An Instance-Oriented Approach to Constructing Product Lines from Layers

George T. Heineman
WPI Computer Science Department
Worcester, MA 01609
heineman@cs.wpi.edu

Abstract

The Model/View/Controller (MVC) paradigm, and its many variants, is a cornerstone of decoupling within object-oriented design. MVC leads to clear reuse benefits regarding the class hierarchies for the model and view elements. In practice, however, the controllers appear to defy reuse, most likely because they encapsulate specialized business logic. Within an effective product line, however, such specialized logic must be reused. We combine the MVC paradigm with feature-oriented programming (FOP) to produce a novel instance-oriented design pattern for layers that brings reusability back to controllers. We demonstrate the effectiveness of our approach using a product-line example of a solitaire game engine.

1. Introduction

A *product line* shares a common set of features developed from a common set of software artifacts [4]. We assume that a *feature* is a unit of functionality within a system that is visible to an end-user and can be used to differentiate members of the product line. One can specify (at the requirements level) that a member of the product line should support a set of features; however, the engineering of the resulting system is complicated because one cannot cleanly encapsulate features as modules to be simply linked together, as with code libraries.

There is a tremendous amount of information in the software engineering literature on features and feature engineering (see [25] for a summary). We are focused on ways to engineer software product lines by synthesizing Model/View/Controller (MVC), feature-based layers, and components. We assume components are designed using an object-oriented language, although the overall approach could still apply to other programming languages, with some effort.

A member of a product line is assembled from a set of components, each of which has been tailored from template components within the software product line [4]. We need to show (1) how to construct the template components and (2) how to create the tailored components. Our solution uses the same mechanism to accomplish both tasks. We rely on existing techniques to specify the overall architecture and assemble the final tailored components into the actual member application of the product line.

We chose to work on this problem because we had developed dozens of plugin components for a card solitaire game engine (described in Section 2.1) used for an undergraduate software engineering course. These solitaire plugins embraced the MVC design pattern and were all members of a product line. While the M/V classes showed excellent reusability and extension via inheritance (30 subclasses within the model hierarchy and 22 subclasses within the view hierarchy), we found it nearly impossible to reuse controllers. Since variation-specific logic was encoded in the controllers, the lack of reusability meant that seemingly similar solitaire variations had virtually no code shared between them. This research effort is our response to this lack of situation.

1.1. MVC and components

MVC is a pervasive technique that separates responsibilities in software to avoid overly restrictive coupling that otherwise might occur [8]. While MVC has most commonly been associated with GUI programming, it can also be applied to separately manage the input, processing, and output of software [14]. The primary benefit of MVC is the resulting extensibility and ease of change. When partnering MVC and product line components, a spectrum of possibilities appears, as shown in Figure 1.

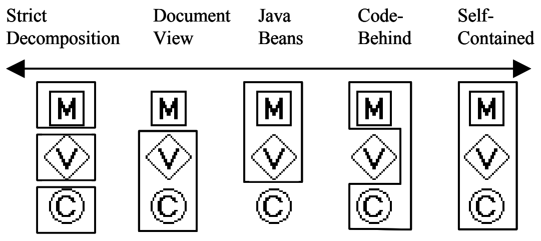


Figure 1. MVC/Component Spectrum.

In Figure 1, the enclosing boxes are units of encapsulation within a component. On the left side, members of the product line are strictly decomposed into tiers, such as User Interface, Workflow & Process Control, Business Services & Legacy Wrapping, and Data & Operating System Services [16]. Components are wholly contained within a given tier and their responsibilities are restricted to those allowed within the tier; for example, an exclusively view component must belong to the User Interface tier. Alternatively, on the right side, a component may be "self-contained", responsible for modeling and storing state information, performing computations over this state, and presenting an interface to the user. In between there are distinct families of possibilities: *DocumentView* [27] merges V/C to operate over a model document; the JavaBeans component architecture [20] effectively merges M/V using properties; finally, *Code-Behind* refactoring [21] separates the view from the M/C.

Regardless of the way in which MVC and components are integrated, we must consider how to tailor a template component to support a particular feature. The product line community has developed numerous approaches [2][12][17][24][26]:

- Parameterization (either build-time via compiler directives or run-time arguments)
- Inheritance and Polymorphism
- Delegation
- Extensions and Extension Points (built-in Variability Mechanisms within the component)
- Component substitution
- Adaptable Software components [10]
- Code generation (when using a higher-level language to define desired properties).

Most, if not all, of these mechanisms rely on the language itself in which the components are programmed. Many also rely on run-time testing of designs that should have been validated at design-time. We seek an approach that captures the higher-level concept of features, scales to enable components to be extended with multiple sets of features, and validates the tailored component at design time.

While features may cross-cut numerous components, one common characteristic is that the feature implementation itself can be subdivided into its effect on the MVC. New classes may need to be added to the model, or existing models refined; new view classes may compose information from one or more models, or existing views refined. Most importantly, we believe, new controller classes will need to be defined to contain the specific unique business logic required by individual features, or existing controllers refined. Product line designers must simultaneously manage the MVC artifacts within a template component **and** the tailored components. Our observation of the relationship between feature layers and MVC was an important step in understanding how to solve the problem.

