Implementation of Programming Languages

– Praktikum WS08/09 –

Software Technology Group,
Fachbereich Informatik, TU-Darmstadt.

(Version 1.6)

Anthony Anjorin

Supervised by Vaidas Gasiunas

9th May 2009

Contents

Introduction

Context

The goal of the AMPLE[1] project is to identify and develop methodologies to help cope with the challenges that arise when enabling and supporting SPL[2]s.

In this practicum, new concepts and extensions to existing OO[3]-technologies are to be implemented in the Caesar-J [1] compiler, using the JastAdd [2] framework.

Aim

The aim of this document is threefold: It serves as a reference for the actual implementation, as information and results gleaned from discussions with my supervisor are carefully recorded. In this way nothing important can be lost or "forgotten".

Documenting planned tasks and related complications and expressing solution concepts in my own words, provides a simple means for my supervisor to validate the respective points and avoid misunderstandings and an erroneous implementation.

Last but not least, this document serves as an informal documentation of not only what was implemented but also why and reasons for making certain decisions and simplifications. This complements comments in code and will be extremely helpful for anyone trying to get an overview of the implementation. The section "implementation status" in every chapter consists of entries each corresponding to approximately 6 hours of work. This should be helpful when planning a new undertaking.

Scope

This document is neither a complete documentation of parts of the Caesar-J compiler nor an exact or formal description of implemented features.

There is furthermore no guarantee that discussed solutions correspond exactly to what is finally implemented.

Structure

Each chapter treats a certain feature to be implemented in the Caesar-J compiler. The last chapter concludes with a case study to showcase the new features and demonstrate advantages.

A glossary on page 21 defines important terms and acronyms used in the document while the bibliography on page 22 lists related references.

[1] Aspect-Oriented, Model-Driven, Product Line Engineering
[2] Software Product Line
[3] Object-oriented

1 Contracts between features

Families of virtual classes enable feature-oriented programming, whereby each sub-family can provide a further-binding for certain virtual classes and thus implement a single feature. This leads to a decomposition according to features – enabling a flexible means of combining and composing features to realise a *complete* implementation. Numerous sub-families, each implementing a certain feature are combined via multiple-inheritance and mixin-composition.

1.1 The problem

A sub-family can be a provider of a certain feature, a client depending on other sub-families that provide certain features, or both. How can contracts for these different roles be expressed? In OO languages the keyword `abstract` can be used to convey that a class is *incomplete* and thus can not be instantiated.

Using `abstract` alone is however insufficient, as one is unable to differentiate between sub-families that *fulfil their contracts* but are still incomplete as they only implement a single feature, and sub-families that*break their contracts* and are thus incomplete – even w.r.t. the feature they promise to implement. For huge projects with various teams implementing different features, this differentiation is of vital importance.

This problem doesn't occur in OO languages as a similar decomposition in features would have to be realised via multiple interfaces and a multi-level inheritance hierarchy. This solution however, doesn't provide the same ease of composition necessary for supporting SPLs. A solution using composition would be similarly awkward and result in a myriad of interfaces and uses-associations. A further disadvantage of composition is not being able to overwrite certain aspects of an aggregated provider.

1.2 The solution

In the following, a *normal class* is a cclass, that neither contains any classes nor is itself contained in any class.

A *virtual class* is a class that is contained in another class. Virtual classes are different from inner classes as they can have further bindings and can thus be overridden and changed via multiple-inheritance and mixin-composition.

A *family* is a class containing 1 or more virtual classes. A family can itself be a virtual class contained in another family.

A *top-level family* is a family, which is itself not contained in any other family. In other words, a top-level family is at the top of the virtual class include hierarchy. According to this definition a top-level family is not a virtual class.

1.2.1 What does *abstract* and *concrete* mean?

A normal class is declared as being *abstract* when it is not to be instantiated. This is specified with the keyword `abstract`. Abstract normal classes do not have to have abstract methods – instantiating such a class might be forbidden for various semantic reasons. Normal classes with abstract methods on the other hand, must be declared as abstract.

A normal class is regarded as being *concrete* when it is not declared as abstract. Concrete normal classes can be instantiated.

