Pattern Language for Framework Construction

Shai Ben-Yehuda

email: shai@sela.co.il

http://www.sela.co.il/~shai

Abstract

One of the most effective form of reuse in OO technology is the framework. The pattern language introduced in the paper tries to capture some design and architectural patterns that reoccurs while constructing OO frameworks. The pattern language is based on a different perspective, regarding frameworks as a tool for developers.

A Framework As A Product For Developers

A framework is mainly a ***product for developers*** *used as an integrated development environment that facilitates, supports, guides, confines, and helps the developers in building application in a well defined domain*. A framework is an environment built to reduce development costs, maintenance costs, and implementation costs.
An OO framework is a product for developers who specially use OO constructs and methods like abstractions and specialization, interfaces, contracts, classes, and objects. OO frameworks can support both implementations and design reuse.

Other definitions of frameworks are:

- "A framework is a reusable design for an application or a part of an application that is represented by a set of abstract classes and the way these classes collaborate " (Johnson 88)
- According to Coplien & Schmidt [PLOP I] a framework can be characterized by the followings:
  1. A framework provides an integrated set of domain specific functionality.
  2. Frameworks exhibit an "inversion of control" at run-time.
  3. A framework is a "semi-complete" application.
- Pree prefers to distinguish between framework and application framework: "The framework is a collection of abstract and concrete class and the interface between them and is the design for a subsystem ... and the term application framework is used if this set of abstract and concrete classes comprises a generic software system for an application domain." [Pree]

The wider perspective of frameworks allows the author to enlarge its scope. Hence, new kinds of elements may be considered as parts of the framework; for instance, utilities and building blocks.

In the paper the framework user will be referred as both *developer* and *framework user* to distinguish him/her from the *framework constructor*.

Intent

Generally, the pattern language suggests a top down approach to framework construction. Means, it is tuned to be used to build a framework from scratch or while re-designing an existing framework. The pattern language may be used in any phase of the framework development. The patterns classified as *architectural* are patterns that affect the whole framework while the *design* patterns have only a local effect.

General Context

The field of OO framework engineering has not matured yet. We have tasted the effectiveness of frameworks like MacApp, MFC, ET++, InterViews, JDK, ACE and more that may speed-up the development cycle significantly. Using these frameworks we can produce even better products compared to conventional tailored made OO development because we get hard to implement features for free. E.g. portability, error handling, etc.

However, many attempts to build successful frameworks have failed. Most of the failures occurred because of lack of some framework qualities. However, framework qualities contradict each other in many situations. The framework constructor should balance between these qualities trying to achieve the best mixture in his/her environment. We may regard the framework qualities as the conflicting forces drive the design of framework construction. The major qualities/forces are listed here:

- **Effectiveness**- How effective is the framework for the developer (the framework user)? We can measure framework effectiveness by the speed-up factor which weigh development cost with the framework against the development cost without the framework. Effectiveness means also faster time to market of the products produced by the framework.
- **Extensibility**- How rigid is the framework architecture? In some cases it is almost impossible to add a feature in a given architecture; for instance, a triangle window in a windowing framework. It is important to state that any architecture impose constraints, when trying to evaluate framework extensiveness we should try to look at specific domains that are likely to be covered by the framework. In many cases, extensibility and effectiveness conflicts each other.
- **Coverage -** How wide is the application space in which the framework is effective? A framework may be extensible but have a narrow coverage.
- **Simplicity and Understandability**- What is the learning curve of a typical developer? The framework concepts may be too abstract or the architecture may be too complicated to grasp.
- **Framework Integration**- Can the framework be integrated with other frameworks or libraries?
- **Application Qualities- Efficiency, Portability, Adhering to standards-** Do the applications made by the framework have some non-functional qualities? Although important, this aspect will be disregarded in the pattern language because it affects architectural issues more than framework issues. The author chooses to ignore this issues in order to emphasize the “tool for developer” aspect of frameworks over the architectural aspects.

The patterns listed in the paper suggest some known compromises between these conflicting forces. Some of the patterns tend to be more risky (non extensive) and effective, and some are safer and less powerful. The starts attached to each pattern reflect author confidence in the pattern.