Given the sheer number of possible members of a product line, we need a way to rapidly and safely assembly the tailored components to be used. This paper proposes a model that defines this capability, describes a prototype tool that builds on top of Batory's AHEAD tool suite [3] to perform the approach, and presents and evaluates a case study showing the practical application of the approach.

We first state the requirements that drive this research followed by a detailed description of the solitaire game engine product line. Section 3 presents our formal model together with a full discussion of the layered solution to the product line. Section 4 evaluates more fully the instance-oriented layered design against more traditional OO solutions. We conclude with related work and more information about the AHEAD Component Development Kit (ACDK). Section 6 describes lessons learned and presents future work.

2. Requirements

A design method for a product line starts with the definition of a common architecture and then captures any variabilities; but then it needs to go further. It must:

- Enable the assembly of a valid product line member from primitive building blocks, paying special attention to the interaction between these building blocks. The definition of *validity*, its specification and evaluation, is flexible.

- Provide traceability to bridge information captured within use case scenarios that describe the variability and the underlying design and implementation of the resultant system.
- Incorporate an existing Feature-oriented model instead of developing a new one.

In a product line system, one needs to organize the design artifacts to better support at the implementation level the allowable composition of PL features. We also don't want to force a major rewrite of the software base, so we must show how to organize existing object-oriented classes into feature-oriented layers.

2.1 Product Line Domain Example

Kombat Solitaire (KS) is a Java application that enables head-to-head competition of solitaire variations played simultaneously over the Internet. KS was developed as part of an undergraduate software engineering course. Each plugin represents a single solitaire variation. KS is constructed from a set of components – such as, userManager, pluginManager, client, server – and the individual solitaire variations can be loaded as plugins. We have accumulated nearly twenty-five different solitaire variations. Because of the commonality among variations, we knew there must be a better way to design and implement these plugins, which is why we investigated using AHEAD. The solitaire game engine itself is an excellent case study in product lines, since it offers four distinct members (in addition to the plugin components that remain the focus of this paper):

- PT – Solitaire Plugin Tester
- DA – Stand-alone Desktop application
- DS – Distributed Solitaire Repository
- KS – Head to Head competition over Internet

KS (version 2.2.1) contains about 67K lines of Java code, of which 31K forms the core Solitaire playing engine. To support KS, we also developed a small application PT that simply enabled one to execute and test a solitaire plugin DA and DS have not yet been developed because of limited resources. However, given the success of the product line approach regarding the plugin variations, our future work includes building these product line members from the same set of software components.

2.1.1. Plugin Design. To enable the rapid development of solitaire plugins, a rich set of model elements are already provided, as shown in Table 1. Each model element shown (except for abstract **Stack**) has a corresponding view element that depicts the model element within the solitaire playing field.

Each KS plugin is responsible for constructing a model of the game, which may include a deck, columns where cards are stacked, a running score, and waste piles. The plugin then defines the views for these model elements over a 2-dimensional playing field such that no two views intersect each other. Finally, a controller is registered with each view to manage mouse events (press, release, click) and perform moves as allowed by the solitaire variation. The sum total of all the controllers enforces the rules of a solitaire variation.

Table 1. Classes within KS Model Hierarchy

Stack	abstract representation of cards in sequence from bottom to top
BuildablePile	pile of cards face down on top of which a column can be built (as in Klondike)
Card	single card
Column	stack of cards that reveals cards lower in the column
Deck	deck of playing cards
MutableInteger	integer that can change during play (such as the score)
MutableString	string that can change during play
Pile	stack whose topmost card is visible

2.1.2. Object-oriented Support for MVC. The object-oriented paradigm rapidly converged on use cases during analysis to identify the desired objects. Use cases capture a slice of functionality in a system that is initiated by an actor and involves interactions with specific elements within the system. As designers develop use cases, there are two common modeling relationships:

- `«extend»` – When a use case adds behavior to a use case without changing the original use case.
- `«include»` – When a use case contains behavior that is shared by multiple use cases.

Once captured in the analysis model, these relationships can be used to identify reuse possibilities in the underlying object model [1], though many advise against trying to transfer these relationships into class inheritance [13][15]. Jacobson also advises to "never extend an extension" of an existing use case to avoid complexity [13]. Layers provide a more amenable artifact to the relationships between use cases; we return to this point in Section 3.1.

Both Jacobson [10] and Cockburn [5] describe an approach to identify objects from use cases by defining three overall divisions – Entity, Boundary (or Interface to Jacobson), and Control. Entity objects represent the persistent data used by an application. Boundary objects provide the functionality to interact with the environment and receive requests from system actors. Control objects contain functionality "not contained in any other object" and encapsulate business logic. These divisions are reflective of the MVC division that appeared quite early in the evolution of object-orientation, starting with SmallTalk.