A virtual class is declared as being *abstract* when it is never to be instantiated. Abstract virtual classes do not have to contain abstract methods. Since families can however be used to create member virtual classes via polymorphism, it would be wrong to enforce that virtual classes containing abstract methods have to be themselves abstract. Instead it suffices to demand that for virtual classes with abstract methods, at least one family in the include hierarchy be abstract. This prevents any unsafe instantiation.

A virtual class is therefore *concrete* when it is not abstract. This means concrete virtual classes can contain abstract methods as long as at least one family in the include hierarchy is abstract.

Families that are themselves virtual classes have the same semantic for being *abstract* and *concrete* as virtual classes.

A top-level family however must be *abstract* and thus can not be instantiated, if it contains any abstract members – virtual classes or methods – either directly or in member virtual classes.

A *concrete* top-level family is not abstract and can be instantiated.

In summary *abstract* means "can not be instantiated" while *concrete* means "can be instantiated".

1.2.2 What changes with contracts?

In the following, the keywords `abstract` and `requires` as well as the terms *complete*, *incomplete* and *contract* are defined and used to differentiate between client-contracts and provider-contracts.

Complete should mean "can be instantiated".

Incomplete should mean not complete and thus "can not be instantiated".

A *contract* of a class constitutes all inherited abstract members that must be implemented for the class to be complete.

Abstract should mean "does not have to fulfil its contract" or "does not have a contract". Abstract classes are declared as such using the keyword `abstract` and are per definition incomplete.

Concrete and thus not abstract should mean "fulfils its contract". Concrete classes do not have to be complete.

A class can act as a client by *requiring* a part of its contract. Abstract methods inherited via a `requires` as opposed to an `extends` do not need to be implemented for a class to be concrete.

A class can act as a provider by inheriting a part of its contract via `extends`. These methods must be implemented for the class to be concrete.

A normal class is complete when it is concrete and has no requirements.

A top-level family is complete when it is concrete and has no requirements.

Presently, requirements are only allowed for top-level families and of course normal classes. Virtual classes and families that are virtual classes *cannot* have requirements and are thus complete when they are concrete (same as without introducing `requires` keyword).

In the future, allowing requirements for virtual classes shall be considered.

Inheritance relationships of superclasses are propagated as follows:

- If A `extends` B and B `extends` C, then A `extends` C. B's provider-contract is defined by C. B can fulfil this contract completely or partially and even extend it by new abstract methods. As A fulfils the provider-contract specified by B it obviously fulfils B's contract as well.
- If A `extends` B and B `requires` C, then A `requires` C. This means A fulfils the provider-contract specified by B. B's client-contract does not have to be fulfilled and thus all requirements are inherited.
- If A `requires` B and B `requires` C, then A `requires` C. B's client-contract is inherited...
- If A `requires` B and B `extends` C, then A `requires` C. B fulfils the provider-contract defined by C. However, since B specifies A's client-contract, A doesn't provide an implementation and thus requires B which of course entails requiring at least C and possibly more.

1.2.3 Design decisions and fine details

The following decisions were made while implementing. Reasons and arguments are briefly presented and alternative solutions discussed.

Redundant requirements:

A requirement is considered redundant if it is already part of the *direct* provider contract of a class. This is the case is when a class requires and extends the same class:

```
Class A extends B requires B { ... }
```

Although senseless, redundant requirements are allowed and are simply treated as requirements that happen to be fulfilled. This means abstract methods inherited by a concrete class via `extends` are *always* to be implemented, even if the same methods happen to be required as well.

An improvement would be to issue a warning when a requirement is redundant.

Implementing requirements directly:

Implementing required methods might seem contradictory but actually makes sense: A class may wish to perform certain tasks before and after calling the required core functionality supplied by another class in a complete product. This is semantically similar to an around advice of an aspect. Unfortunately, enabling this with the current implementation of mixin-composition is problematic: Calls to `super` would result in requests to the required abstract contract classes and not to the appropriate providers.

Implementing this feature is therefore not a trivial task and is for the time being postponed – a direct implementation of requirements is currently disallowed to avoid potential problems.

Required methods in virtual classes:

Although only top-level families or normal classes can have requirements, the required methods can of course actually be present in enclosed virtual classes (family members).

Determining if a method is required or not can be a little tricky and involves the so called ETL[4]: To decide if a certain method of an arbitrarily nested virtual class is required, the enclosing top-level family is determined. All enclosed methods in the required classes of the ETL, either as direct members or as member methods of an arbitrarily nested family member, are then compared with the current method. A match means the method is required and doesn't need to be implemented!