The Patterns in a nutshell

Category	Pattern	Problem	Solution
Structural, architectural	1) Conceptual Layering	Lack of understandability and problems with framework Integration.	Divide framework elements into two layers: product level and building block level.
Coupling, architectural	2) Collaborating Product Concepts	The framework is not effective. Even with the framework it takes a long time to build an application.	Define abstract behavior for the framework that defines the collaboration between the product concepts
Coupling, architectural	3) Independent Building blocks	General framework components in the building blocks level can be used only within the framework. We would like to use them in other frameworks to enhance code reuse and promote framework integration.	Make the building blocks independent. E.g. string, date, matrix, point and math class
Structural, architectural	4) Multi-level Framework	The framework becomes too complex because the developer wants to build new building blocks.	Create a new framework that produces building blocks and let each framework to develop by its own forces.
Structural, architectural	5) Nested Framework	We need both a general purpose abstract framework with a wide coverage and an effective framework in a more specialized field.	Build a general framework with abstract product concepts and general building blocks, and build the specialized framework inside it.
Design	6) Developer Contracts	The developer does not have a clear contract with the framework when specializing a product concept.	Define half baked implementation classes that adhere to the pre-defined collaborating behavior and externalize a useful interface (contract) to the developer.
Design	7) Multiple Developer Contracts	The developer is confined to only one contract when specializing a product concept.	Define some half baked implementation classes, each of them should adhere to the pre-defined collaborating behavior and externalize a different interface (contract) to the developer.
Design	8) Framework Utilities	Classes alone do not deliver the speed up factor.	Add utilities to the framework to enhance developer effectiveness. Utilities are products which are given to the developer but will not be installed or used in the end user site.

(1) *** Conceptual Layering
Product Concepts and Building Blocks

Problem

The developer does not have a clear understanding about the kinds of products he/she can produce using the framework, and the means the framework gives him/her to produce them.

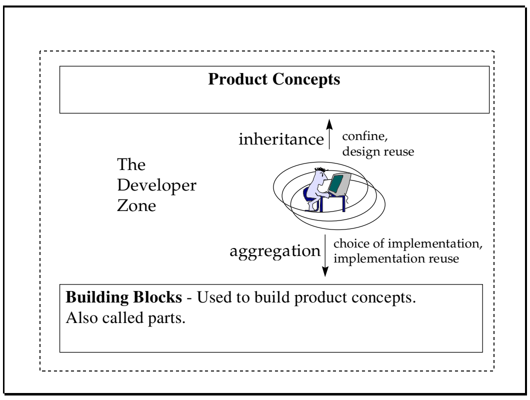
Forces

- **Understandability-** Products and means to create them should be clear to the developer.
- **Effectiveness versus Simplicity-** Uncoupling between framework elements may improve simplicity but reduce effectiveness.
- **Extensibility-** New kinds of products should be easily supported by the framework.
- **Framework integration-** We should look for common elements between frameworks.

Solution

Divide Framework elements (classes, in most cases) into two layers:

- **Product level-** High level concepts that define products that may be created using the framework. These products are known to the developer and he/she has chosen the framework in order to produce them.
  E.g: Document is a product level concept in a MFC and ET++. Shape and surface are product level concepts in a CAD framework.
  The developer (framework user) uses the framework to specialize pre-defined product level concepts.
- **Building block (part) level-** Concepts/classes that may be used as parts in the construction of the product level concepts. These concepts conceptually resides in a lower level than the product level. The framework user assembles building blocks to constructs product level concepts.
  E.g: Edit box, list box and buttons building blocks in graphical framework are used to build dialog boxes products.
  Line, arc and matrix building blocks in CAD framework are used to build shape products.



According to the regular definition of a framework, building blocks may be regarded as a supporting class library. The wider definition of framework allows the author to consider building blocks as part of the framework.

Examples

In MFC we use containers, strings, fonts, threads, and controls in the building blocks level to build documents and views at the product level.

In Java we use the general building blocks in java.util classes like vector, dictionary, calendar, random, math, etc. to build applets in the product domain.

In the Xtoolkit framework the product concepts are widgets and the Xlib primitives are used as building blocks and.

The ACE framework helps the developer to build his/her own Active Objects (product layer) using reactors, mutexes, connector classes, and acceptors in the building blocks layer.

Consequences

+ Simplicity- The developer knows what to expect from a class according to its layer. The developer role is clearer, she/he should define a product class by using building blocks.

