Building Software Agents by Stepwise Feature Introduction—A Case Study

Xiaocong Fan[1], Bixin Li[2], and Ralph-Johan Back[3]

[1] School of Information Sciences and Technology
Penn State University
University Park, PA16802
zfan@ist.psu.edu
[2] Department of Computer and Information Science
Norwegian University of Science and Technology
No-7491 Trondheim, Norway
lbxin@idi.ntnu.no
[3] Turku Centre for Computer Science
Department of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
backr@abo.fi

Abstract. Stepwise feature introduction is an approach to construct software systems by incrementally extending a system with new features. This approach emphasizes reliable software construction in the meaning that each layer is a superposition refinement of its preceding layer, and each evolution version of the implementation is a data refinement of its preceding version or the initial specification. This paper argues for using stepwise feature introduction as a building methodology for reliable cooperative agent systems, and explain how to resolve feature ordering and feature interaction problems between different layers when implementing agent systems by this approach.

1 Introduction

Software agent is a "self-contained program capable of controlling its own decision making and acting, based on its perception of its environment, in pursuit of one or more objectives"[14]. The major differences between (software) agents and objects lie in the following aspects.

First of all, an agent is more like an *active* object. Each iteration of its non-terminating loop involves such steps as context-based reasoning, acting, state (information) revision, etc. Context-based reasoning might be taken as the most distinguished features of agents, which raises issues like hierarchical mental structures, plan priorities, etc. Secondly, the behavior of an agent is context dependent. Resource and time permitted, a rational agent will always carry out context-based reasoning before responding to the external trigger events. Accordingly, an agent might respond differently to the same triggers. Such kind of *context-based responds* is not necessarily enforced to ordinary objects. Thirdly,

an agent is able to act based on its accumulated information. Such *history dependent* feature[19] distinguishes agents from active objects. The next action function of an agent is of form $\mathcal{R} \rightarrow Ac$, where $\mathcal{R}$ is the set of all possible runs; while the next action function of an object is normally $S \rightarrow Ac$, where S is the set of all possible states.

For the first and second issues, active object is supported by most of the modern programming languages, and context-based reasoning can be solved gracefully by using stepwise feature introduction. Besides, there is a theoretical foundation in [5] where an action system (agent) is defined by a recursion equation as:

$$\mathcal{A} \triangleq (recX \cdot S; X \sqcap \langle g \rangle)$$

where the action statement $S = \neg S_1.false \rightarrow S_1 \sqcap \cdots \sqcap \neg S_n.false \rightarrow S_n$ is a choice statement, $S_1, \cdots, S_n$ are conjunctive predicate transformers(methods) given in the specification of $\mathcal{A}$. For the detail of the theory we refer to [5]. Intuitively, the action statement S inside $\mathcal{A}$ is iterated as long as the agent wants. The agent is normally terminated if the *exit condition* g holds when the agent decides to stop the iteration.

The third issue can be dealt with either by introducing stacked attributes, or by encapsulating every such attribute separately into a self-manipulating data object. In the former approach, each mental attribute can be modeled as a stack(list) object with certain type. The top element of a stacked attribute represents the current status of the corresponding mental attribute, while the bottom part keeps record of its history status. In the latter approach, each mental state is actually implemented as a pointer to an appropriate data object (such as dictionary), which maintains related information, and provides appropriate manipulating methods.

Thus, the problem of constructing agent systems can be reduced to the problem of constructing object-oriented systems, by using some encoding mechanisms or design patterns.

In designing and implementing large software systems, layering is an important technique to make the resulting software artifacts easier to construct and understand. Stepwise feature introduction[2], which uses the refinement calculus[5] as its logical basis, results in a kind of layered software architecture. Unlike the traditional software process, which emphasizes strong separation of requirements analysis, design, coding, testing, and maintenance, stepwise feature introduction emphasizes building a new working layer of the expected system over an existing working layer, which results in a collection of software systems with different set of functionality.

Our experiences have shown that stepwise feature introduction can be used to construct object-oriented systems effectively. Since agent systems are much more complex than object-oriented systems, it's more natural to consider different concerns separately when constructing agent systems. In this paper, by a case study, we show stepwise feature introduction can even benefit us more than that in constructing ordinary object-oriented systems. The paper is organized as follows. The next section briefly introduces the basic idea of Stepwise Feature

Introduction and its theoretic cornerstone. Section 3 gives our case study on PEANut, a running E-Bank system. In section 4, we give the layered specification for PEANut, and at the same time, we try to identify the feature ordering problem in Stepwise feature Introduction more formally, and show how to resolve it when implementing agent systems by this approach. The concept of specification space as a lattice structure is also introduced into Stepwise feature Introduction in section 4. Section 5 outlines the basic building steps when a layer is implemented. In section 6, we discuss what stepwise feature introduction approach can bring to us when it is used for building reliable cooperative agent systems. Conclusion and some comparison are given in section 6.

