CSPL Challenge

Version 0.1

email: constantine.plotnikov at gmail.com

Changes:

Version	Date	Description	Author
0.1	2012-01-29	Initial version	Constantine Plotnikov

Table of Contents

Challenge rationale

If we listen to a common line of reports from the trenches, we will see that the management of component systems is the most acute root problem that causes many other reported problems. We could also see that what makes users excited are tools the help them to manage component systems (dependency injection like Spring, Java EE 6, etc.). On first sight looks like library problem and DSL problem. And it is being attempted to be solved in that way. But if we look at IDEs (like Spring support in IDEA), they handle this library problem as language problem. And this is a major hint.

On other hand, if we look to contenders for next mainstream language like Scala, we will see an interesting thing. Scala gives developers warm and nice feeling, it helps, but it does not makes people in trenches excited as Spring. So it looks more like a box of booze vs. a box of first-aid kits. The Haskell now looks to commercial developers like symphonic orchestra concert: probably is a great things and people feel great after it. But how it is related to the problem at hand?

Note that I'm describing the opinion of the typical Senior Developers in normal commercial setting that work on project that started more than 10 years, has changed several development teams, still being actively developed and works in production.

So lets interpolate what would be a next mainstream programming language basing on the past history.

The common programming language evolution pattern was:

1. *Complexity Pain*: complexity of reasoning about some aspects of the programs growth fastest with program growth. At some time amount of efforts required to reason about some piece of code becomes unreasonable.
2. *Virtual Structure*: Organizing program according to virtual structure that makes reasoning about that aspect of the program easier.
3. *Explicit Structure*: Develop language, that makes virtual structure explicit by introducing additional meta-level constructs, that organize constructs of the previous levels into the structure.

I'm not stating that researchers and language developers actually thought in this way. It just looks that only evolutionary steps that followed that pattern were actually accepted in the mainstream (in the commercial setting).

There were nice languages that did not followed that pattern of evolution: LISP, Forth, Haskell, etc. However these languages never became mainstream. Probably the fact that they did not

followed evolution pattern was a reason why they did not became mainstream, the solved some other problem instead of the most acute one, and they broken connection to previous evolutionary stage too abruptly.

Note the level numbering is according addition of new meta-constructs that organize elements of the previous level. The level numbering is local to this analysis and it does not corresponds to the generation numbering.

I'm skipping programming with soldering iron, machine code, and assembler. The reason I'm skipping them, because they looks like independent lines of evolution.

Level 1. Statement-oriented programming(?) (FORTRAN, BASIC (with line numbers))

Possibly it should be really named punch-card oriented programming, since much of FORTRAN syntax and usability features are related to punch-cards and they looks somewhat strange on the terminals.

The primary element of the program here is a statement and and memory location. Almost all variables are global and there is no need to define their type (it could be identified by name). Flow of control is expressed in the terms of the statements.

Level 2. Structured Programming (C, Pascal)

The most acute complexity problem of the statement-oriented programming was to figure out control flow of the application. And what the statement could do at that point. The problem is thoroughly described in the the famous writing “Goto considered harmful”, so I will not dwell on it here.

The explicit meta-level structure on this level was code blocks and structures. Blocks allow to organize statements in the single unit, and the control flow is expressed in the terms of the blocks. So control flow is obvious from the text of the program. Structures do the same for the data.

Level 3. Object-Oriented Programming (C++, Objective-C)

The problem of the structured programming was that invoker of the procedure should know all about almost all needs of that procedure. So to design a procedure one should known complete functional decomposition of the task up to some primitives. Also data should be provided by

caller in the correct format. This obviously does not scale well.

The probably the first time it was IO. The problem was nicely dodged using ad hoc solutions, since a number of generally usable IO operations is limited. But Unix file handles, they provided a common interface for all file-like objects hiding behavioral and state details behind common interface. But was not a full OO system, since it was not possible to introduce own IO objects.

The next time it was GUI systems. But it was much harder to dodge, so there virtual structure that binds state and behavior together in form of controls. The state is hidden, and interaction is done by sending the messages. So typical GUI system has a full set of OO features:

- Dynamic dispatch
- Encapsulation
- Subtype polymorphism
- Object inheritance
- Open recursion

The features are expressed using constructs from structured programming, but they have virtual class-based structure.

The language support for such pattern was quite a natural thing to do. State and behavior were grouped in the classes. So for reasoning about piece of the code it was needed to just look to reachable objects and their contract, instead of thinking about the entire call tree and selecting appropriate functions. It was responsibility of object rather then caller to manage all state related issues.

Note, that I do not list Smalltalk, Ada, and other OO languages as language of this generation, since it did not became mainstream. Also Java and C# belong to the next generation.

Level 4: Component-oriented programming (Java, C#, VB)

This level is almost virtual from point of view of programming languages, since it did not required much of language support. The new features appeared and they are used, but they almost always technical.

C++ was a nice language, but there were major problems with reuse in it. If we skip technical ABI issues, the biggest problem were that objects have too much of responsibility. The was business aspects in the code and the technical aspects (serialization, memory management, etc). And the technical aspects soon started to require more developer time then business aspects. The different framework tried to address this issue, but these framework introduced own technical aspects, often incompatible with other frameworks. Just look at all ways to manage memory in the different frameworks.