+ Framework integration- frameworks can be integrated more easily if they share the same building blocks. JDK is a good example for that. The java.util building blocks are shared between java-beans framework and the AWT framework. See *Nested framework* for more.

+ Extensibility- Separating the product concepts gives the framework constructor a clearer view of what may be developed using the framework. It gives the developer a language to define the desired extensible axes.

- Effectiveness- Separating between these layers may reduce framework effectiveness because user-defined products can not be used as building blocks. Furthermore, if standard building blocks are used they can not contribute to the framework specific design. See the *Independent Building Blocks Pattern* for more.

- Framework Coverage- Implementing the pattern means that the framework specializes in product concepts construction. However, what happens when the developer wants to define or customize a building block; for example, the developer wants to define another container in Java. Adhering to this pattern, we must claim that the developer should work in another framework in which the former building blocks are the product concepts. See *Multi-level framework* for more.

Related/Advanced issues

Building blocks by the developer choice- In many cases, building blocks are used by choice and the product level concepts are forced .

The Composite pattern and working modes- The Composite pattern is a good example for abstracting the product and the building block together putting them in the same layer. E.g. Control is an abstraction of dialog box (product) and button (building block). However, working consciously in two modes may both preserve the conceptual layers while using the composite pattern. The abstraction should be ignored while working on a product class. But when reusing the product class as a building block we should use the abstraction and put the product in the building blocks section.

Building Block as parameters in the product classes interface- In many cases, we use the building block classes to define the interface of product classes. E.g. using string and date as method parameters. We must remember that it forces the developer to use the defined building blocks. It breaks an important principle that the framework developer should have the ability to choose the appropriate building block.

When to implement? In many cases, we use the pattern in the first stages of the development because of its simplicity. Then we may break the pattern to enlarge framework coverage and effectiveness. Implementing the pattern in later phases of framework development happens if we find a better layered model or during understandability or framework integration crises.

(2) *** Collaborating Product Concepts
Conceptual Dependency

Intent

Determining dependency between framework elements is a key issue when trying to balance between the conflicting forces, especially between effectiveness and the other forces.

The pattern tries to deal with specific dependencies between the product concepts. framework level collaborations may support design reuse and significantly affect the speed-up factor and framework effectiveness. However, any collaboration creates a dependency and affect understandability, extensibility, framework integration, etc.

Problem

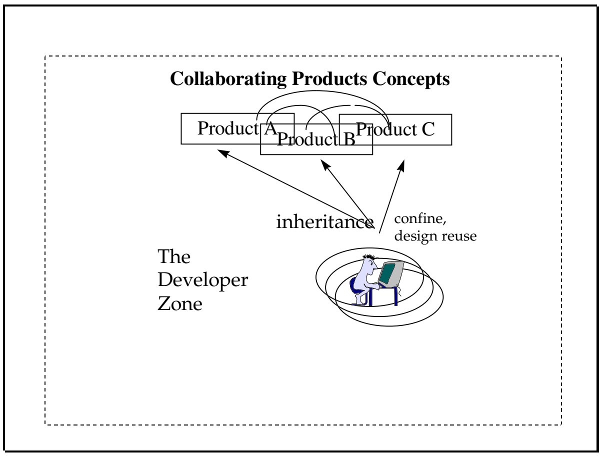
The framework does not deliver the expected development speed up factor.

Forces

Effectiveness versus Extensibility- To make to framework more effective me must add behavioral coupling, doing that we loose extensibility and coverage.

Solution

Make the framework more effective and less extensible. Make the product concepts more coupled. Define abstract behavior for the framework that defines the collaboration between the product concepts.



Implementation

Define *Template Methods* [GoF] at the base product classes to define the collaborating behavior. The *Mediator Pattern* [GoF] may be used to define the collaborating behavior in external objects or functions that controls registered product objects.

Examples

In MFC the collaboration between the product concepts *document* and *view, frame* and *document template* is hidden from the framework developer, especially the creation process and the notification process. This collaboration defined by the framework allows the user to define document and views without caring for these problematic design issues.

In Java AWT the message passing mechanism is hidden and determined by the framework. The layout mechanism is hidden to some extent as well.

Consequences

+ Effectiveness- The framework becomes more effective because the developer should define only the differences between his/her needs and the framework behavior.

