Fractal Containment® Technical Brief

August 18, 2011

ZedaSoft, Inc.
2310 Gravel Dr
Fort Worth TX 76118
817-616-1000
www.zedasoft.com

Introduction

The Container-Based Architecture for Simulation (CBA) is a family of object-oriented software frameworks that together implement ZedaSoft's simulation system design "A Container-based Architecture for Simulation of Entities in a Time Domain" (U.S. patent 7,516,052).

CBA is a general system for managing the update of arbitrary data over time. It makes no assumptions about the data being managed; instead, it provides an object assembly model, execution life-cycle management, network distribution and other general services that can be used to model specific problem domains in a flexible way.

CBA's object assembly model and execution life-cycle management strategy is called Fractal Containment®.

Early versions of CBA were based on a container model that was enforced formally only at the topmost domain object level. This approach was successful, but while using it on a daily basis we learned that the object assembly and life-cycle API calls were often needed by lower-level objects too. Over time it became clear that the container model should be generalized and applied in a consistent way at most levels in the object graph.

Why "Fractal"?

In mathematics a fractal is defined as "a geometric pattern that is repeated at every scale". We chose this term because CBA's object assembly model is recursive, and its general structure and execution are identical at every level.

Motivations

Fractal Containment® is based on the observation that nearly everything in traditional object-oriented programming is a container of one kind or another. Most classes include some set of internal fields, each of which in turn may be instances of other classes that define other sets of internal fields, and so on to some arbitrary level of complexity. Instances of such classes can be woven into a tree of objects within the limits of assembly rules hard-coded in the various classes. Such a tree of objects is called an object graph. Object graphs are the software representation of some real-world problem that we want to model in the computer.

In traditional object-oriented programming the assembly and execution of object graphs is based on code that weaves components together in a rigid way based on a static model. Sometimes this is what you want, but in environments where flexibility is a primary concern this approach is less desirable because software developer intervention is required to extend, delete, replace, or rearrange the object graph's components. In addition, object graphs are typically closed once built: they can only be modified at runtime within hard-coded constraints, their structure can't be examined, and arbitrary locations in the graph can't be interrogated for data.

What if we could generalize the container model and allow object graphs to be assembled declaratively outside of code? Potential benefits include:

- Software components would follow common patterns for discovery of surroundings, execution, and object graph assembly, making it simpler for any development team member to contribute components to any point in the graph.

- The software component assembly model would better follow the analogous assembly of components in the problem domain. The code components would define the possible parts that could be assembled, and the declarative assembly would say how to weave them together into a functional unit.
- Assembly could be pushed downstream into the hands of the end-users so the object graph could be easily customized to meet the end-users' requirements. New components could be added, or existing components could be deleted, replaced, or rearranged, without requiring modification and rebuilding of code.
- Object graph modifications could take place at runtime as well as at assembly time with fewer constraints on where and when components execute.
- Components could be easily inserted into any container that could provide a sufficient execution context, providing the foundation for "what-if" container assemblies.
- A consistent execution life-cycle could be enforced across the entire object graph.
- Object graph structure could be interrogated and described to other applications that may want to display the structure graphically to an end user.
- Arbitrary data could be extracted from any component in the object graph to support data recording or graphing.

The Formal Fractal Containment® Model

Fractal Containment® is a formalization of the general container model discussed in the previous section, and is based on the following simple concepts:

- *Contained objects are named and uniquely identifiable.* These names are the equivalent of hard-coded field names in the traditional class/field model, and can be concatenated together to define a path to a specific object anywhere in a container object graph.
- *Contained objects implement a formal life-cycle.* The life-cycle defines a formal interface that lets contained objects identify themselves, allows their enclosing container to let them know when there are structural changes in the container (including their own addition and removal), and allows their enclosing container to manage their routine execution.
- *Contained objects discover their surroundings at runtime.* This means we are free to place a specific object anywhere under a container that can provide a proper execution context.
- *Contained objects run in a sandbox that protects all objects from the potential misbehavior of any object in the graph.* This makes object graph assembly and execution more robust. If an object does something undesirable, then it is discarded, a message is logged, and the rest of the graph continues normally.

- *A container contains zero or more contained objects, and a container can be a contained object in some other container.* This is what puts the “Fractal” in Fractal Containment®: any contained object may in turn be another container, and if you look inside you see the same pattern again.