Definition of completeness:

The check for completeness is based on the following definition:

[4]Enclosing Top Level

- All concrete (not declared as abstract) classes implement all inherited abstract methods and methods from implemented interfaces.
- $[provides]$ and $[requires]$ are not symmetric as cyclic inheritance is disallowed.
- All types are either interfaces, normal classes or top-level families.
- $\langle\ \rangle$ is used to indicate terminal relationships while $[\ ]$ implies a recursive, possibly transitive relationship.
- A binary relationship rel_B is expressed as $A\ rel_B\ B$, a unary relationship rel_U as $A\ rel_U$. E.g. $A\ uses\ B$ and $A\ complete$
- Expressions of the form $(A\ rel_B\ B)$ or $(A\ rel_U)$ can be negated and used in any logical expression: $\frac{\forall A\ :\ \neg(A\ rel_U)}{A\ rel_B\ B}$
- "..." $\in P$ means the code snippet ... is present in the program P
- In a code snippet, [ ] has the usual BNF meaning of "optional".

The following define $[extends]$:

$$\frac{\text{“}[abstract]\ class\ A\ extends\ B\text{”} \in P}{A\ \langle extends\rangle\ B} \tag{1}$$

$$\frac{\text{“}interface\ A\ extends\ B\text{”} \in P}{A\ \langle extends\rangle\ B} \tag{2}$$

$$\frac{A\ \langle extends\rangle\ B}{A\ [extends]\ C} \tag{3}$$

$$\frac{A\ \langle extends\rangle\ B, B\ [extends]\ C}{A\ [extends]\ C} \tag{4}$$

The following defines $\langle implements\rangle$:

$$\frac{\text{“}[abstract]\ class\ A\ implements\ B\text{”} \in P}{A\ \langle implements\rangle\ B} \tag{5}$$

The following define $[provides]$:

$$\frac{\neg(A\ abstract), class\ A\ \langle extends\rangle\ B}{A\ [provides]\ B} \tag{6}$$

$$\frac{\neg(A\ abstract), A\ \langle implements\rangle\ B}{A\ [provides]\ B} \tag{7}$$

$$\frac{\neg(A\ abstract), A\ \langle implements\rangle\ B, B\ [extends]\ C}{A\ [provides]\ C} \tag{8}$$

$$\frac{A\ \langle extends\rangle\ B, B\ [provides]\ C}{A\ [provides]\ C} \tag{9}$$

The following defines $\langle requires\rangle$:

$$\frac{\text{“}[abstract]\ class\ A\ requires\ B\text{”} \in P}{A\ \langle requires\rangle\ B} \tag{10}$$

The following defines $[subtypeof]$:

$$\frac{A\ \langle extends\rangle\ B}{A\ [subtypeof]\ B} \tag{11}$$

$$\frac{A\ \langle requires\rangle\ B}{A\ [subtypeof]\ B} \tag{12}$$

$$\frac{A\ \langle implements\rangle\ B}{A\ [subtypeof]\ B} \tag{13}$$

$$\frac{A\ \langle extends\rangle\ B, B\ [subtypeof]\ C}{A\ [subtypeof]\ C} \tag{14}$$

$$\frac{A\ \langle requires\rangle\ B, B\ [subtypeof]\ C}{A\ [subtypeof]\ C} \tag{15}$$

$$\frac{A\ \langle implements\rangle\ B, B\ [subtypeof]\ C}{A\ [subtypeof]\ C} \tag{16}$$

The following define $[requires]$:

$$\frac{A\ \langle requires\rangle\ B}{A\ [requires]\ B} \tag{17}$$

$$\frac{A\ \langle requires\rangle\ B, B\ [subtypeof]\ C}{A\ [requires]\ C} \tag{18}$$

$$\frac{A\ \langle extends\rangle\ B, B\ [requires]\ C}{A\ [requires]\ C} \tag{19}$$

The following defines *complete*

$$\frac{\neg(A\ abstract), \forall B : A\ [requires]\ B \Rightarrow A\ provides\ B}{A\ complete} \tag{20}$$

1.3 Implementation status

16.09.08 The first step is to extend the parser to recognise `requires` as a keyword. This should serve as a small task to get used to the testing framework and the general syntax and format of .jrag and .ast files.