+ Simplicity - The developer does not know about the collaboration mechanisms. Hence, he/she has a simpler model to work with. However, when the developer is not satisfied with the abstract behavior, she struggles against the framework to change it, and forced to understand most of its internals

- Framework integration- The framework products become more coupled therefore less reusable in other frameworks.

- Extensibility- If we regard the product collaboration as an architectural decision, we won't be able to use the framework in the cases that the collaboration is defined differently.

- Coverage - If we do not regard the product collaboration as an architectural decision. The framework is effective in a smaller domain. The framework covers only applications that behaves similarly to the framework. The framework does not cover application that share the same concepts but behaves differently. See *Nested Framework Pattern* fore more.

Related/Advanced issues

Product classes interfaces- This pattern causes the product interfaces to be fat and unclear because they reflect the collaboration as well. E.g. CWnd in MFC.

Developer Contracts- In order to hide the complex interfaces, the framework constructor adds Template Methods [GoF] with primitive operations [GoF] or uses the Strategy pattern. The developers overrides only the primitive operation [GoF] or add a strategy to customize the product. The framework becomes more Hollywoodic: "Don't call us. We'll call you."See the *multiple developer contracts pattern* for more.

Nested framework- If the collaboration does affect the products interfaces, the *Nested framework Pattern* can be used. The pattern supports the abstract level as the upper framework and the collaborating framework as a deeper framework. See *Nested framework pattern* for more.

(3) ** Independent Building Blocks

Intent

Modularity is a crucial factor for reusablity and clarity. Class independence reflects modularity. In many cases, we can achieve this modularity at the building blocks level with a reduced effort.

Problem

The frameworks building blocks are not reusable in other frameworks.

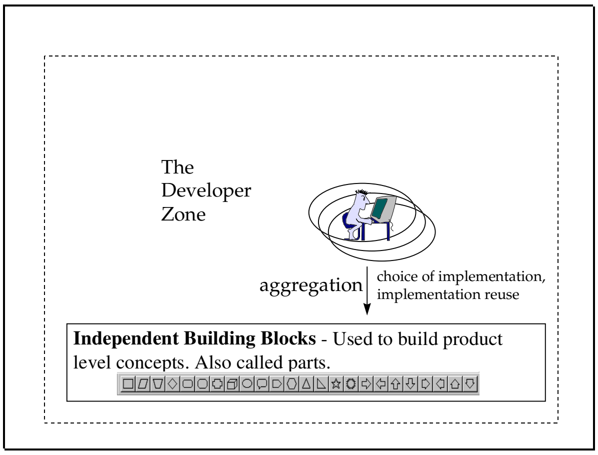
Forces

- **Ownership versus framework elements reuse-** framework constructors prefer to own all framework components in order to have control on framework evolution. However, some components may be used in some frameworks if built to serve this purpose.

- **framework integration and understandability versus effectiveness-** framework constructors prefer to reuse standard building blocks but to promote framework effectiveness they should customize their building blocks.

Solution

Trade effectiveness for understandability, code reuse, and framework integration. Make the building blocks independent. Make sure the building classes do not know about the other framework classes. E.g. string, date, matrix, point and math class.



Examples

java.util classes are independent.

In ACE mutexes are independent.

Anti example- In MFC we see that most of the general building blocks are not independent. *List* is dependent upon the *object* in MFC and even *string* is dependent upon the *archive* class in MFC for persistency.

Consequences

- Effectiveness- The building block is not smart enough to be integrated seamlessly within the framework. E.g: the string objects do not serialize themselves automatically and the framework user should remember to store them every time she uses them.

+ Simplicity - The building blocks are simple and have no side effects. It is easier to understand, use, and debug them.

+ Code reuse inside the framework- The independent building block classes are reusable.

+ framework integration- The building blocks can be shared between frameworks because the building blocks are not dependent upon specific framework elements. See the *Nested framework pattern*.

Related/Advanced issues

Building blocks layering- Building blocks layers are often defined by the framework constructor. In many cases, we separate between the general building blocks layer and the framework specific layer. The separation promotes framework integration. E.g.: java.util as the general layer and java.awt.Font at the AWT framework layer. It may be considered as a pattern for itself. See the *Nested framework pattern* for more.

