetil and Magne Jørgensen *Comparison of Software Project Overruns-Flexible versus Sequential Development Models*. IEEE Vol 31, NO. 9, 2005
9. Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995
10. Manning James, *Lean Software Development*. University of Wisconsin – Platteville