2 Stepwise Feature Introduction

A software system is a coherent society of many components, each of which provides certain services. Feature and layer are the basic concepts in stepwise feature introduction. Informally, a *feature* is a functional aspect of a component (or components) upon which services can be built, while, a service is composed of a collection of features.

By stepwise feature introduction, a project starts from a basic system, which forms the bottom layer of the software architecture. Each layer is built upon its preceding layer by adding exactly one required feature into the existing system. There are three meta-operations that can be used to manipulate the layered architectures: feature introduction, feature combination and re-structuring.

Feature introduction is an activity by which a new feature is introduced as new classes that form a new layer for the system. The new classes are specified as extentions of the classes in the previous layer, so that every such new class will have all the features introduced in the classes it extends. In this step the existing collection of classes is untouched, hence, the previous classes can be preserved for future use, and different features can be separated cleanly. However, this activity must be carried out carefully to guarantee all the old features are preserved after the new feature is introduced.

Feature interaction problems might occur when two seemingly independent features are required to work together (The features might have implicit data flow or control flow relationships). *Feature combination* is a step where the unwanted feature interaction problems are resolved. To make the independent features function in a desired way, a new class extending the corresponding feature classes is introduced in the way that the behavior of each method that is bi-feature sensitive should be overridden properly.

Generally, the order of layers are out of design decisions. Different designers might create different layer structures for the same software system. However, sometimes a feature is forced to be introduced before some others, and sometimes the feature order may later turn out to be suboptimal or just plainly wrong after other features have been incorporated. Accordingly layer restructuring and reordering are essential to keep the system function as desired. When the layered structure runs into problems, specially, when it becomes increasingly difficult to

add new features, usually *Re-structuring* (or [11]) is needed to get rid of the bad design decisions made in the earlier layers. In stepwise feature introduction, however, special attention must be paid to ensure that the layered software architecture in question should not be corrupted after refactoring is applied.

A *layer* can be a single, or a sequence of feature introductions, or a few feature introduction branches together with a feature combination step followed.

While stepwise feature introduction tries to separate different features into different layers, in the constructing process of each layer, however, the different activities (concerns) such as specification, design, coding and testing, are actually encouraged to be considered together in an integrated way. Layered specification, layered coding, and layered testing are interdependent and each of them plays an important role in the correctness concerns that stepwise feature introduction emphasizes.

In the phase of layered coding, every class is annotated with information about how the class is intended to be used. Specifically, each class should have a *class invariant*, which expresses the conditions on attributes that must be established upon instantiation and preserved by any operations of the class. Also, each class method might have a *precondition*, which expresses the assumptions that must hold when the method is called, and a *postcondition*, which states properties that hold when the method call returns.

In the phase of layered verification and testing, there are four correctness conditions that must be checked for each feature layer. 1)Internal consistency. The safety properties expressed as invariants (class invariants, loop invariants) must be satisfied; each method invocation terminates and the corresponding postconditions hold, typing is correct. 2)Inter-respect. A client class must respect the constraints of its server classes, in the sense that the client class has to satisfy the preconditions of all the methods that it calls. 3)Preserving old features. The layer should be proved to be a superposition refinement[4] of its supporting layer. 4)Satisfying requirements. Each feature layer is introduced to meet certain requirements, hence it's necessary to show that such requirements is really realized by this responsible layer.

The correctness conditions can be expressed, and their satisfaction can be reasoned about in the refinement calculus[5]. Superposition refinement mentioned above is a special case of data refinement. While a data refinement step might add new attributes and delete old attributes, in superposition refinement only adding new attributes is permitted. The layers should be built as superposition refinements on top of each other, i.e., at the same time as new features are introduced one has to check that the old features are preserved. This will distribute the total verification effort over the layers in an even way.

In addition to the above correctness conditions, for correctness of an implementation, we also need to show that the implementation is a data refinement of the initial specification. This can be done by using standard data refinement techniques[3, 6], where an abstraction function (relation) mapping the attributes of the implementing class to those of the specification class should be provided.

The main idea behind such reliable software construction process is that: assuming that all features on lower layers do satisfy their corresponding requirements, if we can show that each feature extention satisfies the above correctness conditions, it will follow, by induction, that the software system as a whole will satisfy the correctness conditions.