We can now start to form a picture of what a fractal object graph might look like in a general case:

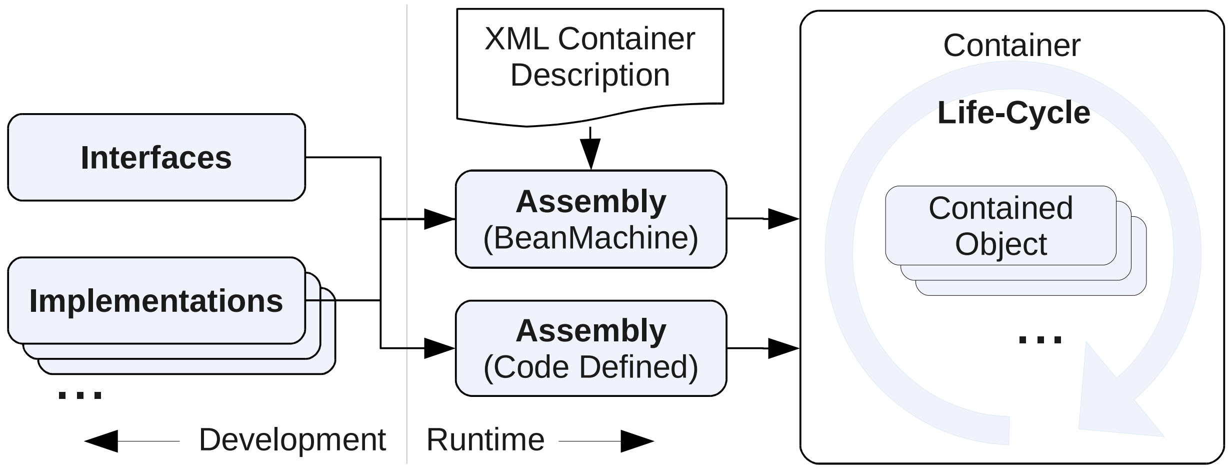
```
Container “root”
    Container “1.1”
        Container “2.1”
        Container “2.2”
        Container “2.3”
            ContainedObject “3.1”
            ContainedObject “3.2”
            ...more...
        ...more...
    Container “1.2”
        ContainedObject “2.1”
        ...more...
    Container “1.3”
        ContainedObject “2.1”
        ContainedObject “2.2”
        ...more...
    ...more...
```

This looks at lot like a table of contents for a text document, and the analogy is applicable. Under Fractal Containment® you can move an object around in the object graph just as easily as you can grab a block of text in a text document and drop it into a different section. Similarly, you can define an arbitrary hierarchy of objects just as easily as you can define the section arrangement in a text document.

In this illustration the Containers and ContainedObjects at the various levels represent arbitrary code sequences that perform state data maintenance operations. CBA doesn't care what the state data represents, and doesn't care what the operations are as long as they are well-behaved.

Application Within CBA

Within CBA Fractal Containment® allows any set of objects that implement the execution life-cycle to be assembled into an object graph based on either an XML description or an assembly defined in code. For each container the general process is as follows, regardless of the container's ultimate level within the object graph:



Interfaces – define the types of components in the system and the methods that must be implemented by each type. Put another way, interfaces define the contract between types without defining how the contract is implemented. A particular set of interfaces defines a “domain model”.

Implementations – provide concrete implementations of a set of a domain model interfaces. For each interface there can be one or more implementations. Properly developed implementations fulfill the contract of their corresponding interface and can be interchanged freely depending on the user's needs. Reasons you might need multiple implementations of a given interface include varying levels of fidelity, accommodating export restrictions, changing classification levels, or general experimentation.

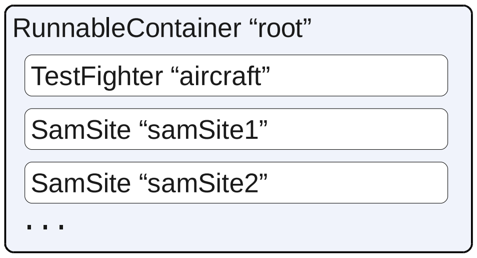
Assembly – provides runtime assembly of specific implementations into a functioning object graph. Any combination of accessible domain model interface implementations can be used as long as they realize the contract defined by the domain model interfaces. Assembly can take place either in code, or declaratively through XML. CBA's XML-based assembly mechanism is called BeanMachine. BeanMachine consumes an XML file that specifies which implementations to use, what their initial property values should be, and how they should be connected. BeanMachine creates the specified objects, initializes them, then weaves them together into a functional object graph.