The premise of this paper is that using MVC naturally leads to the inability to reuse controllers. Domain experts have considerable expertise in using inheritance to capture the rich information to be stored in a model. HCI experts show how to build user interfaces that decouple the model from the view presented to the users. But the complex logic found in controllers can quickly be unmanageable because of the inherent limitations of the basic extension constructs in OO programming languages. Since business logic is encapsulated within controllers, MVC may actually be an impediment to the proper reuse or extension of business logic.

Rather quickly one sees the limitations of using inheritance (a typing mechanism) as a means of capturing the way that one (complex) behavior is related to, or extends, another; this is especially true when one requires multiple sets of simultaneous extensions. To manage the multiple tailoring of several components within a product line, we must provide a more rigorous foundation.

3. Formal Model

When a product line member exhibits a set of n features, we say that $m_{pl} = \{FE_1, FE_2, \ldots, FE_n\}$. m_{pl} is constructed from a set of components $\{C_1, C_2, \ldots, C_k\}$ according to the architectural definition of the product line. Because features can cross-cut multiple components, we define a feature implementation FE_i to be a *k-sized* vector whose elements are $\mathrm{fe}_{i,j}$, fragments of feature FE_i that are composed into component C_j. When a feature is located entirely within a component, its vector contains only one non-empty element. The definition of m_{pl} is thus a set of k equations, one for each component C_j, of the form $\mathrm{fe}_{1,j} \bullet \mathrm{fe}_{2,j} \bullet \ldots \bullet \mathrm{fe}_{n,j}$.

The compose operator $\bullet$ is as defined by Batory, thus each $\mathrm{fe}_{i,j}$ is an AHEAD layer [3]. Each layer $\mathbf{l}\ (\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_m})$ contains a set of m Jak artifacts that are composed together to produce a set of Java classes. Each artifact $\mathbf{a_i}$ is either a refinement of an existing class or a newly defined class. The equation $[\mathbf{h}\ (\mathbf{a_1}, \mathbf{a_3}) \bullet \mathbf{j}(\mathbf{a_2}, \mathbf{a_3})]$ will result in three artifacts and the order of the composition shows that design artifact $\mathbf{a_3}$ in $\mathbf{h}$ refines the existing design artifact $\mathbf{a_3}$ in $\mathbf{j}$.

Each layer can define whether it is *constant* (i.e., forms a base artifact) and if it is *single* (i.e., can only appear once in an equation). Layers can declare their *requirements* and their *provisions*. Provisions and requirements are directional; for example, if an equation composes a layer L_i with a *flowleft* requirement, then that requirement is satisfied if some layer L_j to the right of L_i has a *flowleft* provision. Given a layer **h** in an equation, layers to the left of **h** are "downstream", since they are being composed after **h**, while "upstream" layers are to the right of **h**.

The use of MVC was critical in our understanding of constructing components from composed behaviors. The essential point is that we show how to build complex component behaviors by assembling reusable primitive behaviors defined in layers.

3.1 Extensions to Batory's AHEAD

While we use Batory's AHEAD tool suite "as is", we make three novel contributions. (1) `jak2java` composes layers "in place", which makes it hard to reuse layers. ACDK transparently manages layers in an equation by reference, copying all layers into a temporary location when composition is required; (2) ACDK provides a developer interface that enables the GUI construction of layers, supports arbitrary search through all layers (both Jak files and composed Java files). ACDK enables the rapid prototyping of layer compositions; (3) we developed the *instance-oriented layered* style of design.

In *instance-oriented layered* design, we partner MVC with layers. Layers can introduce new "types" which are like object factories [8]; as "instance" layers are composed downstream, refining the type layer, objects of that type are constructed. Using the chain of responsibility pattern [8], each layer performs its task, and then invokes the appropriate logic on the upstream layer (similar to the way subclasses should invoke `super()` in constructors).

Use cases "roughly" (by our experience) map into layers. Each use case that «`extends`» a use case becomes a layer that refines an existing upstream layer; use cases that «`include`» a base use case translate into layers that have a flowleft requirement provided by the layer representing that base use case. When features can be described as extensions or additions to existing use cases, our methodology quite nicely bridges the gap between requirements and code as it appears in layers.

3.2 Visitor Pattern Example

To provide a complete example, consider the set of classes shown in Figure 2 where a **Layer** is composed of a set of **Concerns** (these classes are selected from ACDK itself). Assume that the white classes form the base of a piece of software, where there are six subclasses of **Concern**s, five of which are directed.

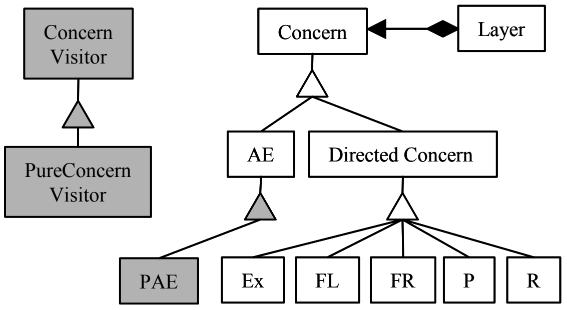


Figure 2. Visitor Pattern Example.