Virtually Independent building blocks- Button is a building block in Java AWT framework. However it depends upon the *component* class from which it inherits. AWT designers would like the Button to have no side-effects within the framework and appear like an independent component for the developer. Virtual dependency has nothing to do with virtual method invocation mechanism. It may be considered as a pattern for itself. There is a great risk in virtual dependency because the framework deceives its users.

(4) ** Multi-Level Framework

Context

The *Conceptual Layering Pattern* was implemented. The developers need to refine building blocks.

Problem

The *Conceptual Layering Pattern* does not let the developer define/refine building blocks.

Forces

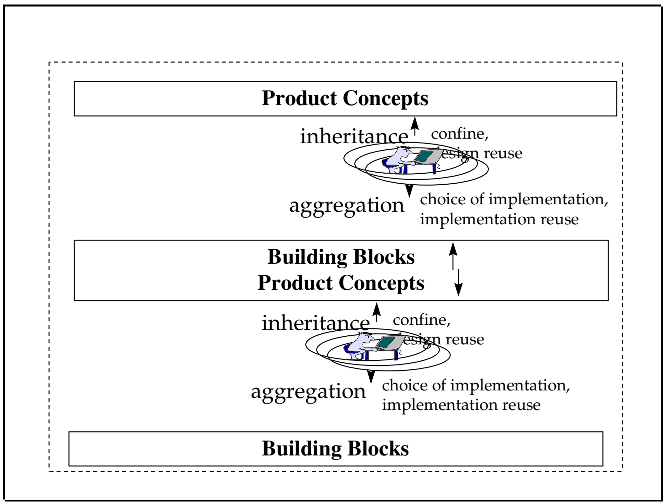
Effectiveness versus simplicity- The framework may be more effective if the developer can add his/her own building blocks. However, if the framework supports the development of both products and building blocks we may end up with a complex framework.

Framework construction costs- The framework construction costs should be paid back by reducing development costs. If we do not see that clearly we may end up with “White elephant frameworks”.

Solution

Create a separated framework that produces some of the building blocks for the existing framework and let it develop by its own forces.

The product domain framework and the building blocks framework may have completely different building blocks, conceptual dependencies, utilities and developer contracts. In most cases, the building-blocks framework (lower level) has no conceptual collaboration.



Examples

LayoutManager in Java AWT - Layout manager is considered to be a building block when constructing applets. However, we may need a framework to define customized layout managers.

Acceptor in ACE- Acceptor is a building block when creating an active object. But to create customized acceptor we need a framework for dealing with network protocols, advertising, screening, etc.

Consequences

+ Effectiveness- The overall frameworks become more effective because each framework can specialize in its own field.

+/- Simplicity - Each framework is simpler but the whole system may be more complicated.

+ Framework integration- The building blocks framework may be used to support other frameworks that uses the same building blocks. See *Nested Framework Pattern* for more.

- Framework construction costs not paid back

Framework coverage- No effect.

(5) *** Nested Framework

Intent

The abstraction level of the framework depends upon the coverage and the effectiveness we expect from the framework. Frequently, we can identify a general framework with abstract products that consists of more specialized frameworks. The specialized framework extends the abstract framework.

Building frameworks recursively promotes framework integration and framework coverage with no effectiveness payoff.

Context

We have a general purpose abstract framework with a wide coverage. The *Conceptual Layering Pattern* was implemented and the *Collaborating Product Concepts Pattern* was not implemented. An effective specialized framework is needed. The specialized framework products are sub-concepts of the base framework products.

Problem

How to implement an effective specialized framework?

Or, we would like some siblings frameworks to share common infrastructure that may be used as a framework.

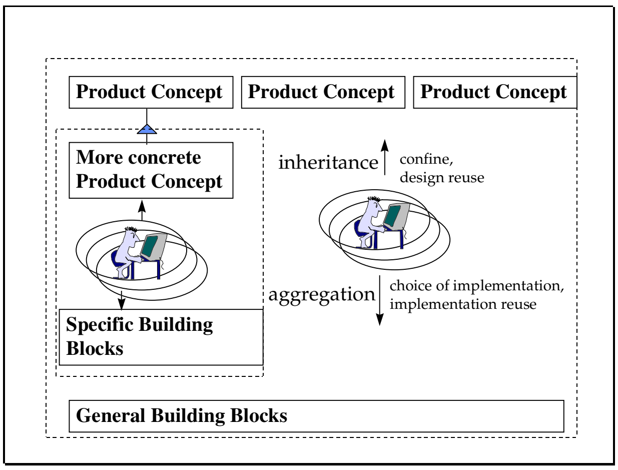
Forces

Effectiveness versus simplicity- The framework may be more effective if the developer can work in both the abstract level and the concrete level. However, if the framework supports the development of both products and building blocks we may end up with a complex framework.