22.09.08 Implemented parsing of `requires`.

23.09.08 The next step is to extend or overwrite the methods that determine the complete list of superclasses - these should include all required classes. When this is working and has been tested, existing type-checks have to be extended to implement the extra semantics of requirements. A further step is then to implement the transformation to valid java and if necessary byte-code.

24.09.08 Implemented adding required classes to list of superclasses. The next step is now to extend or change checks in `ASTNode::CollectErrors(...)`. Changes should be test-driven and take possible rewrites into consideration.

A tentative plan:

1. Change checks to allow concrete but incomplete classes (successful compilation).
2. Bytecode of incomplete classes should nonetheless contain abstract flag (necessary for valid bytecode).
3. Check to disallow requirements for virtual classes.
4. Check that only complete classes are instantiated with appropriate and comprehensible error message.
5. Propagation of requires relationship from super to subclasses.

30.09.08 Concrete but incomplete classes now compile. Essentially removed required methods from the list of unimplemented methods. Required methods do not need to be implemented and should thus not be regarded as being unimplemented.

01.10.08 Concrete, incomplete classes now retain their abstract flag in byte-code. This is unfortunately not testable as an automated junit test, since the class-loader doesn't mind loading "invalid" byte-code containing classes with abstract methods but without a corresponding abstract flag. Therefore, manual tests were carried out to ensure that classes with requirements are now abstract classes in byte-code.

02.10.08 Instantiation of incomplete classes (classes with requirements) is now disallowed.

03.10.08 Requirements for virtual classes are now disallowed.

04.10.08 Requirements are now propagated. This has to be tested further...

12.10.08 Improved implementation of completeness check. Defined completeness formally. Implemented handling of redundant requirements. Forbid direct implementation of requirements.

14.10.08 Some regression tests are failing and have to be investigated. Allowing java interfaces as requirements proves to be quite challenging – have to decide if this is worth the extra complexity.

03.11.08 Made a list of all tests that were failing prior to contracts and saved this as a test-report in project. Can now use this report to evaluate regression test-suite. Refactored solution to avoid dependency on execution sequence. A brief plan for the next few weeks:

1. Efficiency of `unimplementedMethods()` and generalisation of `getRequirements()` for virtual classes...
2. Extra tests for contracts

14.11.08 Completed `unimplementedMethods()`

1.4 Knowledge base

- List of requirements implemented to be after extends and implements.
- Number of combinations for instantiation in parser grows exponentially with new keywords! Adding a further keyword might require refactoring and a solution to cope with this.
- Syntactic tests are not possible with current test framework - compilation always fails instead of checking if a specific error was expected. Test-framework would have to be extended to support this.
- Possible tests are: `compile`, `compile-run` with tests in a `test` block and `compile-check-error` with a `Test` class.
- Logging can be enabled in `test.properties`
- Synthesised attributes are to be specified in all subclasses – this enables an upward flow of information in the AST[5]. This is comparable to overriding methods in OO.
- Inherited attributes are to be specified in parent-nodes and refined in children-nodes – this enables a downward flow of information in the AST. This *inheritance* is however structural and has nothing to do with normal OO-subtyping.
- It is possible to add new non-terminals in the AST and define these in .jrag files. Typical mistakes are however forgetting to define all inherited attributes and not giving careful thought to the implementation of the corresponding getter and setter method (e.g. not overwriting in getter when setter has been called).

[5]Abstract Syntax Tree

2 Private Inheritance

Inheritance is an important means provided by most OO-languages of defining the relationship between a base-class and it's sub-classes. Inheritance enables not only code-reuse but also allows for polymorphic treatment of sub-classes.

2.1 The Fragile Base Class Problem

A problem with inheritance is that seemingly safe changes to a base-class can have detrimental ripple effects on it's sub-classes. This is because all sub-classes "inherit" and thus depend on implementations of methods and behaviour in the base-class [3]. In a large system, determining if a change to a base-class is safe or not, requires considering *all* sub-classes – a difficult, non-modular and non-scalable task.

2.2 The Solution

Introducing a new key-word `uses` enables so called *private inheritance* where a sub-class inherits as normal from a base-class, but doesn't pass on the inherited behaviour to its sub-classes. This is conceptually identical to *using* the base-class via composition, but without the hassle of having to explicitly call methods and to implement trivial methods that only delegate requests.