The new generation solved the problem with different technical ways, some affecting the language and others did not. The effects were mostly in language semantics rather than syntax. But they all serve the purpose of separating intentional and technical logic.

- *Garbage collection* is probably most noticeable thing that got into the mainstream languages. This has killed 90% of framework integration problems. The answer to the question “who will free the memory has become none”
- *Dynamic code loading* and compilation to byte code is another thing that helped. It enabled wide usage of *containers* that took responsibility for technical aspects of the code. The practical containers started with XML descriptors for describing dependencies and technical aspects (.ser descriptor in Java did not work).
- The *code annotations* were an important steps here, since they have allowed to easier specify component-container contract. Aspect-oriented programming tried to explicitly handle some technical aspects, but it did not took off to mainstream on the language level. It is used to mostly support containers.
- *Properties and events* – the all languages have virtual or explicit support for properties and events that allow discoverable configuration and linking of components. In Java it is convention, in VB and C# explicit language support.

VB+ActiveX/COM was possibly the first mainstream component-oriented language. But it was used mostly for desktop applications and but later got to server side in form of ASP.

Note that Java from the start supported a component framework in from of AWT. And there were even a container for applets. So component requirements implicitly leaked in the language, and Java were able to address need for components in other areas. C# started as the better Java and VB, so it inherited its component orientation from the both (and in this area VB is possibly bigger inspiration).

Level 5. Component System Languages

Lets see what problems we have now in commercial settings.

- There are multiple configurations of the application in different environments. Sometimes it is just configuration parameters in other cases it is different component graphs (unit testing and production environments are radically different).
- There are multiple sources of application configuration in different environments
- There is a need to link different component graph with different life-cycle (for example plugins to IDE)
- There are implicit requirements from one component graph to another (in Spring it is very hard to understand where the referenced component could be found).
- Since component graph definition languages are mostly dynamic, global analysis of application is required (for example Spring xml files). Run-time errors are quire common. Unit tests are less useful, since local correctness is less linked to global correctness.

- Concurrency is either severely limited (GUI and EJB) or cause hairy problems (Spring + manual thread management). In case of EJB sometimes there are both problems.

These are quite acute problems for big systems that are maintained for several years or more (new teams make the problem harder since they lack global application knowledge). So the next logical step would be developing a language for component systems. It could be internal or external language with respect to component language, there are advantages and disadvantages with respect to both approaches. But it would be a new language with own type system and own operations over component systems.

Challenge Description

I would like to propose a challenge for designing programming language that supports component systems (graphs) as first class citizens. There are two sub-challenges:

- **CSPL-A**: Application programming language (web and desktop applications, compilers, IDEs, etc.). This proposal is targeted to long-living applications typical for commercial settings, where application could outlive several teams working on it, and support is done in hurry and by people that lack global knowledge about application.
- **CSPL-D**: Dynamic application programming language (web and desktop applications, compilers, IDEs, etc.). This proposal is targeted agile teams that known ins and outs of application and in some sense own it.
- **CSPL-L**: Low-level system programming language (operating systems, drivers, real-time applications, etc).

Note that components mentioned in this challenge are defined as the following: A component a set of classes and interfaces that implement some functionality and generally has a container and consumer contract regarding to provided functionality. And some part of the functionality might be provided by container. The component usually have a published interface. The examples of components:

- GUI controls
- EJB
- Servlets
- ASP controls
- ActiveX/COM/DCOM
- CORBA objects
- Spring Beans

The component system is a system of components that has a common life-cycle and collectively serves a common purpose. The examples of different component systems:

- Plugins for IDE (Eclipse, IDEA, Netbeans)
- OSGi Services and related frameworks

- Guice, Spring, and others
- Web Applications
- Web Application Sessions
- GUI Forms
- Web Pages
- Object Databases

Common Requirements

This section lists requirements common for CSPL-A, CSPL-D, and CSPL-L.

CSPL.1: The component system should support component-based programming

Component-based programming should be available with minimal overheads. This includes:

- Common support for implementing container/component contract (AOP falls in this category)
- Support for component life-cycle operations
- Generic OOP programming

Note that support for component system programming could be internal DSL or external DSL with respect to component-based programming language.

CSPL.2: There should be support for defining structure of the component system

The structural aspect is very important, and there should be an easy way preferably with language support to define the internal structure of sub-system in the terms of the components. This should include:

- Listing components and component set
- Specifying component parameters through system requirements
- Specifying system requirements
- Having static and dynamic components in the system
- Specifying links between components and component dependencies
- Specifying link inference rules (e.g. auto-wiring in Spring)
- Specifying collective behavioral rules

Examples:

- Probably the best example here is Spring Framework. But it has problems with dynamic components.
- OSGi also have interesting ideas particularly with respect of dynamic components.
- Examples of collective behavior rules are eclipse extension points, event propagation rules and visibility rules in GUI components, and spring life-cycle rules.
- Groovy builders are also interesting as object creation syntax.
- Scala and Haskell parser combinators in practice also create a component system that parses text.