Framework construction costs- The framework construction costs should be paid back by reducing development costs. If we do not see that clearly we may end up with "White elephant frameworks".

Solution

Build a general framework with abstract product concepts and general building blocks and build the specialized framework inside it. The specialized framework product concepts derived from the general framework and the general framework building blocks may be used in the specialized framework.



Examples

Motif is a nested framework inside the Xtoolkit framework. They share the Xlib building blocks and the abstraction of widget is defined at the toolkit level and refined in the Motif framework.

MFC- We can identify general persistency and RTTI framework that defines high level product concepts like *object* and *archive*. The general framework building blocks are containers and strings. Inside the general framework there is a windowing framework that deals with document and views which are kinds of objects and use the persistency and RTTI framework.

Consequences

+ META Effectiveness- The framework code is reusable by the framework constructor, hence, the framework constructor becomes more effective.

+/- Simplicity - More frameworks but each framework is simpler.

+ Promoting framework integration- Some concrete frameworks may use the same general framework and may be integrated together.

- Framework construction costs

Effectiveness- No effect.

Framework coverage- No effect.

Related/Advanced issues

Some frameworks in the same field- Some frameworks or some versions of the same framework may live together within the same general purpose framework. However, product objects of the specialized frameworks will not be able to collaborate with each other.

(6) ** Developer Contract

Context

The *Collaborating Product Concepts Pattern* was implemented. The developer must implement primitive operations [see template method in GoF] or override some methods. The framework constructor is not sure about the primitive operations which are useful to the user.

Problem

The developer does not have clear interfaces to work with. The framework parameters of variation should be well defined.

Forces

Understandability versus coverage- Clear developer contracts may promote understandability but being overly precise may confine the developer, hence reducing framework coverage.

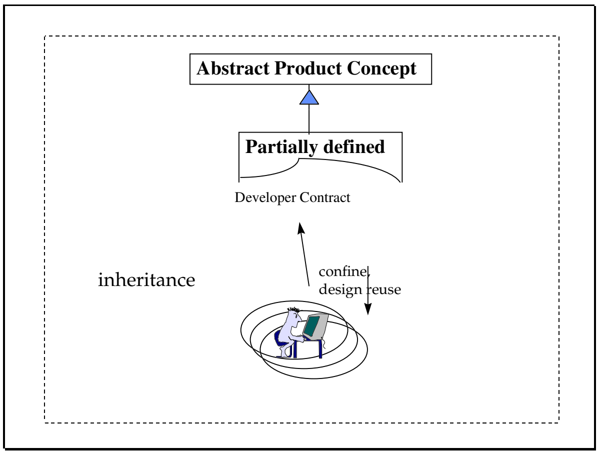
Framework construction costs- We would like to avoid the "white elephant" syndrome. The framework should pay back its construction costs.

Effectiveness- The developer contract should reflect the developer's needs and not the open part of the concept collaboration.

Solution

Analyze users and developer needs. Define partial (half baked) implementation classes which adhere to the pre-defined collaborating behavior and externalize a useful interface to the developer. The developer should inherit from these class.

The developer interface can be defined by primitive operations using the template method pattern [GoF] or using the strategy pattern [GoF].



Examples

MFC Views- To define a view, the developer can use CFormView and he/she should only add a dialog box the defines the form.

Java Image Filter (java.awt.image) - To define an image filter the developer can use the RGB image filter and he/she should only define the color translation table.

Consequences

+ Effectiveness- The framework becomes more effective because the contracts are aimed to help the developer.

+ Simplicity - The developer contracts are well defined. Hence, the developer role becomes clearer.

- Framework construction costs- More effort is put in the framework construction phase. In many cases we would prefer iterative definition of developer contracts.

Framework integration- No real effect.

Framework coverage- No real effect.