Base-classes only used to implement internal details should thus be inherited via a `uses`-relationship to avoid unwanted dependencies deeper down in the class-hierarchy. Sub-classes are hence oblivious of all `used` classes higher up in the class hierarchy.

2.2.1 Design decisions and details

The following design decisions for usage semantics were made:

Propagation of behaviour:
Methods and behaviour are propagated to direct users but not to classes deeper down in the class-hierarchy.

Subtyping:
Subtyping to `used` classes is generally not possible, even for direct users.

Requirements:
Requirements cannot be *hidden* and thus must not be propagated via `uses`. A direct user has to re-declare, fulfil or inherit (via normal means) all requirements of the classes it uses.

This way, sub-classes of direct users remain oblivious to the existence of all `used` classes – also with respect to requirements.

`Used` classes cannot fulfil requirements implicitly as this would lead to sub-classes of users being dependent on exactly these `used` classes with respect to their completeness. This kind of transitive dependency is what should be avoided via `uses`!

If indeed a certain base class is to be `used` to fulfil specific requirements, then this should be explicitly stated as part of the provider contract of the user:

```
Class A requires X {}

Class C extends X {}

Class B extends A & X uses C {}

Class D extends B {}
```

In the above example, `B` inherits the requirement of `X` from `A`. Although `B` uses `C` (to fulfil its requirement of `X`), it still has to explicitly extend `X`. `D` can then safely assume that `X` is provided by `B` and doesn't care if this is done directly or by using some class.

Contracts:
Analog to requirements, abstract methods inherited via `uses` cannot be hidden and must be either implemented, inherited via an `extends` relationship or re-declared in the direct user.

Virtual Classes:
As the motivation for `uses` was to prevent unnecessary dependencies between *features*, the `uses` relationship is to be considered as a top-level relationship between features. As a consequence, `uses` is to be disallowed for virtual classes and only allowed for top-level families[6]. In the future this restriction might be removed.

Bytecode:
As valid bytecode must be produced, all usage semantics must be removed and replaced by normal inheritance via `extends`.

Transitivity of usage:
The following defines $\langle uses\rangle$:

$$\frac{\text{“}[abstract]\ class\ A\ uses\ B\text{”} \in P}{A\ \langle uses\rangle\ B} \tag{21}$$

The following defines $[uses]$:

$$\frac{A\ \langle uses\rangle\ B,\ \ B\ [requires]\ |\ [extends]\ C}{A\ [uses]\ C} \tag{22}$$

[6]Exactly what is meant by these terms was defined in 4.2

2.3 Implementation status

22.12.08 Completed specification, documentation and tests for private inheritance.

29.12.08
1. Introduced new keyword *uses* in parser definition.
2. Added used classes to list of parents so that they are treated in a first step exactly as extended classes.
3. Disallowed uses for virtual classes.

08.01.09
1. Abstract member classes, abstract methods and requirements cannot be inherited via uses.
2. Prevented access to members from indirect users.
3. Contracts should not be provided via uses.

12.01.09
1. Prevented subtyping to used classes.
2. Generalised usage semantics for virtual classes.

2.4 Knowledge base

- Adding `uses` as a new keyword to the parser would require specifying 64 possible combinations! A different solution is clearly needed and a possible idea would be to define "optionals" for each token that can either be present or not.
- This has been implemented for `uses` and adding further keywords is now simplified.

3 Case Study: Sales scenario

A simplified version of a typical sales scenario as specified in [4] is to be realised using the new features introduced and implemented in 1 and 2.

The aim of this case study is twofold:

- As a means of testing the extended compiler with a realistic system. This should complement unit testing and reveal new bugs and problems.
- To showcase the new features and show advantages using a concrete example.

3.1 Design decisions and advantages of using new features

Based on [4] and existing implementations in CaesarJ, the system depicted in fig 1 was implemented and is described in the following section.