CSPL.3: It should be possible to define contract of component system in terms of its parameters and what it provides

It should be possible to figure out what component system needs to be created and what it provides to container and consumers.

The parameters of component system might include:
- Links of different kinds to other component systems
- Some values or objects of the specific types as parameters
- Used containers for components
- etc

Parameters could be static or dynamic. If parameters are dynamic, it should be defined what happens when parameter changes.

The system itself could provide:
- Link ends with some semantics
- The exported values or objects
- Containers for components to be used by other systems
- etc

The provided aspects also could be dynamic or static.

Examples:
- Most of the current component systems lacks explicit contract. But implicit contract is defined for all of them.
- ML module system is one of few examples of explicit contract, but lacks many other aspects of component system.

CSPL.4: It should be possible to extend and specialize system

definition

It should be possible to create definition of new component system using the definition of other system. It might be required for the extended or specialized system to define extension and specializing points.

Examples:

- In Spring it is possible, but looks horribly since these points are implicit.

CSPL.5: It should be possible to link to other component system

It should be possible to link two component systems together.

- There should be implicit or explicit contract associated with such link
- The effect of the link on life-cycle should be defined (subsystem/super-system relationship)
- It should be possible to make dynamic links with different parameters
- It should be possible create subsystems that live as long as link lives
- There should be support for inferring links

Examples:

- There are few good examples here. The version implemented in the Spring is somewhat messy (imports and parent context).

CSPL.6: There should be expression/query language for system

There should be a language what allows queries and expressions in terms of system elements that considers system links, subsystem - super-system relationships, etc.

Examples:

- Spring expressions
- JSTL expressions
- LINQ
- ODMG OQL

CSPL-A Requirements

There are requirement for CSPL-A variant.

CSPL-A.1: The language should support garbage collection

I guess I do not need to explain why.

CSPL-A.2: The effect of garbage collection and program termination on component system life-cycle should be well-defined

Also no comments.

CSPL-A.3: There should be static type system for component systems

The requirements and provided aspects should be mostly listed explicitly. It should be possible to understand what the subsystem listed to basing on components listed in the scope, so the local analysis is needed in the most cases.

Examples:

- ML module system could be quite close to what is needed (?)

CSPL-A.4: There should be support for refactoring

Refactoring is important for the mess most of commercial applications become with the time (and time could be quite long):

- It should be possible to define refactoring support for system definitions.
- It should be possible to separate: safe, probably safe, and known breaking refactorings.
- It should be possible to identify all real usages of the component system an components included in it.

Examples:

- No good examples. IDEA implements refactoring over Spring. But things gets messy with multiple configurations where beans are named the same, but use different configuration.

CSPL-A.5: There should be a common concurrency model

The concurrency for the components is messy, for component systems it will be even more messy.

Examples:

- The best concurrency model that I known for such language would be sub-actor concurrency model from E programming language (www.erights.org). Note that the same concurrency model is used by GUI loops, and GUI it practically the first place that had to deal with component systems. The version in E language is just a bit more generic. Note that promises also provide a very nice way to coordinate life-cycle dependencies.
- The traditional actor models are too coarse grained here (in sub-actor concurrency model one vat or event loop corresponds to the single actor).
- Synchronization does not works here in practice, as we can see on example of GUI toolkits. Almost all of them insists on single-thread access.

CSPL-A.6: There should be a common security model

There should be a security model for component systems. Naturally this security model should be dependent on component links and it should be possible to specify different security rules for different instances of component systems (for example depending on parameters).

Examples:

- Stack-based security checks (Java and C#) are problematic and they are often turned off for application servers.
- Capability security model looks like the only serious contender here, since it is connection based (www.erights.org).

CSPL-A.7: There should be a common unit and integration testing model

It should be reasonably easy to write tests for component system. It should be possible to instantiate component system for test or set of tests (with appropriate test containers, stubs, and mocks) with reasonable efforts.

Examples:

- Spring Test

CSPL-A.8: There should be support for different possibly interleaving life-cycle scopes

The life-cycle of different component system is obviously different and some component system could know about each other just for a short time.

Examples:

- Scopes in Spring

CSPL-A.9: There should be support for packaging component systems

There should be a support for packaging component systems. This could be include:

- The compiled format for component system definitions. It should be fast to load and execute, and with appropriate correlation with source for error reporting. It could be
- The support for automatic linking of component systems on module load.
- The support for module-level visibility for system definitions

Examples:

- Spring OSGi
- Plugins in Eclipse and IDEA

CSPL-D Requirements

I would argue that requirements are the same as for CSPL-A except for CSPL-A.3 requirement. I also prefer static type systems, so I do not really interested in such language and I could not come with ideas what one would want from it.

CSPL-L Requirements

All requirements are the same as for CSPL-A except for CSPL-A.1, CSPL-A.5, CSPL-A6, CSPL-A.9 requirements. CSPL-A.5 is an optional requirement. There are also some additional requirements. I also lack fresh experience in the system programming, so requirements here are somewhat out the blue.

CSPL-L.1: The memory management strategy should be specified for the system

It should be possible to define memory management strategy for the component system.