Related/Advanced issues

White box versus black box contracts [Johnson]- The developer contract may be well defined when using the strategy pattern or documenting the primitive operation by defining their responsibility clearly. But when the framework constructor is not sure about the contract, he/she can use the white box approach, letting the developer to explore the base class and override open method opens. Johnson says that the white box approach should be used in early stages of framework construction, while we have no idea about the needed developer contracts. However, white box contract means in most cases that the developer should be aware of the collaboration between the product classes.

Declarative developer contracts and profiles- In addition to primitive operations and strategy pattern, the developer contract can be define as a set of open parameters, and the developer may declaratively assign values to the open parameters. A set of values is called a profile. The developer can use profiles to define sub-concepts in the product domain. Profiles can inherit from each other and aggregate each other. For more details about profiles see [The TGP methodology].

(7) *** Multiple Developer Contracts

Context

The *Developer Contract Pattern* was implemented.

Problem

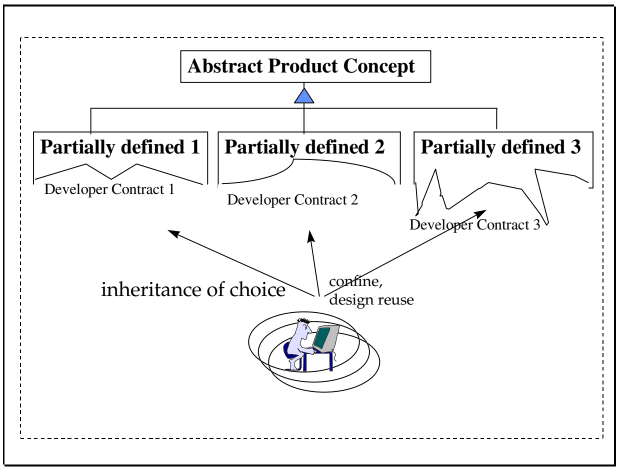
The developer has only one contract to specialize a product concept. He/She does not have the ability to choose an adequate contract when defining his/her product concept.

Forces

Effectiveness versus framework construction costs- Many developer contracts may promote effectiveness but may lead to a fat framework that does not pay itself back. We would like to avoid the "white elephant" syndrome.

Solution

Define the product domain interface as a base class (preferably only the interface). Define **some** partially defined (half baked) implementation classes, each of them should adhere to the pre-defined collaborating behavior and externalize a different interface to the developer. See the *Developer Contract Pattern* for contract definition details.



Examples

MFC Views- To define a view, the developer can choose between CFormView or CCTRLView according to his/her needs. The developer contract is defined using the constructor and overridables methods (primitive operations) in each base class.

Java Image Filter (java.awt.image) - To define an image filter, the developer may choose to override RGB image filter or Replicate image filter. Both classes defines the developer contract with primitive operations.

Consequences

+ Effectiveness- The framework become more effective because the developer can choose the most suitable contract. The framework can give both simple contracts for frequent needs and complicated, powerful contracts. The pattern enables the framework constructor to enhance framework effectiveness with no pay on simplicity or coverage.

- **Extensibility-** In the partial classes implementation we adhere and depend upon the abstract product collaboration. Doing that we make it harder to change the product collaboration.

+ Simplicity - The developer contracts are well defined. Hence, the developer role becomes clearer. The framework can support incremental learning by introducing more complicated developers contracts when needed.

- Simplicity- If the developer contracts are similar in content, it may confuse the developer and she/he will not know what contract to use.

- Framework construction costs- More effort is put in the framework construction phase. Half baked classes are expensive in terms of development costs. The framework constructor should be careful not to build non-useful contracts.

Framework integration & Framework coverage- No real effect.

(8) ** Framework Utilities

Intent

A framework is not only the set of classes. To be complete, most frameworks should have utilities like specialized code generator, debugger, test and help.