3 Case Analysis: Personalizing Bank Account Information

We have been using stepwise feature introduction to construct a cooperative information agent system. In PEANut project, we are required to build a Personal Assistant, which is expected to help a user to manage his/her bank account information. More specifically, the resulting software system should be able to mirror the personal account information from Bank servers into local sites(information retrieval), organize the local information in some way which facilitates the manipulation (total, average, sum, etc.) of the data. In addition, it might be able to predict the potential expenditure/income of the coming months based on the history information it currently has, and display such prediction in graphs like pie-percent chart, histogram, Column Graph, etc. We might have the following scenarios, from which desired features originate.

1. Cooperation required: Suppose one agent is only responsible for dealing with the bank accounts opened in one bank. Cooperations occur between the agents taking care of accounts in different banks.

2. Agent splitting might occur: When a user has bank accounts opened in different banks, there will be more than one agent running on his/her computer. These agents have the same security/privacy information about the user, have the same training experiences from the user, and have the same services for the user. From such viewpoint, it might be reasonable to take them as instances of the same (static specified) class. However, this is not enough since as time goes on the agents will not share the same dynamic information, even though they did initially. Thus, any new created agents should be replications of the existing agent, which has the dynamic information of its environment.

3. Mobility required: The assistant should be able to transfer itself to an appropriate host when during data retrieving, it is running on a host that is forbidden to be connected to the Internet, or the network bandwidth is unbearably low; when during data manipulating, its host only has limited computing resources; etc.

4. pro-activity might occur: Usually, there are bills (the rent of an apartment) to be paid periodically. If the agent knows some behavior of its user occurs periodically (paying bills, etc.), it might intend to initiate such behavior automatically, only leave the final decision to its user. In another case, the wife's agent might pro-actively ask the husband's agent some information in order to generate the annual family expenditure/income report.

From these scenarios, we can see that there are lots of complicated concerns to be considered, which have different, interdependent features. If we deal with

all the features together at the same time, different features might crosscut each other, which is hard to understand and re-engineering.

4 Layered Specification

In this section, we present the layered specification of PEANut, and explain how we separate the different features and why the features are ordered into the layers as shown in Fig. 1, rather than the other ordering options.

We first identify the desired features for the agent system to be build.

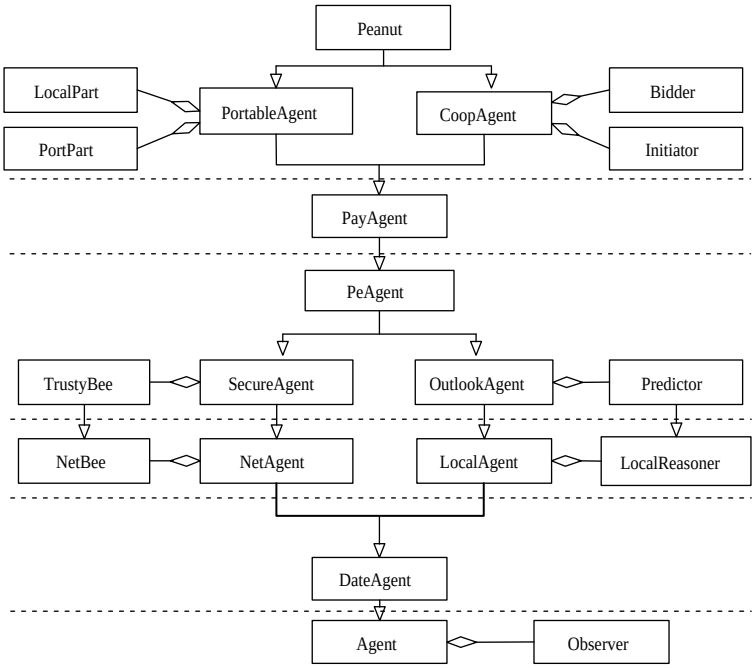


Fig.1 Layered Specification of PEANut

Feature Identification The assistant agent will be situated in a dynamically changing environment, it must be able to interact with its users, learn its users' current interests and behavior patterns from the interactions. We call this feature *observing*, and encapsulate it into class Observer. The agent will be working with volume of data stamped with transaction time and date. Thus, it should be sensitive to date: responds with exactly the relevant information to the date-specific requests from users; and when mirroring data, only the up to date information is retrieved. We identify this feature as *date-sensitive*. Since there is supposed to exist a mirror of the bank account information in local site, the agent can manipulate the local data in whatever way it wants, without considering where the data came from. This feature is called *local-reasoning* and its behavior is encapsulated in class LocalReasoner. On the other hand, the data that our agent will

reason about must be retrieved from bank servers anyway, without concerning about how the data will be used later. We call this feature *net-mining*, for which class NetBee will be responsible.