Starting from this base, assume a designer wished to add the visitor pattern [8] because processing over the **Concern**s using an Iterator returned by **Layer** was too complex. We construct an **aggregate** layer containing these nine classes; this layer is *constant*, to use AHEAD terminology. To add the visitor design pattern in Java requires changes to each class. Using ACDK, we construct a new **visitor** layer that refines each class (by adding the method `void accept (ConcernVisitor cv)`) and adds two new classes (shaded in the upper left corner of Figure 2). Our equation is now **[visitor ● aggregate]**.

Next, the designer adds a new subclass, PAE, to the AE class. A new **specialization** layer is composed, requiring refinements to both visitor classes, as well as defining the new PAE subclass. The final equation is:

**[specialization (ConcernVisitor, PureConcernVisitor, PAE) ●
visitor (Concern, AE, Ex, FL, FR, P, R, Layer, ConcernVisitor, PureConcernVisitor) ●
aggregate (Concern, DirectedConcern, AE, Ex, FL, FR, P, R, Layer)]**

3.3 Solitaire Domain Revisited as Layers

We now describe how we developed a set of layers that can be assembled to form solitaire variation plugins. The **game** layer describes the empty solitaire plugin; it is analogous to an abstract base class except it can be instantiated and it generates a working plugin (albeit with no real behavior). **game (Game, Layout)** is shown in Figure 3 (all details of the actual implementation are omitted unless relevant). Note that the equation **[game]** is invalid because of the unsatisfied flowright requirements.

game layer	
flowright	**flowleft**
→ requires scoreDefined → requires numCardsLeftDefined	← provides pluginBase

Game
M
	public class Game extends Solitaire
	+ Game ()
	+ Layout getLayout() // return object that places widgets
	+ boolean hasWon() // has variation been won?
	+ String getName() // name of variation
	+ void initialize() // build controllers for playingArea
	+ void setDefaultControllers (Widget) // … for widget

Layout
V
	public class Layout {
	+ Layout (CardImages ci) // Layout needs card images
	+ CardImages getCards() // return card images
	+ void setLocation (Widget) // refined by other layers

integer layer	
flowright	**flowleft**
	← provides integerDefined

Game
M
	refines class Game
	+ void resetHand() // clean up
	+ void initialize() // as needed for integer
	+ IntegerManager getIntegerManager() // expose

IntegerManager
V
	refines class IntegerManager
	+ void setLocation (Widget) //place widget
	+ void createInteger() //create model
	+ void createIntegerView() //create view

numLeft layer	
flowright	**flowleft**
→ provides numCardsLeftDefined	← requires integerDefined

IntegerManager
M V
	refines class IntegerManager
	+ void createInteger () // add to Model
	+ void createIntegerView() // add to View

score layer	
flowright	**flowleft**
→ provides scoreDefined	← requires integerDefined

IntegerManager
M V
	refines class IntegerManager
	+ void createInteger () // add to Model
	+ void createIntegerView() // add to View

layout layer	
flowright	**flowleft**
	← requires pluginBase
IntegerManager V	refines class IntegerManager + void setLocation (Widget) //place widget

Figure 3. Definition of layers.

The **Game** artifact is part of the model while the **Layout** artifact belongs to the view. Given the existing layers defined for the solitaire product line family, the first valid equation E_1 is **[score ● numLeft ● integer ● game]**. Reading from right to left (as we must with equation compositions) this composes with **game** the **integer** layer (which introduces the type of integer) and then two instance layers (which add the elements of the number of cards left together with the score of a solitaire game). This equation, while correct, produces a solitaire plugin whose playingArea is empty; we need to compose a specialized **layout** layer that knows how to place the integer widgets on the screen. The final equation is **[layout ● E_I]**.

To see how these layers interact, consider the `initialize` method. The **integer** layer refines **game**, which means that it will first receive control when `initialize` is invoked; it performs its task by calling `createInteger` and `createIntegerView`. Within **integer** these methods are empty, but **numLeft** and **score** refine the methods to create a chain of responsibilities [8] where *score* M/V elements are first created, and then *numberCardsLeft* M/V elements. As each IntegerView element is created, `setLocation` is invoked. The **integer** layer provides the definition of this method, but it only has meaning when the **layout** layer refines the method to properly layout the score and numCardsLeft widgets. The interactions between the layers show how (1) decisions are deferred to downstream layers; and (2) layers refine behaviors of upstream layers.

4. Evaluation

Table 2 compares the reusability factor for ACDK-generated layers of four solitaire plugin components against their hand-coded counterparts. Note that we omit references to "core" classes provided by the KS model and view hierarchy, since these are used as is by both solutions; we are interested in identifying opportunities for reuse across the solitaire variations.