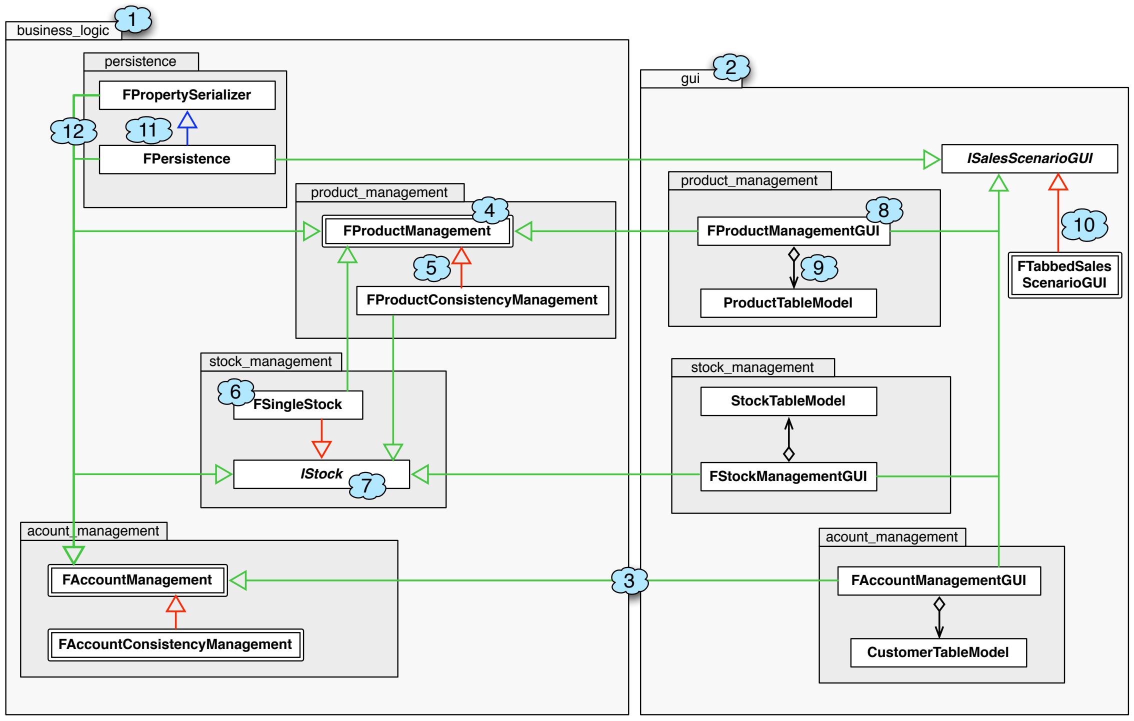

Figure 1: A simple sales scenario

Concerning (1): The system was clearly divided into two layers: The business logic layer and a user interface layer. This clear division allows for mixing features independently to provide interaction not only via graphical widgets but also possibly via a web browser or a command line. `business_logic` provides four groups of features: Account management (managing a list of customers and delivery addresses), product management (managing a list of products the company offers), stock management (managing a list of which products are in stock and what quantity is available) and persistence (saving and loading accounts, products and the stock across sessions).

Concerning (2): A graphical user interface layer was provided, consisting of features mostly corresponding to features present in the business layer. Persistence for example however, doesn't require interaction with the user and is thus not mirrored in `gui`.

Concerning (3): Requirements are represented by green, normal extension (providers) by red and usage by blue inheritance arrows. Features in `gui` that correspond to features in `business_logic` naturally *require* these features. An interesting exception is the feature `FPersistence` that needs to know when the user wishes to terminate the program.

Concerning (4): In fig 1, *complete* features are depicted with a double lined border. These features have no requirements and can be instantiated on their own. `FProductManagement` for instance, can be used on its own even without a GUI. Most GUI features however make no sense on their own and have to be paired with required business logic features to create a complete product.

Concerning (5): `FProductConsistencyManagement` is an incomplete feature that showcases the advantage of using contracts. It provides `FProductManagement` and extends it with product consistency. To do this it needs to update and hold the stock of products consistent. Without requirements, the semantic difference between the roles of `FProductConsistencyManagement` as a provider and as a client can only be discerned by actually viewing the implementation in code. Moreover, it would have to be declared as abstract since we do not wish to implement abstract methods in `IStock`. This is disadvantageous since the compiler can no more enforce implementation of abstract methods possibly present in the provider contract (`FProductManagement`).

Using requirements however, reasons for an incomplete product can be clearly differentiated by the compiler: Unfulfilled provider contracts – meaning some provider has broken its contract – or unfulfilled requirements – meaning the current mix of features does not define a complete product.