Since the agent will run on the Internet, especially in such case where the information flow is a money flow, *security* is an indispensable concern. Besides, the agent is able to predict the expenditure in the next fiscal season for its user(*predicting*), to initiate new payment (*auto-paying*), to transfer under certain circumstances (*mobile*), and to proactively participate cooperative-problem solving(*cooperative*). In total, 9 features can be identified, i.e., {*observing, date-sensitive, local-reasoning, net-mining, security, predicting, auto-paying, mobile, cooperative*}. To simplify the following presentation, we will refer them by their initial letters as $\{o, d, l, n, s, p, a, m, c\}$.

Feature Ordering Before ordering the identified features and working out a layered specification for the system, one question must be answered: should we specify-design-code-test a layer, then decide which way we are going and iterate the same procedures for the next layer from bottom up until the expected system has been evolved successfully; or should we first figure out the overall specification for the system, then design, code and test one layer after another along the way outlined in the specification?

Stepwise feature introduction says nothing about which way it prefers. The choice is a trade-off between flexibility and determinism. The former is more flexible, since it supports the design of software families in the means that it provides an easy way to branch off the software process and allows the system to evolve toward different directions, each of which provides a different set of extension features upon a common basic set of features. The latter is more deterministic. Given the overall specification of the system, we are fixing the way the software might evolve, and focusing our concerns only on the way that the system is expected to evolve. In this case study the latter way is chosen, since we care more about whether a system having the above identified features can be implemented (satisfied) in a layered way, rather than obtaining a family of software agents that have different features.

From the correctness point of view, it does not matter how to order the identified set of features, as long as the internal consistency of each layer can be checked easily, and the expected system can be evolved ultimately in a natural way. The layered architecture shown in Fig. 1 is only one possible solution for PEANut. In this architecture, the agent, as a perceiver of the outside world, keeps observing and learning from the interactions with its users. After adding date-sensitive feature, the agent will attach anything it ever manipulated with a time stamp, and it will be able to reason in a temporal way. Then, it evolves into two branches. In one branch, it's equipped with net-mining feature, and further takes security concern into consideration. In the other branch, it's able to analyze its user's account information locally, show information in graphical ways, and further outlook to the near future. Then the two branches are combined, followed by the adding of auto-paying feature. Afterwards, another branching

is introduced, where the two features, mobile and cooperative, are added separately. In the top layer, the branches fuse together and the final specification for the full-featured system is resulted. This layer architecture can be represented by $[o\backslash d\backslash(n\backslash s||l\backslash p)\bullet\backslash a\backslash(m||c)\bullet]$, where $x\backslash y$ means y is layered upon x, $x||y$ means x and y are in paralleled branches, and $\bullet$ stands for feature combination ($\backslash$ binds stronger than $||$).

It's quite interesting to explore the other ways to order the 9 features. For instance, $[o\backslash d\backslash a\backslash(n\backslash s||l\backslash p)\bullet\backslash(m||c)\bullet]$, $[o\backslash d\backslash(n||l)\bullet\backslash s\backslash p\backslash a\backslash(m||c)\bullet]$, $[o\backslash d\backslash(n||l)\bullet\backslash a\backslash s\backslash p\backslash m\backslash c]$, or even linear order like $[o\backslash d\backslash n\backslash l\backslash a\backslash s\backslash p\backslash m\backslash c]$, might also be suitable as the layered specification for our system.

However, either due to the restrictions enforced by the semantics of the problem domains in question, or due to feature dependence relations, some features have to be introduced before some others. For example, the security issue is too trivial to be taken as a separate concern by its own before the *net-mining* feature was introduced. It's more natural to postpone introducing *predicting* until *local-reasoning* feature was already settled down, since prediction can only be carried out by analyzing the accumulated, structured knowledge of the past. *Cooperative* feature cannot be introduced too early (say, immediately after *date-sensitive* feature was introduced), or the resulting agent, being lack of necessary capabilities, won't be able to participate in any cooperations. Thus, to be a feasible solution, an ordering must satisfy some ordering equations. Let $a < b$ mean b is settled in a higher layer than a. Obviously $<$ is transitive. $a \leq b$ iff $a < b$ or $a = b$. For PEANut, at least the following ordering equations must hold:

$$\begin{cases} o < x, x \in \{o, d, l, n, s, p, a, m, c\} \\ n < s \\ l < p \\ n < m \\ x < c, x \in \{o, d, n, l, a\} \end{cases} \tag{1}$$

We take two orderings as *equivalent by refactoring* when one can be transformed to the other, and vice versa, without changing the overall behavior of the system and the branching structures of the original architecture. More formally, we have the following definition.