Life-Cycle – a formal interface that allows objects to participate in the object graph assembly step and regulates their routine execution. An important feature introduced by the Life-Cycle is a formal initialization phase in between object creation and routine execution. During this initialization phase the objects' property values are initialized, and they're given an opportunity to discover their surroundings in the graph.

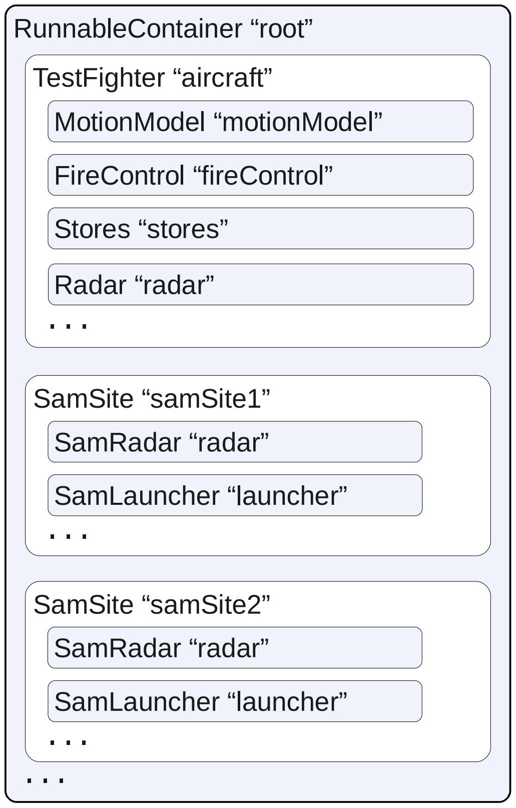
Practical Examples

Simulation

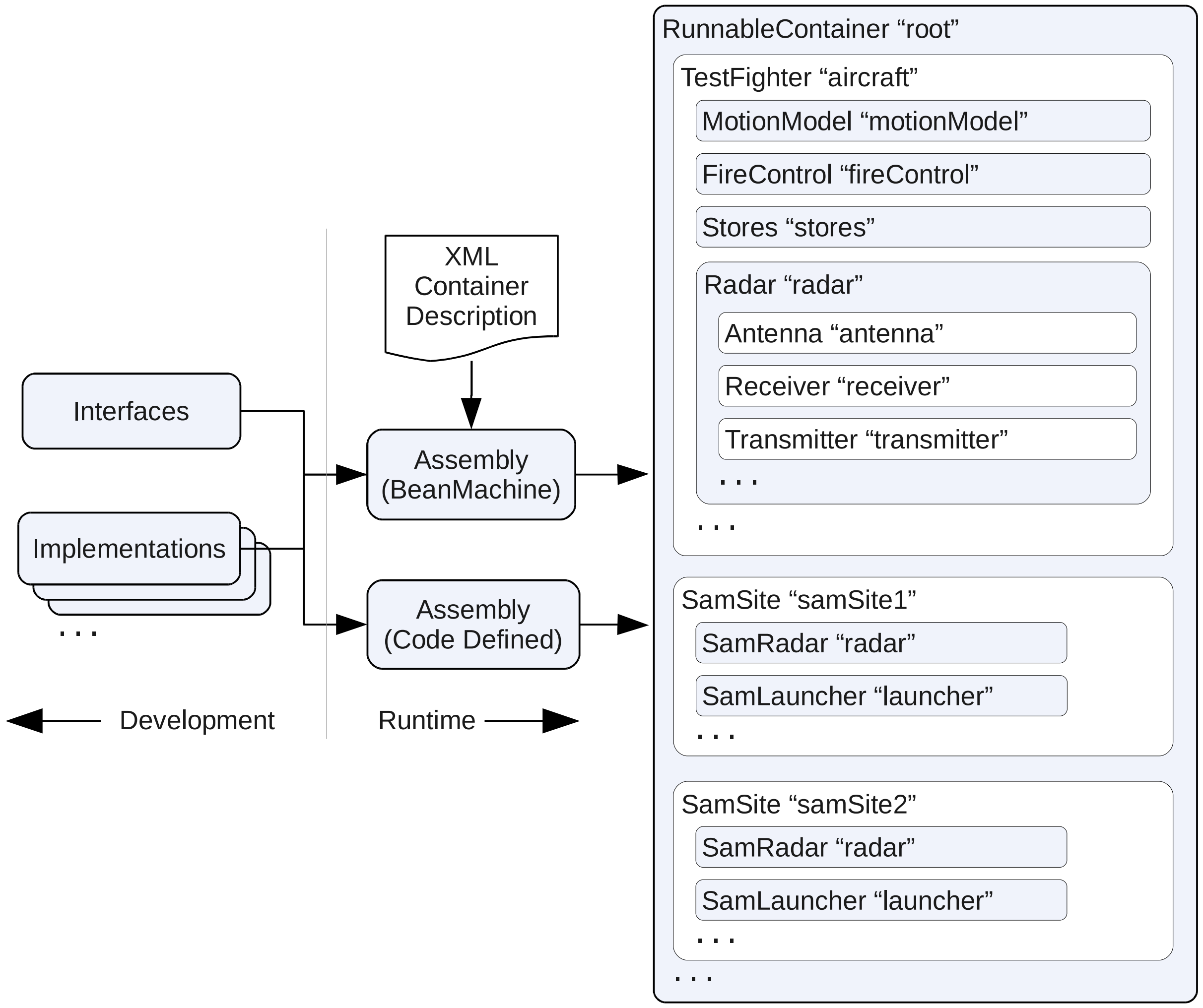
Suppose we developed a set of components to support an aircraft simulator. The top-level container may contain an aircraft and some SAM sites to act as targets. This was as far as the formal container model was taken in early versions of CBA:



Fractal Containment® made the container model pervasive. Now the aircraft and SAM sites can each be formal containers with immediate contained objects to represent internal subsystems:



The pattern may continue with some of the subsystems implemented as formal containers. In this example we have expanded the aircraft radar, and added the assembly mechanisms to the illustration:

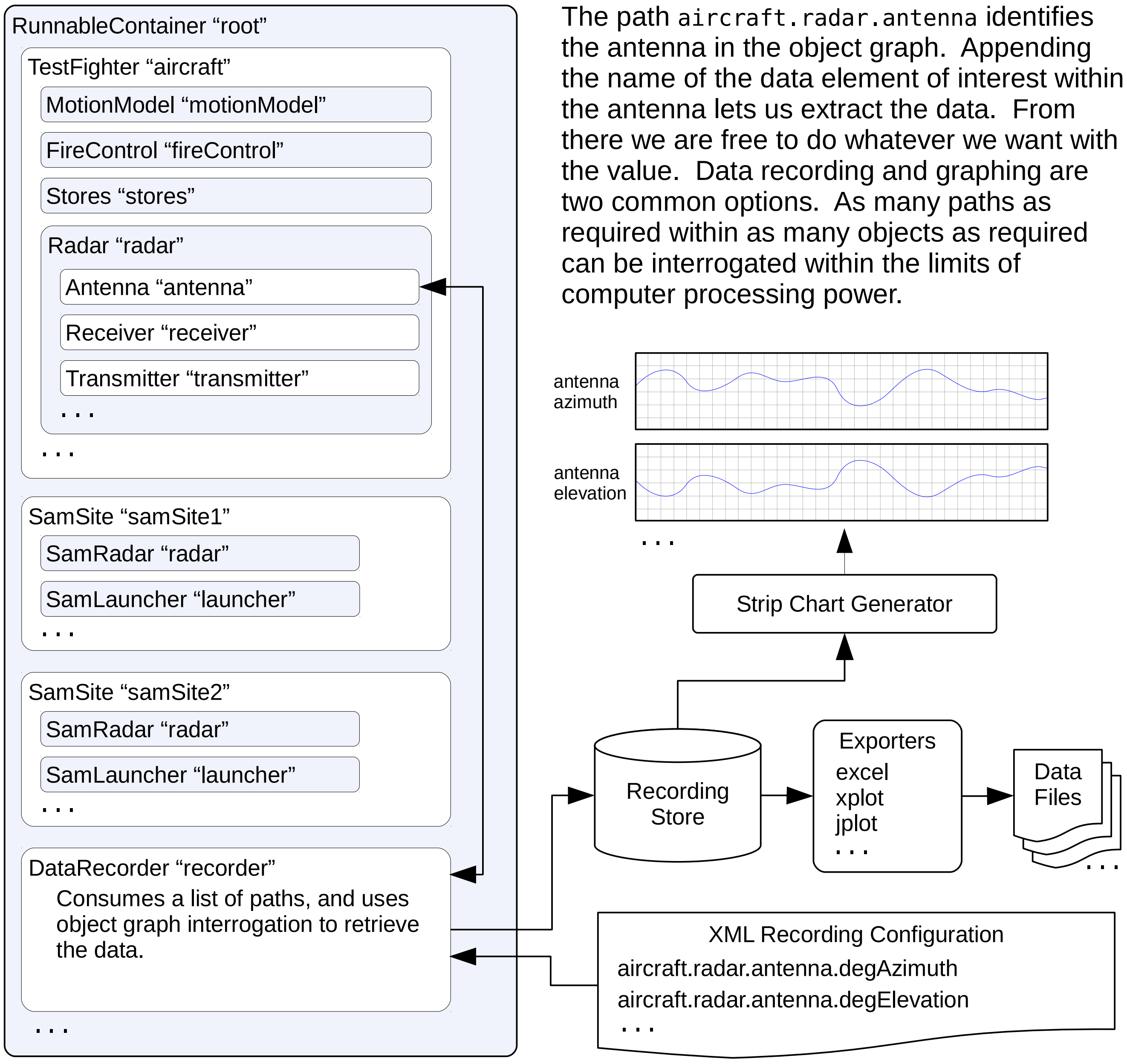


Hierarchical object graph structures like these are easily described in XML as assembly instructions for BeanMachine. The graph can use any accessible interface and implementation classes that implement the life-cycle interface, and arrange them however is required as long as the arrangement provides a sufficient execution context for each object. The graph has no inherent complexity limit; it can be as wide and deep as necessary.

When implemented with a language that supports a public runtime system with field-level access, Fractal Containment® will also allow you to easily interrogate ad-hoc data anywhere in the object graph by evaluating a symbolic path against the root container. The symbolic path locates an object by specifying every object name between the root and the final object of interest.

For example, suppose the radar's antenna includes a simple data property named “degAzimuth”. This item can be identified and extracted by evaluating the path “aircraft.radar.antenna.degAzimuth” against the root container.

The path `aircraft.radar.antenna` identifies the antenna in the object graph. Appending the name of the data element of interest within the antenna lets us extract the data. From there we are free to do whatever we want with the value. Data recording and graphing are two common options. As many paths as required within as many objects as required can be interrogated within the limits of computer processing power.



If you are experienced with object-oriented development techniques you may be wondering at this point why this is a different approach. Why not just declare some static hierarchy of objects and assemble them in code? You could, but you don't get dynamic assembly that way. You could fix that problem by making some or all of your static model pluggable, but you still don't have the flexibility to move objects around in the graph easily. You could further generalize your model, but at that point you have Fractal Containment®.

Fractal Containment® allows you to interrogate the graph structure in a language-neutral way. Languages with respectable public runtime systems would support this without Fractal Containment®, but languages like C++ would not. Fractal Containment® gives you all these things at once, and could be implemented in a variety of computer languages.

The important point to have absorbed by now is that Fractal Containment® provides a public structure that is extensible, open-ended, and can be interrogated. Furthermore the assembly of the object graph is declarative, assembled at runtime, and can make use of any appropriate available components including ones you've written yourself.