Table 2. Reusability Comparison

	Java	ACDK
	#Classes (#reused)	# Layers (#reused) %
Idiot	6 (0)	16 (13) 81%
Narcotic	7 (0)	17 (13) 76%
GrandFatherClock	6 (0)	31 (29) 93%
Klondike	11 (0)	31 (25) 80%

The ACDK equations for these plugin components are as follows (* means unique to the variation, a number means the number of times the layer is composed):

Idiot: **[stacktostack ● layout* ● solve* ● rules* ● decktostacks ● aCol^4 ● column ● aDeck ● deck ● numCardsLeft ● score ● integer ● game]**

Narcotic: **[solve* ● rules* ● stacktostack ● reassembleDeck* ● layout* ● decktostacks ● aPile^4 ● pile ● aDeck ● deck ● numCardsLeft ● score ● integer ● game]**

GrandfatherClock: **[layout* ● aDeck ● rules* ● stacktostack ● aPile^{12} ● aCol^8 ● numCardsLeft ● score ● deck ● pile ● column ● integer ● game]**

Klondike: **[rules* ● buildablePileMoves* ● pileMoves* ● restockDeck ● flipCard ● stacktostack ● deckMoves* ● deal* ● klondikeLayout ● aFanPile ● fanpile ● aPile^4 ● pile ● aBuildablePile^8 ● buildablepile ● aDeck ● deck ● numCardsLeft ● score ● integer ● game]**

As the reader can verify, the ACDK solutions showed tremendous gains in reusability.

4.1 Comparison with other OO Solitaire Engine

The lack of reusability within KS could simply have been poor design and/or programming. To determine whether this was the case, we compare KS with an open source object-oriented solitaire game engine, PySol [22]. PySol is written in Python, an interpreted object-oriented programming language [23] that uses an easy-to-read syntax. PySol has an extensible solitaire engine and supports features such as multi-level undo/redo, loading and saving games, storing statistics, help, and hints for next moves.

In PySol, each solitaire **Game** has a *talon* that holds the initial deck, a *waste* pile of cards dealt from the talon, a set of *foundation* piles where cards are placed for the final solution, a set of *row* piles to hold intermediate storage as allowed by the solitaire variation being, a set of additional *reserve* piles for holding cards, and a set of *internal* piles that are invisible during game play and are used to simplify the coding of a particular variation. The **Game** class thus provides a rich set of primitive objects that the PySol designers expected would be in any variation.

The definition of **Klondike** as an extension to the base **Game** class is shown in Figure 4. The behavior for the Klondike variation is encoded in several ways: (a) By fixing the class for an object to determine allowable moves (i.e., in Klondike the foundation piles are **S**ame **S**uit piles of increasing card rank, and the row piles must be **A**lternating **C**olor and start with a King if empty). The definitions of **SS_FoundationStack** and **KingAC_RowStack** are provided by the PySol infrastructure, and are themselves extensions of abstract base classes.

It is clear that PySol satisfies its main objective of providing an extensible engine for solitaire games (with over 200 variations). Yet the design has flaws:

- In PySol, there is no separation of Model, View, and Controller. In fact, it supports what it calls a "pseudo MVC scheme" by creating three class variables `model`, `view`, and `controller` that are all set to `self`, the python version of `this`! The **Stack** class has 23 methods that access/update the model, 15 that access/update the view, and 31 methods that access/update a controller. OBSERVATION: the design is complex.

- If a new variation requires a specialized layout, the **Layout** class must be modified to include a method written for the new variation. For example, the `freeCellLayout` method in **Layout** exists only for use by the FreeCell variation. OBSERVATION: avoid changes to core classes just to encode a variation.
- Often logic for a variation is spread throughout multiple Python modules. In PySol, one can use an integer seed to select a random game. If the same seed is used, the deck will be shuffled identically. Because FreeCell is so popular, the base **Game** class in PySol has a sub-case (used only by FreeCell) that will shuffle the deck to appear exactly as it would have if played on Windows. OBSERVATION: avoid intermingling specific with generic functionality.
- Much of the logic is embedded within the objects themselves. In **Klondike** in Figure 4, for example, the **WasteTalonStack** knows that the cards dealt from the talon end up in the waste pile. OBSERVATION: separate model from view.

The KS and PySol approaches offer similar solutions: Reuse existing classes "as is" where possible to construct the solitaire game, and extend hierarchy classes with specialized logic to encode variations. Two variations of Klondike allow for 1 or 3 cards to be dealt from the talon, and for multiple re-deals once the talon is exhausted. To realize all four possible variations, KS and PySol would do the following:

- In PySol, **Klondike** offers pre-defined flexibility by the `createGame` method, where the invoker can specify the number of cards dealt (`num_deal`) and the number of allowed rounds (`max_rounds`). Each subclass of **Klondike** would be required to have its own `createGame` method to define the proper values.
- In KS, the controllers encode the logic for the variations and would be parameterized with `num_deal` and `max_rounds` information to prevent illegal moves.