Definition 1. *Given a set F of feature orderings satisfying the ordering equations in E. For any $m, n \in F$, m is equivalent with n by refactoring iff there exists a sequence of orderings $o_0, o_1, \cdots, o_x$ satisfying E such that $o_0 = m$, $o_x = n$, and $o_i \sim o_{i+1}$ holds for all $0 \leq i < x$, where $\sim$ is given by the following rules:*

$$[f_1\backslash(f_2||f_3)\bullet\backslash s\backslash f_5] \sim [f_1\backslash(f_2\backslash s||f_3\backslash s)\bullet\backslash f_5] \quad \text{(top-branching)}$$
$$[f_1\backslash(s\backslash f_3||s\backslash f_4)\bullet\backslash f_5] \sim [f_1\backslash s\backslash(f_3||f_4)\bullet\backslash f_5] \quad \text{(bot-branching)}$$
$$[f_1\backslash(f_2\backslash s||f_3)\bullet\backslash f_4] \sim [f_1\backslash(f_2||f_3\backslash s)\bullet\backslash f_4] \quad \text{(branch-shifting)}$$
$$[f_1\backslash s\backslash t\backslash f_2] \sim [f_1\backslash t\backslash s\backslash f_2] \quad \text{(line-shifting)}$$
$$[f_1\backslash s\backslash s\backslash f_2] \sim [f_1\backslash s\backslash f_2] \quad \text{(compressing)}$$

where f_i are individual features or slices of some feature orderings, s, t are individual features.

Lemma 1. *Given a set F of feature orderings satisfying the ordering equations E, $o_1 = [f_1 \backslash (f_2 \backslash s \backslash t || f_3 \backslash s \backslash t) \bullet \backslash f_4] \in F$, $o_2 = [f_1 \backslash (f_2 \backslash s || f_3 \backslash t) \bullet \backslash f_4] \in F$, $o_1 \sim o_2$ if $\forall d \in f_2 \cup \{s\} \cdot (d < t) \notin E$ and $\forall d \in f_3 \cup \{t\} \cdot (d < s) \notin E$.*

Proof. $o_1 \sim [f_1 \backslash (f_2 \backslash s || f_3 \backslash s \backslash t \backslash t) \bullet \backslash f_4]$ by (branch-shifting) and the assumption that $\forall d \in f_2 \cup \{s\} \cdot (d < t) \notin E$; The latter equivalent to $[f_1 \backslash (f_2 \backslash s || f_3 \backslash s \backslash t) \bullet \backslash f_4]$ by (compressing); Then it equivalent to $[f_1 \backslash (f_2 \backslash s || f_3 \backslash t \backslash s) \bullet \backslash f_4]$ by (line-shifting), equivalent to $[f_1 \backslash (f_2 \backslash s \backslash s || f_3 \backslash t) \bullet \backslash f_4]$ by (branch-shifting) and assumption, and finally equivalent to o_2 by (compressing).

For example, $[o \backslash d \backslash (n || l) \bullet \backslash s \backslash p \backslash a \backslash (m || c) \bullet]^{(1)}$ and $[o \backslash d \backslash (n || l) \bullet \backslash a \backslash s \backslash p \backslash m \backslash c]^{(2)}$ are both equivalent to the architecture shown in Fig. 1. For (1), $[o \backslash d \backslash (n || l) \bullet \backslash s \backslash p \backslash a \backslash (m || c) \bullet] \sim [o \backslash d \backslash (n \backslash s \backslash p || l \backslash s \backslash p) \bullet \backslash a \backslash (m || c) \bullet] \sim [o \backslash d \backslash (n \backslash s || l \backslash p) \bullet \backslash a \backslash (m || c) \bullet]$ by the above lemma.

Among the set of potential orderings for a specific system, one may be better than the others in some aspects, but worse in other aspects. For example, compared with the ordering shown in Fig.1, the linear ordering $[o \backslash d \backslash n \backslash l \backslash a \backslash s \backslash p \backslash m \backslash c]$ is much easier to control the overall software process, but it loses in the way that it does not support parallel software construction. Moreover, re-structuring might be needed more frequently than using the other orderings as the layer architecture rises higher and higher. However, the evaluation of different orderings is out of the focus of this paper.

Specification Space The specification for a system is the specification for the current feature layer together with all the specifications for the feature layers it extends. The specification for the top feature layer specifies thoroughly the full-featured system, while the specifications for the middle feature layers specify the partial-featured systems that only have a subset of the identified features. After the identified set of features have been ordered, the specifications for the whole and partial systems, derived from the layered structure, form a Lattice structure.