Changing Simulator Classification Levels

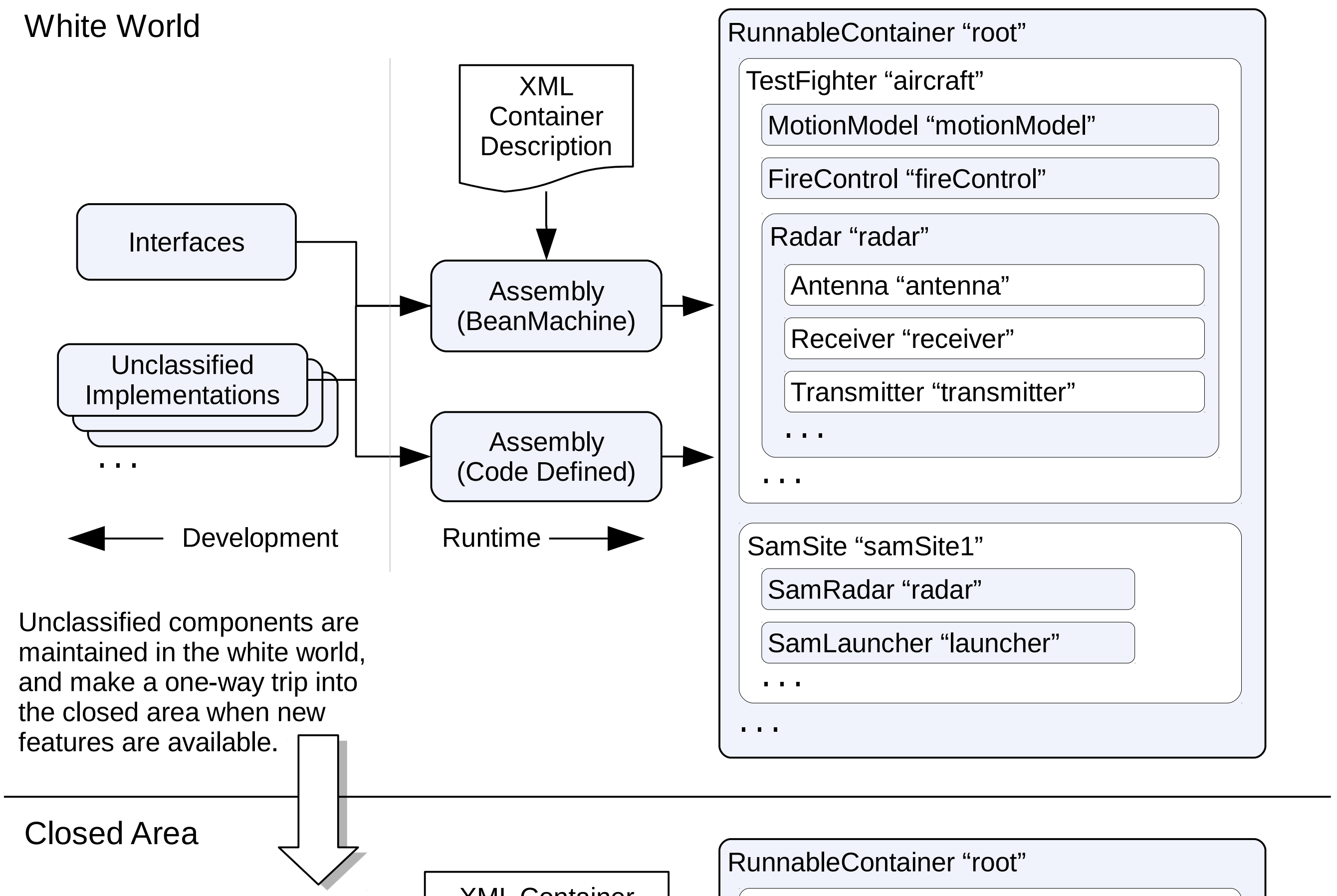
In high-fidelity military simulation it is common to need at least some components whose code and/or data are classified. A common and unfortunate side effect of this in typical monolithic software systems is that the entire code base can be tainted by the presence of a handful of classified components. This usually leads to a split in the source code so it can enter the closed area, making it difficult to coordinate modifications to components in the unclassified parts of the system. The typical result is that the code base inside the closed area deviates from the point of the split, resulting in revision control difficulties.

A better solution would be to make it easy to run the unclassified system outside the closed area so the core system can be maintained from a common code line, then bring releases of the unclassified system into the closed area when necessary and replace selected unclassified components with their classified counterparts. The only components whose development would need to be confined to the closed area would be the classified components proper.