These solutions are indeed serviceable, yet the concepts of multiple-card deals, or multiple rounds, is more general and would likely appear in lots of other solitaire variations. For example, PySol, has three classes, **FreeCell_AC_RowStack**, **Spider_AC_RowStack**, **Yukon_AC_RowStack**; all ensure that cards are in alternating colors/decreasing rank, but additional variation-specific logic is woven together. In addition, the PySol designers have "fixed in concrete" the possible variation points through parameters.

```
class Klondike(Game):
    Layout_Method = Layout.klondikeLayout
    Talon_Class = WasteTalonStack
    Foundation_Class = SS_FoundationStack
    RowStack_Class = KingAC_RowStack
    Hint_Class = KlondikeType_Hint

  def createGame(self, max_rounds=-1,
                 num_deal=1, **layout):
    # create layout
    l, s = Layout(self), self.s
    kwdefault(layout, rows=7, waste=1,
              texts=1, playcards=16)
    apply(self.Layout_Method, (l,), layout)
    self.setSize(l.size[0], l.size[1])

    # create stacks
    s.talon = self.Talon_Class(l.s.talon.x,
                l.s.talon.y, self,
                max_rounds=max_rounds,
                num_deal=num_deal)

    if l.s.waste:
      s.waste = WasteStack(l.s.waste.x,
                l.s.waste.y, self)
    for r in l.s.foundations:
      s.foundations.append
        (self.Foundation_Class(r.x, r.y, self,
           suit=r.suit))
    for r in l.s.rows:
     s.rows.append(self.RowStack_Class(r.x,
           r.y, self))
    # default
    l.defaultAll()
    return l

  def startGame(self, flip=0, reverse=1):
    for i in range(1, len(self.s.rows)):
      self.s.talon.dealRow
         (rows=self.s.rows[i:], flip=flip,
          frames=0, reverse=reverse)
  self.startDealSample()
  self.s.talon.dealRow(reverse=reverse)

  # deal first card to WasteStack (if exists)
  if self.s.waste:
    self.s.talon.dealCards()

  def shallHighlightMatch(self, stack1, card1,
                          stack2, card2):
    return (card1.color != card2.color and
             (card1.rank + 1 == card2.rank or
              card2.rank + 1 == card1.rank))
```

Figure 4. Klondike PySol Implementation.

It is inappropriate to localize variation-specific logic in **Klondike**, but it is equally incorrect to "pollute" **Game** or **Layout** with arbitrary logic that appears only within a few (or even one) variations. We find that instance-oriented layered design enables us to assemble a valid component variation from primitive building blocks, paying special attention to the interaction between these building blocks.

4.2 LOC Comparison between KS and ACDK

The complete Klondike assembly consists of 31 layers and 17 Jak entities consisting of 2,879 lines of Jak. The total composed Java files account for 3,089 lines of code. This implementation compares against a

Klondike implementation created manually within KS that consisted of 12 Java classes and 1,632 LOC.

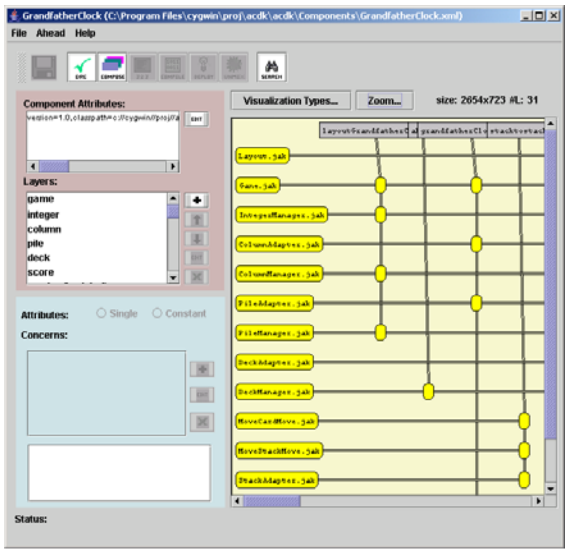
4.3 Extending to Full KS Component Set

The majority of this work was focused on constructing solitaire plugin components using ACDK. The template component, in this case, was the set of base MVC classes and the tailored component was the resulting plugin. We now briefly show how the approach extends to the “core” components that make up the KS application. The userManager component within KS is responsible for storing statistical information for each user about games played (such as number of games lost or won). If we wanted to store variation-specific information (in Klondike, for example, how many cards remained face down) we would compose a new feature $FE_i = fe_{i,1} \bullet fe_{i,2} \bullet fe_{i,3}$ where $fe_{i,1}$ represents the feature fragment layer composed with userManager, $fe_{i,2}$ represents the layer composed with the Klondike plugin, and $fe_{i,3}$ represents the layer composed with the pluginManager that ensures at run-time that only the Klondike plugin can be loaded.

The more routine form of layered equations within KS would relate to the core features visible to the users – creating virtual tables for solitaire games to be played over the Internet, or a chat subsystem. The communication protocol describes the full set of responsibilities for the KS client and KS server. One can construct stripped down (or super-enhanced) client and server applications by composing appropriate features, as required by the individual components that make up KS.