Concerning (6): A similar example as in (5) with the roles of provider and client contract switched. The compiler can now ensure that `FSingleStock` *provides* `IStock` whereas abstract methods (if any) in `FProductManagement` need not be implemented.

Contracts enable a completeness check that ensures that no incomplete mix of features is instantiated and gives detailed reasons for incompleteness. In a complex system with a myriad of features, this clear semantic difference can be very crucial.

Concerning (7): `IStock` is a clear abstract contract for a stock of products. This means that different implementations of such a stock are already envisioned and planned for the system. Compare this to `FProductManagement` that also functions as a contract but is itself complete and not a pure abstract specification. One has to make a trade-off between flexibility (defining an abstract contract for product management and letting `FProductManagement` provide this) and complexity.

Concerning (8): `FProductManagementGUI` allows the user to view and manage the inventory of products in form of a series of workflows. To achieve this, it requires a main window in which it can register and integrate itself and of course a feature that implements the business functionality offered in its workflows. `FProductManagement` or any of its providers would satisfy this requirement.

Concerning (9): `FProductManagementGUI` displays the inventory of products in form of a table. The table widget uses a model – `ProductTableModel` – to refresh its contents. Although `Product-TableModel` is conceptually a virtual class in `FProductManagementGUI`, this is technically not possible as it has to extend a Java class (a cclass cannot extend a normal Java class).

Concerning (10): `ISalesScenarioGUI` specifies some functionality that every main window must provide. Most details are however, like how sub-windows are to be arranged, are left open. `FTabbedSalesS-cenarioGUI` provides a simple implementation of a main window, adding sub-windows as tabs and interacting with the user via swing dialogues.

Concerning (11): `FPersistence` *uses* `FPropertySerializer` to save and load lists of customers, products and stock items. The choice of private inheritance here means that we consider this functionality to be an implementation detail and we want to avoid dependency in any means. We could of course define a clear abstraction and let `FPersistence` require and `FPropertySerializer` provide this – we however don't want this added complexity at the moment and don't plan to implement many different "serializers". Without private inheritance we would have had to either use composition (if the class preexisted) or simply implemented the functionality directly in `FPersistence` as private methods. This is however either cumbersome (explicit delegation) or clutters up `FPersistence`. *Uses* therefore provides a better trade-off between complexity and flexibility.

Concerning (12): An interesting point when considering private inheritance and features, is that abstract methods and requirements cannot be inherited privately and must be implemented or redeclared. In the case of `FPersistence`, all requirements are redeclared.

3.2 Implementation status

15.03.09

1. Released present version of compiler
2. Started testing with case study "SalesScenario"
3. Found and fixed first bugs (transitive usage, constructors and requirements)

22.03.09

1. Found and fixed further bugs (definition of "provided" for virtual classes)
2. Completed implementation of case-study
3. Completed class diagram and documentation

3.3 Knowledge base

- How to set up new version of CaesarJ compiler:
  1. Produce compiler and runtime jars in `caesar2` by changing ant targets (from `gen` to jars).
  2. Replace compiler and runtime CaesarJ jars in `org.caesarj.ui`
  3. Launch a new eclipse workspace with `org.caesarj.ui` as plug-in
- The CaesarJ-plugin seems to have a problem with deleted files. Deleted files are resurrected and appear in the package explorer although they clearly don't exist anymore. Restarting eclipse with `-clean` solves the problem.
- Java's serialization API doesn't seem to work well with CaesarJ. Writing out objects (families and virtual classes) works, but reading in objects fails as normal constructors are expected.

4 Mixin Methods

Mixin composition uses a linearisation algorithm to allow for multiple inheritance - a necessary feature for FOP[7]. The linearisation that occurs is however not always unambiguous and sometimes *the order of superclasses* has to be used to resolve such problems:

```
cclass A {
  void doSth(){}
}

cclass B {
  void doSth(){}
}

cclass C extends B & A {}
```

The class C inherits the method `doSth()` from `B` only because `B` appears before `A` in the list of superclasses.

4.1 The Problem

Although this is often acceptable, it can be dangerous and unexpected, especially when many features are involved and one is not aware of, or cannot avoid redundant functionality.