Formally, let S be any layered specification for a system which has a finite set I of identified features. Let D_S be the set of feature combination nodes in S. We call each $g \in D_S$ a *derived* feature to distinguish them from features in I. Let $F = I \cup D_S$, then $\langle F, \leq_S \rangle$ is an ordered set derived from S, where $\leq_S$ is the transitive closure of the feature ordering relation $\leq$ in S. Given an ordered set $\langle F, \leq_S \rangle$, the down-closure of any $f \in F$, denoted by $f\downarrow$, is defined as

$$f\downarrow \triangleq \begin{cases} \{t \in F | t \leq_S f\} & \text{if } f \in I \\ \bigcup\{\downarrow t | t \leq_S f\} & \text{if } f \in D_S. \end{cases} \tag{2}$$

Let $F\downarrow = \{f\downarrow | f \in F\}$. Obviously, $\downarrow: \wp(F) \to \wp(F)$ is a closure operator, and $\langle F\downarrow, \subseteq \rangle$ is a complete lattice, where $\subseteq$ is set inclusion, the meet and join operators are given by $\bigwedge\{A_i \in F\downarrow | i \in J\} = \bigcap_{i\in J} A_i$, $\bigvee\{A_i \in F\downarrow | i \in J\} = (\bigcup_{i\in J} A_i)\downarrow$. For instance, the layered architecture shown in Fig. 1 actually gives us a family of specifications which form a lattice as shown in Fig.2, where an upper specification, by adding new features while preserving old features, is a superposition refinement of the lower specifications.

5 Building Agents Stepwisely

Layered specification of a system actually provides us a roadmap to implement the system. By building agent systems layer by layer, introducing new features one by one, we can get a family of implementations, which has exactly the same layered structure as the specification.

In this section, rather than zooming in to expose the details of how each layer was built, we will only propose the necessary steps that should be followed when a layer is built from specification.

Step 1: Specification-level Refinement Before moving on to implement the system according to the specification of the current layer, firstly we should evolve S_0, the initial specification, stepwisely to get a series of specifications $S_1, \cdots, S_n$, such that S_{i+1} is a data refinement of S_i $(0 \leq i < n)$, and S_{i+1} is a little bit easier than S_i to be transformed into executables in a certain programming language. This is the essence of the "programming from specification"[16].

Step 2: Testing Cases for Requirements The specification of a system consists of the interfaces (signatures) of all the involved classes. An implementation of the specification, however, might do something completely different than what we intended. To ensure the final implementation is what we want, we need to specify the necessary requirements as clearly and completely as possible before programming, so that we can check periodically whether such requirements have already been satisfied during programming.

Every component of the system to be built is expected to have a *tester* class. In each method of such a tester a service requirement for the corresponding

component is embedded, the tester tries to use the service provided by the component in a certain way, and check whether the service it gets matches what it expected.

A service requirement R is an executable statement with the form $[P]; S; \{Q\}$, where P and Q are conditions (predicates) on object (agent) attributes. The *assume statement* $[P]$ will succeed directly, if condition P does not hold, while acts as a *skip* statement otherwise. The *assertion statement* $\{Q\}$ will fail directly if Q does not hold, otherwise it acts as a *skip* statement. R succeeds if P does not hold initially, or if Q holds after executing S successfully.

As an example, the following are methods of *CheckDateAgent*, the tester class for *DateAgent*. Since *DateAgent* will be observed by many listeners, the first method is used to check if the list of observers of DateAgent keeps as it was after appending a new listener and then dropping it away, while the second is used to check whether *SetDate* functions properly. The date attribute of DateAgent should be as it was if the new date accepted by *SetDate* is not valid.

```
def checkObserverPattern_Add_Del(self):
    self.failIf(len(self.proxy.DateListeners) >= MaxListener)
    OListener = self.proxy.DateListeners
    obs = DateObserver(self.proxy)
    self.proxy.DelDateListener(obs)
    self.assertEqual(self.proxy.DateListeners, OListener)

def checkSetDate(self):
    inCorrectInput=['010000','013201','023000','022901',
                    '001202','131200' ]
    obs = DateObserver(self.proxy)
    oldDate=obs.Date
    for date in inCorrectInput:
        self.proxy.setDate(date)
        self.assertEqual(obs.Date, oldDate)
```

Tester classes can be structured by inheritance in the same way as the specification classes. This is not only for preserving the beauty of symmetry, where we have the same structure for specification, documentation, testing, software process (from bottom up), and the resulting family of executables, but also for the reason that, by inheritance, all the requirements specified in a tester class can be automatically inherited by all its subclasses, so that when checking the satisfication of new requirements raised in a higher layer, the requirements checked before in a lower layer are guaranteed to be checked again in the new context.

Step3: Coding-Testing Iteration After all the tester classes for the layer to be built are ready, we enter into an iteration of coding–testing, which terminates until all the requirements modeled in the tester classes are satisfied.