Problem

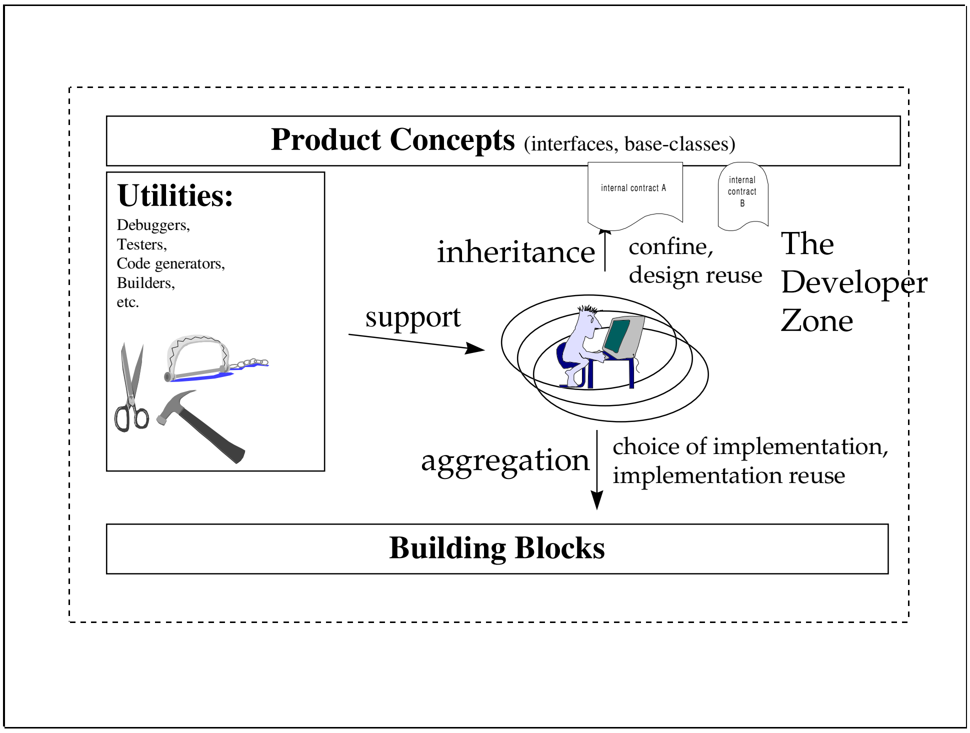
Classes alone do not deliver the speed up factor. The developer needs a fully supported environment that knows the framework.

Forces

Effectiveness versus "white elephant" framework- Utilities may promote effectiveness but if we put too much effort in them we may put a lot of effort in never-used utilities.

Solution

Add utilities to the framework that certainly enhance developer effectiveness. Utilities are products which are given to the developer but will not be installed or used in the end user site.



Examples

The classical utilities are:

- Code generator
- Debugger
- Sensitive Editors

- Browser
- Testers
- Integrated Help system
- Integrated Tutorial and cookbooks

In MFC (Visual C++) there are the following utilities:

- application wizard as framework sensitive code generator and cookbooks.
- Spy as a specialized windows application debugger.
- Class editor sensitive to framework overridables. Dialog Box editor.
- Test framework.
- framework tutorial.

Consequences

+ Effectiveness- Code generators, debuggers and editors are built to promote developer effectiveness with no real side effects.

+ Simplicity - The learning curve become less steep with integrated help & tutorials. Code generators can also help to improve understandability.

- Coverage in practice - Putting many tools in the developer environment may cause him/her to treat the framework differently. Culturally, they may reject direct use of classes and hence it will reduce framework coverage in practice.

- **Framework development costs**

framework integration- No effect.

Acknowledgments

Many thanks to James O. Coplien and Paul Dyson who shepherd me in EuroPLOP 97. Special thanks to my wife Ayelet who gave me inspiration from other fields.

References

[1] GoF - Design Patterns. GHJV. Addison Wesley 95.
[2] POSA - Pattern Oriented Software Architecture. BMRSS. Wiley 96.
[3] Pattern Languages of Program Design I,II. Addison Wesley 1995,96.
[4] Reusability through self encapsulation. K. Auer. PLOP1 95. Pg. 505.
[5] Design patterns for OO software development. W. Pree. ACM 95.
[6] Object Oriented Application Frameworks. Ted Lewis, editor. 95
[7] Software Architecture. M. Shaw. D. Garlan. PH 96.
[8] MFC Internals. G. Shepherd. S. Wingo. Addison Wesley 96.
[9] Tricks of the Java Programming Gurus. G. Vanderburg. SAMS NET 96.
[10] The TGP Methodology. S. Ben-Yehuda. http:\\www.sela.co.il\~shai\tgp.ps
[11] Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. Ralph Johnson, Don Roberts. University of Illinois.
[12] Patterns for Abstract Design. Paul Dyson. http://vasawww.essex.ac.uk/~pdyson/Study/Pad/pad.html.