In the example above, A might be a complex feature that relies on its own implementation of `doSth()` and does not expect it to be overwritten. If `B` is however also a complex feature with numerous methods, a user might not be aware that `doSth()` is provided. In this case, overriding `A`'s version of `doSth()` based solely on the order of superclasses would be undesirable and wrong.

4.2 The Solution

In order to avoid unexpected and possibly wrong overriding of methods, one can require feature providers to annotate methods that can be *mixed in* without problems and methods that should *not* be overridden implicitly via the order of superclasses. When a method not annotated as a mixin would be overridden based on the order of superclasses the compiler should issue an error:

```
cclass A {
  void doSth(){} // Base version

  void doSthElse(){} // Base version
}

cclass B1 {
  void doSth(){} // Not a mixin
}

cclass B2 {
  mixin void doSthElse(){ // A mixin method
    ...
```

[7]Feature Oriented Programming

```
    super.doSthElse(); // Mixins usually call base version

    ...
  }
}

cclass C extends B1 & A {} // An error, trying to override doSth implicitly

cclass C extends B2 & A {} // Ok, doSthElse in B2 is a mixin method
```

Mixin methods are methods that are to be viewed as being composable with a base version in a superclass. This usually means some compatible functionality is added before and/or after a super call to the base version.

4.2.1 Design decisions and details

The following design decision were made:

Overriding methods implicitly:
Overriding methods based on the order of subclasses is an error.

Demand a single base version for every method:
Every method must have a single base version. A method is a base version when it is not annotated as being a mixin method.

Demand mixins before base:
Mixin versions of a method are to appear only before the single base version.

The above decisions lead to the following definitions:

$$M_A := \text{All methods of A} \tag{23}$$
$$L_A := \text{All local methods of A (defined in Body of A).} \tag{24}$$
$$S_A := M_A \backslash L_A \tag{25}$$

$$\frac{\forall m \in S_A : m \ \langle mixinValid_A \rangle}{A \ \langle mixinValid \rangle} \tag{26}$$

$$\frac{\begin{array}{c}\exists! \ B_m : B_m \ \langle providesBase_A \rangle \ m, \ \forall B \in \{sortedParents(A) : m \in M_B\} : \\ (B == B_m) \vee (B_m \ \langle superClassOf \rangle \ B) \vee (B \ \langle providesMixin \rangle \ m)\end{array}}{m \ \langle mixinValid_A \rangle} \tag{27}$$

$$\frac{B_m \text{ is rightmost in } \{B \in sortedParents(A) : m \in M_B\}, \neg(B_m \ \langle providesMixin \rangle \ m)}{B_m \ \langle providesBase_A \rangle \ m} \tag{28}$$

$$\frac{m \in M_B, \text{“}mixin\text{”} \in signature(m)}{B \ \langle providesMixin \rangle \ m} \tag{29}$$

Parameterised super call, issues related to super call:
???

Consequences for virtual classes, contracts, private inheritance etc:
???

4.3 Implementation status

4.4 Knowledge base

Glossary

AMPLE	Aspect-Oriented, Model-Driven, Product Line Engineering
SPL	Software Product Line
OO	Object-oriented
ETL	Enclosing Top Level
AST	Abstract Syntax Tree
FOP	Feature Oriented Programming

References

[1] I. Aracic, V. Gasiunas, M. Mezini and K.Ostermann:
Overview of CaesarJ

[2] G. Hedin, E. Magnusson:
The JastAdd system - an aspect-oriented compiler construction system,
Science of Computer Programming 47 (2003) 37-58, Elsevier.

[3] Leonid Mikhajlov, Emil Sekerinski:
A Study of The Fragile Base Class Problem.

[4] Carsten Luxig, Intesio GmbH, Henrik Lochmann, SAP AG, SAP Research, Elena Martel, pure systems GmbH:
Definition and description of a realistic example suite,
feasiPLe project D 1.1 (2006) Pages 12-20

Version History

Version	Date	Changes
0.1	15.09.2008	Created document.
1.0	22.09.2008	Reduced scope of requires to only top-level families.
1.1	23.09.2008	Added next steps for implementation.
1.2	24.09.2008	Implemented determining list of superclasses, added concrete plan for next steps.
1.3	30.09.2008	Implemented concrete but incomplete classes.
1.4	15.10.2008	Added formal definition of completeness.
1.5	1.12.2008	Completed (specification) for private inheritance
1.6	23.03.2009	Completed case study