Step 4: Checking the Boundary Refinement Relation The coding artifact after the above iteration, however, is just one of the implementations of

the layer. We cannot claim that the implementation we got, which satisfies all the expected requirements, is exactly the implementation that we are looking for the current layer. We need further to show that the implementation is a data refinement of specification S_n. If not, we have to re-engineer the existing code, or re-start a new coding-testing iteration from scratch.

Step 5: Implementation-level Refinement The implementation C_0 of S_n we get from step 4 can be further refined by a series, $C_1, \cdots, C_m$, of new implementations that are increasingly more efficient, understandable, or extendable, or have more and more functionalities. By monotonicity of data refinement, any usage of C_i in the system can be replaced by using C_{i+1} instead without losing any functionality. Actually, the customer of C_i cannot tell any difference before and after the substitution of C_{i+1} for C_i.

6 What SFI Can Bring to Us

Stepwise feature introduction has the following advantages in design and implement object-oriented systems.

First of all, it conforms to the principle of *separation of concerns* [1, 13, 18], which has been ubiquitously adopted in different areas of computer science, and is hotted up more than ever before by Aspect-oriented programming[15]. Since different features are clearly identified and cleanly separated into different layers, software products are more understandable and maintainable than the same systems built in non-layered ways, and it's much easier to locate and regress to the problematic layer when something bad occurs. Moreover, the feature layers facilitate documenting the overall software construction process by preserving all the design decisions.

Secondly, together with the techniques like unit testing, regression testing, and formal correctness reasoning, stepwise feature introduction provides a feasible way to construct *reliable softwares*. Reliability (closely related to security, credibility) is an essential problem in (multi-)agent systems, especially mobile agent systems, and it must be answered in implementing practical systems. Proof-carrying-code approach[17] is a promising way to protect local systems from attacking by untrusted codes from outside, where checking can be carried out timely due to the encoded proofs bound together with the codes. However, to make the approach work, all the mobile agents must be implemented homogeneously in the means that proofs should be encoded in the same logical language, demon for proof checking should be set up appropriately for each host before expecting mobile codes from outside, etc. While, by stepwise feature introduction, the satisfaction of reliability, which can be expressed as a collection of requirements, can be periodically checked during software construction. This guarantees only reliable agents can be generated, hence checking is not necessary any more.

Thirdly, by stepwise feature introduction, *flexible system evolution* can be achieved to some extent. By flexible we mean two aspects. On the one hand,

the order of two distinct feature layers can be arbitrarily chosen, as long as the chosen order conforms to the specified ordering equations. On the other hand, the current software systems can be extended toward different directions, and thus a family of software, which share a common basic set of features, can be generated.

Specific to building cooperative multi-agent systems, stepwise feature introduction approach supports *hierarchical knowledge representation*. The benefit of representing the mental attitudes of agents hierarchically at least lies in three aspects. First, it facilitates *regression agent splitting*, where all the mental components, except intention, of an agent are divided according to the inheritance source (tag) of each element, which results in a set of clones of the original agent[10].

Secondly, it facilitates *context-based reasoning*. Traditionally, concerns are separated only in the physical-level (specification, design, programming, etc.), while here, concerns are separated deep into the mental-level. Since the mental attitudes are cleanly and naturally separated, the agent can deliberate and respond to external requests based on the sub-mental state most relevant to the requests. Besides, by context-based reasoning, run-time revisions to the irrelevant part of an agent's current mental state will be avoided, consequently, the agent system will be more robust against mindless mistakes. And thirdly, it facilitates agent *team formation*[8]. Joint intentions[9] and SharedPlans[12] are the primary theories about teamwork. However, one of the problems of putting them into practice is how to effectively maintain mutual mental attitudes, as well as plans shared by all the team members. Stepwise feature introduction approach solves this problem by organizing the mutual mental attitudes and shared plans in the layers that are lower than all the agents intending to join in team activities.

In addition, it supports *proactive sub-task commitment*. In traditional contract net protocol, a manager agent announces a (sub-)task to be performed, and bidder agents provide "bids" to execute the task. The (sub-)task is eventually assigned to the bidder the manager agent finds most desirable. One drawback of this approach is that the whole high-level task will end up with failure in case that some bidder agent failed in carrying out its sub-tasks. Building agent systems by stepwise feature introduction, we can easily embed "failure-and-try" mechanism. For example, suppose in the current environment there are 4 active agents, n, s, o, p, which are instances of $NetAgent, SecureAgent, OutlookAgent$, and $PayAgent$ respectively, and we have a high-level task "pay a bill, retrieve new transaction information from the bank server, and then generate prediction report". This high-level task can be divided into 3 sub-tasks, $pay, retrieve$, and $predict$, which might be discharged to p, n and o respectively. Now, in case that n failed for some reason (network congestion, etc.), all the active agents who are at a higher level than n (which can be checked on-the-fly by querying the meta-level information through some language-specific mechanisms(Java, Python, etc.). More formally, the candidate agents must have the features of $NetAgent$ as the *least lower bound* in the lattice induced by the layered spec-

ification architecture) can proactively commit to the sub-task *retrieve*, if they have the high-level task as their common persistent goal. Hence, s and p will commit to *retrieve* to avoid abortion of the high-level task.