4.4 ACDK prototype

Figure 5 contains a screenshot showing how the Grandfather Solitaire game was assembled using ACDK. A component is constructed by a set of layers. As each layer is created or added from a library of existing layers, the graphical visualization on the right side reflects the structure with columns representing layers. Each Jak artifact appears as a node within a column; horizontal lines represent refinements of Jak artifacts. The full set of Jak artifacts appears in the leftmost column. The AHEAD tools can be invoked by the toolbar at the top of the window, and one can search for strings in all layers and Jak artifacts. ACDK offers an alternative visualization showing the flowleft and flowright concerns for the layers.


GrandfatherClock (C:\Program Files\cygwin\proj\acdk\acdk\Components\GrandfatherClock.xml)

Component Attributes: version=1.0,classpath=c:\\cygwin\\proj\\a

Layers: game, integer, column, pile, deck, score

size: 2654x723 #L: 31

Figure 5. Sample ACDK screenshot.

5. Related Work

Our instance-oriented approach to constructing product lines is based on Batory’s elegant notion of feature-oriented composition using layers [3]. In Section 3.1 we outlined our extensions. We believe ACDK introduces a new design pattern for layered-based designs, analogous to the design patterns for OO design [8]. More experience with layered design will naturally lead towards better understanding of the best practices in the area.

The most closely related concept to our work is the Presentation/Abstraction/Controller (PAC) design pattern that forms a hierarchy of agents, each of whom is responsible for a particular aspect of system functionality. The primary limitation of using PAC is its complexity. First, one must select the appropriate level of granularity for each PAC; second, the control components increasingly become mediators between the Abstraction/Presentation, as well as with other PAC agents. Third, PAC agents are distinct objects and do not share the ability of AOP or AHEAD to compose together and share state. Finally, while PAC is extensible, allowing one to readily insert new PAC agents into an existing hierarchy, the lifecycle management of the agents quickly becomes a major concern. What we are able to accomplish, essentially, is use the AHEAD tool suite to compose together a set of layers so there is no need to maintain or instantiate objects for each individual layer, as one would need to do for each PAC object.

AOP shares much of the concerns of this paper; however, one common shortcoming is that it does not scale when several aspects are to be woven together over the same artifact. The problem may be the lack of fine-grained control over the ordering of the weaving. AOP simply fails to lay the foundation for designed variability because of its focus on the implementation artifacts. Some methodologies, realizing this limitation, have sought to model the generic creation and customization of modules. OPM is a rich modeling methodology [7], whose weakness appears to be a steep learning curve and lack of visibility in the greater community. These will vanish in the future, at which point OPM will be a serious contender for the way one models, designs, and builds software.

Inheritance and delegation both offer mechanisms to extend existing behavior by "bracketing" a method invocation; a delegate can intercept a method request and perform additional work before and after. With inheritance, a subclass can override a method `C.m()` with `{preWork();C.m();postWork();}`. While these techniques work well for "localized" behavioral modifications, they simply do not scale when unanticipated (seemingly arbitrary) behaviors need to be composed together. The instance-oriented layered design relies on the Chain of Responsibility pattern as well as the Factory Method pattern [8].

6. Lessons Learned and Future Work

Reusable Controllers. One of the most challenging problems with the object-oriented application of MVC is the lack of reuse within the controllers. This limitation must be overcome because most OO methodologies place complex business logic within controllers. Indeed, in many applications of MVC, the controller is simply defined as an interface, limiting reuse opportunities. In our own anecdotal experience in developing KS, we found no controller reuse (in fact, most controllers were created via copy/paste).

Better Change Management. Once a product line is designed, and various members constructed, there is a natural hesitation to make changes to the base classes, for fear of breaking existing working software. Using layers, one can cleanly encapsulate changes with minimal impact on existing code. Indeed, to "back out" of a proposed change, one need only delete the layer containing the change.

Coarse-grained Composition Techniques. Using the AHEAD tool suite, a layer can only (1) add a new class; (2) refine the methods of an existing class (add, override, extend); or (3) add new fields to an existing class. One can envision more fine-grained composition techniques that require more sophisticated mechanisms for weaving Jak files into Java classes; for example, a layer could add a new `case` statement to an existing `switch` statement. We believe that the elegance of the AHEAD refinements – and their simplicity – made possible the success of ACDK.

The KS and ACDK software packages are available for download from http://www.cs.wpi.edu/~heineman

6.1. Future Work

The decomposition into feature layers that we have proposed is also compatible with other research areas that seek, for example, to check the validity of the composition of features by validating individual features in modular fashion [17]. We currently use the existing ability of AHEAD to specify provides and required information for each layer; we will consider in the future more sophisticated means of specifying the *interface* for a layer and validating that compositions satisfy all interface specifications.

6.2 Acknowledgements

We would like to thank Don Batory for the many conversations we have shared regarding the described work, as well as for his excellent AHEAD tool suite.

References

[1] S. Ambler, *The Object Primer, 3rd Edition, Agile Model Driven Development with UML 2*, Cambridge University Press, 2004.

[2] P. America, H. Obbink, R.van Ommering, and F. van der Linden, "CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering," P. Donohoe, Ed., *Software Product Lines: Experience and Research Directions*, Kluwer Publications, pp. 167-180, Aug. 2000.