7 Comparison and Conclusion

Compared with the other incremental approaches and the other layered agent design methodologies, stepwise feature introduction emphasizes reliable software construction in the meaning that each layer is guaranteed to be a superposition refinement of its preceding layer, and each evolution version of the implementation is guaranteed to be a data refinement of its preceding version or the initial specification.

Moreover, since in stepwise feature introduction each layer is a working system, this makes the agent system be capable of representing knowledge hierarchically and responding based on the most relevant context. However, not all the other layered designs inherently share such advantages.

In Stratified architecture[1] approach, the layers, ordered by interaction refinement, incrementally reveal more concerns by making explicit some of the implicit abstractions contained in the preceding layers. While, in Stepwise feature introduction, there is no implicit abstractions slumbering in a layer that need to be uncovered later. Each layer exists in a system only because it introduces a new feature based on another layer (layers), and each (implementation) layer together with the layers below is a full-fledged executable.

The contribution of this paper is not in reducing the problems of constructing agent systems to the problems of constructing object-oriented systems, nor in the construction methodology itself, but in exploring and trying to show that stepwise feature introduction as a construction methodology, can be applied successfully in building reliable cooperative agent systems. In addition, as an extention to the original Stepwise feature Introduction approach, we try to formally identify the feature ordering problem, and show how to resolve it when implementing systems.

References

1. Atkinson, C., Khne, T.: Separation of Concerns through Stratified Architectures. International Workshop on Aspects and Dimensional Computing at ECOOP, 2000
2. Back, R.J.: Software Construction by Stepwise Feature Introduction. To appear in Proceedings of the ZB2001-Second International Z and B Conference, Springer Verlag LNCS Series, 2002.
3. Back, R.J., Mikhajlova, A., and von Wright, J.: Class Refinement as Semantics of Correct Object Substitutability. Formal Aspects of Computing, 12:18-40, 2000.
4. Back, R.J., Sere, K.: Superposition refinement of reactive systems. Formal aspects of computing, 8(3):324-346, 1996.
5. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag, 1998.

6. Back, R.J., von Wright, J.: Encoding, decoding, and data refinement. Formal Aspects of Computing, 2000.
7. Beck, K.: Extreme Programming Explained. Addison-Wesley, the XP Series, 1999.
8. Cohen, P. R., Levesque, H. J.: Teamwork. Nous 25(4), Special Issue on Cognitive Science and Artifical Intelligence, pp. 487-512, 1991.
9. Cohen, P.R., and Levesque, H.J.: Intention is Choice with Commitment. *Artificial Intelligence*, 42, pp. 213-261, 1990.
10. Fan, X.: On Splitting and Cloning Agents. TUCS technical report, No 407, May, 2001 (to appear in AAMAS'2002).
11. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.
12. Grosz, B., Kraus, S.: Collaborative plans for complex group actions. *Artificial Intelligence*, 86,269-358,1996.
13. Hürsch, W., Lopes, C.V.: Separation of Concerns. Northeastern University, technical report, no. NU-CCS-95-03, 1995
14. Jennings, N.R., Wooldridge, M.: Software Agents. IEE Review 42(1), pp 17-21. 1996.
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented Programming. In: 11th Europeen Conference on Object-Oriented Programming, LNCS, vol. 1241, pp. 220-242, Springer Verlag, 1997.
16. Morgan, C.C.: Programming from Specification. Prentice-Hall, 1990.
17. Necula G. C., Lee P.: Safe, Untrusted Agents using Proof-Carrying Code. October 1997, in LNCS 1419: Special Issue on Mobile Agent Security.
18. Smaragdakis, Y., Batory, D.: Implementing Layered Designs with Mixin Layers. In: Proc. of ECOOP, pages 550-570, 1998.
19. Wooldridge, M., Dunne, P. E.: Optimistic and Disjunctive Agent Design Problems. In Y. Lesperanceand C. Castelfranchi, editors, Intelligent Agents VII Springer-Verlag Lecture Notes in AI Volume, February 2001.