[3] D. Batory, J. Sarvela, and A. Rauschmayer, Scaling Stepwise Refinement, *International Conference on Software Engineering*, Portland, Oregon, May, 2003.

[4] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, Boston, MA, 2002.

[5] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000.

[6] J. Coutaz, "PAC, an Object Oriented Model for Dialog Design". In Rullinger, H. I. and Shackel, R., Eds., *Human-Computer Interaction - INTERACT*. Elsevier Science Publishers, 1987, pp 431-436.

[7] Dov Dori, *Object Process Methodology*, Springer-Verlag, August 2002.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison Wesley, 1995.

[9] M. Griss, "Implementing Product-Line Features with Component Reuse", 6th International Conference on Software Reuse (ICSR), Springer-Verlag, Vienna, Austria, June 2000.

[10] G. T. Heineman, A Model for Designing Adaptable Software Components, *22nd Annual International Computer Science and Application Conference*, pp. 121-127, Vienna, Austria, Aug. 1998.

[11] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach,* Addison-Wesley, 1992.

[12] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success*, Addison-Wesley, 1997.

[13] I. Jacobson, Use Cases: Yesterday, Today, and Tomorrow, Rational Technical Library, Nov. 2003, http://www-106.ibm.com/developerworks/rational/library/775.html

[14] B. Kotec, MVC design pattern brings about better organization and code reuse, *Builder.com: beyond the code*, October 2002, http://builder.com.com/5100-6386-1049862.html

[15] D. Kulak and Eamonn Guiney, *Use Cases: Requirements in Context*, Addison Wesley, 2000.

[16] S. Latchem, "Component Infrastructures: Placing Software Components in Context", George T. Heineman and William T. Councill, Eds., *Component-Based Software Engineering: Putting the Pieces Together*, Chapter 15, Addison-Wesley, 2001.

[17] H. Li, S. Krishnamurthi and K. Fisler. Interfaces for Modular Feature Verification, *International Conference on Automated Software Engineering*, September 2002.

[18] D. Muthig, T. Patzke, Generic Implementation of Product Line Components, NetObjectDays, 2002, net.objectdays.org/node02/de/Conf/publish/papers.html

[19] Sun Microsystems, Designing Enterprise Applications with the J2EE™ Platform, 2nd Edition, http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/DEA2eTOC.html

[20] Sun Microsystems, JavaBeans Specification 1.01, http://java.sun.com/products/javabeans

[21] MSDN, Enterprise Solution Patterns Using Microsoft .NET. http://msdn.microsoft.com/library/en-us/dnpatterns/html/Esp.asp

[22] www.pysol.org

[23] Python Software Foundation, www.python.org

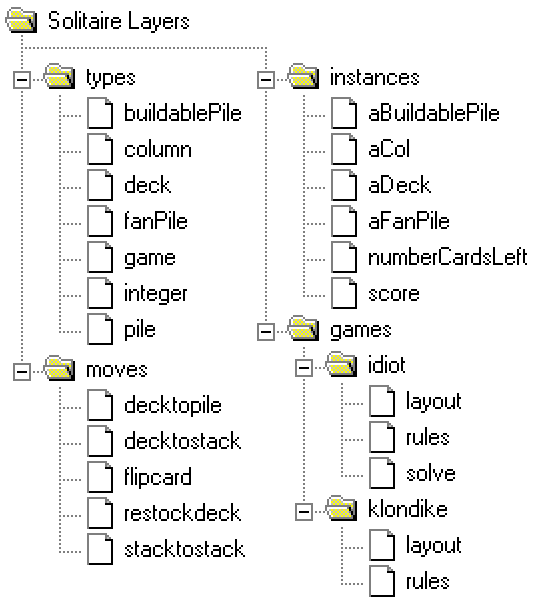
[24] M. Svahnberg and J. Bosch, "Issues Concerning Variability in Software Product Lines", *3rd International Workshop on Software Architectures for Product Families*, Canaria, Spain, LNCS, 2000.

[25] C. Turner, A. Fuggetta, and A. Wolf, "A Conceptual Basis for Feature Engineering", *Journal of Systems and Software*, 49(1), Dec. 1999, pp. 3-15.

[26] J. Wijnstra, "Supporting Diversity with Component Frameworks as Architectural Elements", *Proceedings of the International Conference on Software Engineering* (ICSE), Limerick, Ireland, pp. 50-59, 2000.

[27] Document Object Model (DOM) Level 2 Views Specification, Version 1.0, W3C Recommendation 13 November, 2000, http://www.w3.org/TR/2000/REC-DOM-Level-2-Views-20001113

Appendix A. Solitaire Plugin Layers


Solitaire Layers
types
buildablePile
column
deck
fanPile
game
integer
pile
moves
decktopile
decktostack
flipcard
restockdeck
stacktostack
instances
aBuildablePile
aCol
aDeck
aFanPile
numberCardsLeft
score
games
idiot
layout
rules
solve
klondike
layout
rules