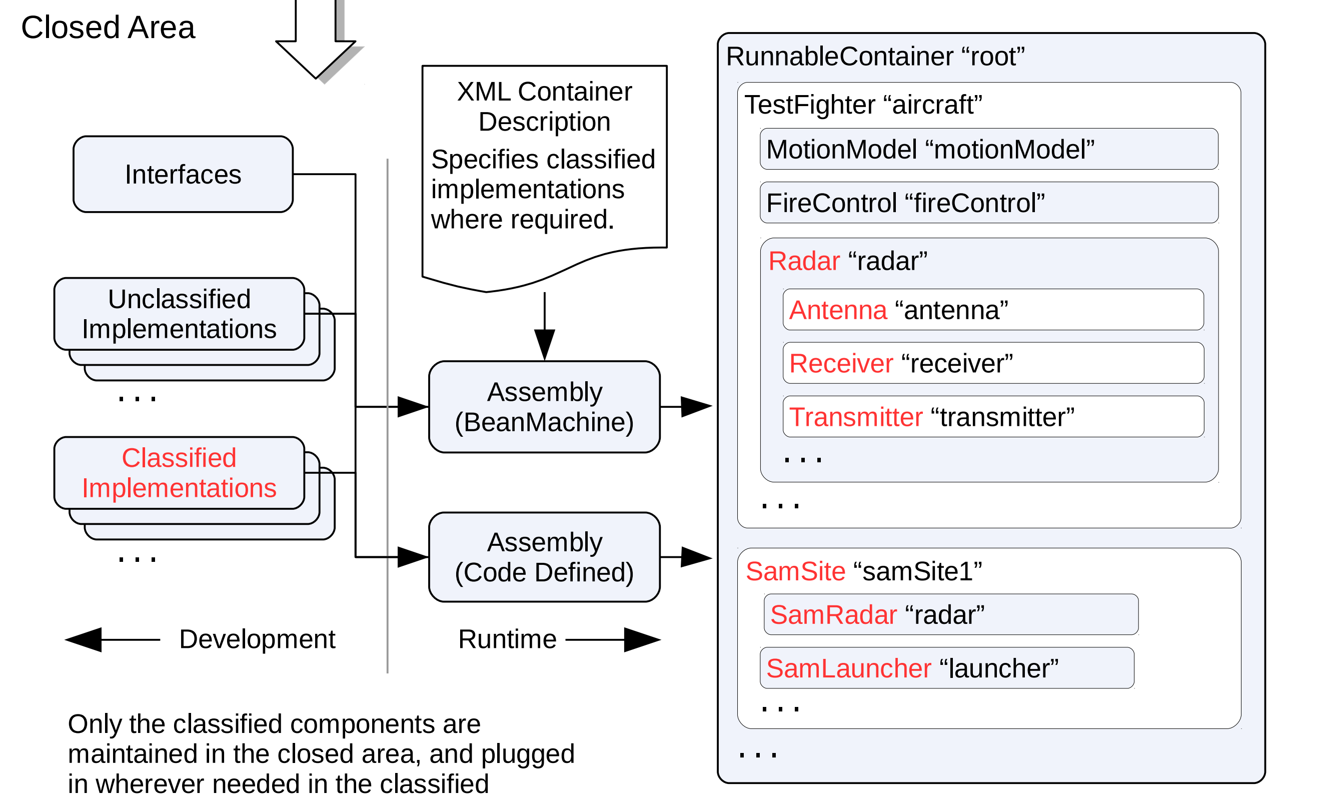
Fractal Containment® provides this capability at whatever level of granularity is required, in a more flexible way than is possible with traditional development approaches. The classified components could be any combination of entire platforms, platform subsystems, or even sub-components of subsystems.

It's important to note that because the assembly is brought together declaratively at runtime the choice of which components to use is per-container. This means any combination of components can be used per-subsystem, per-entity, per-scenario.

In the example that follows we maintain a runnable, unclassified system in the white world, and deploy the entire system into the closed area when new features are available. A classified aircraft radar and SAM site model is maintained in the closed area. The XML container descriptions in the closed area can choose either the unclassified or classified implementations, which are assumed to implement the same interfaces and therefore can be exchanged freely.



Unclassified components are maintained in the white world, and make a one-way trip into the closed area when new features are available.



Only the classified components are maintained in the closed area, and plugged in wherever needed in the classified scenarios via the container description.

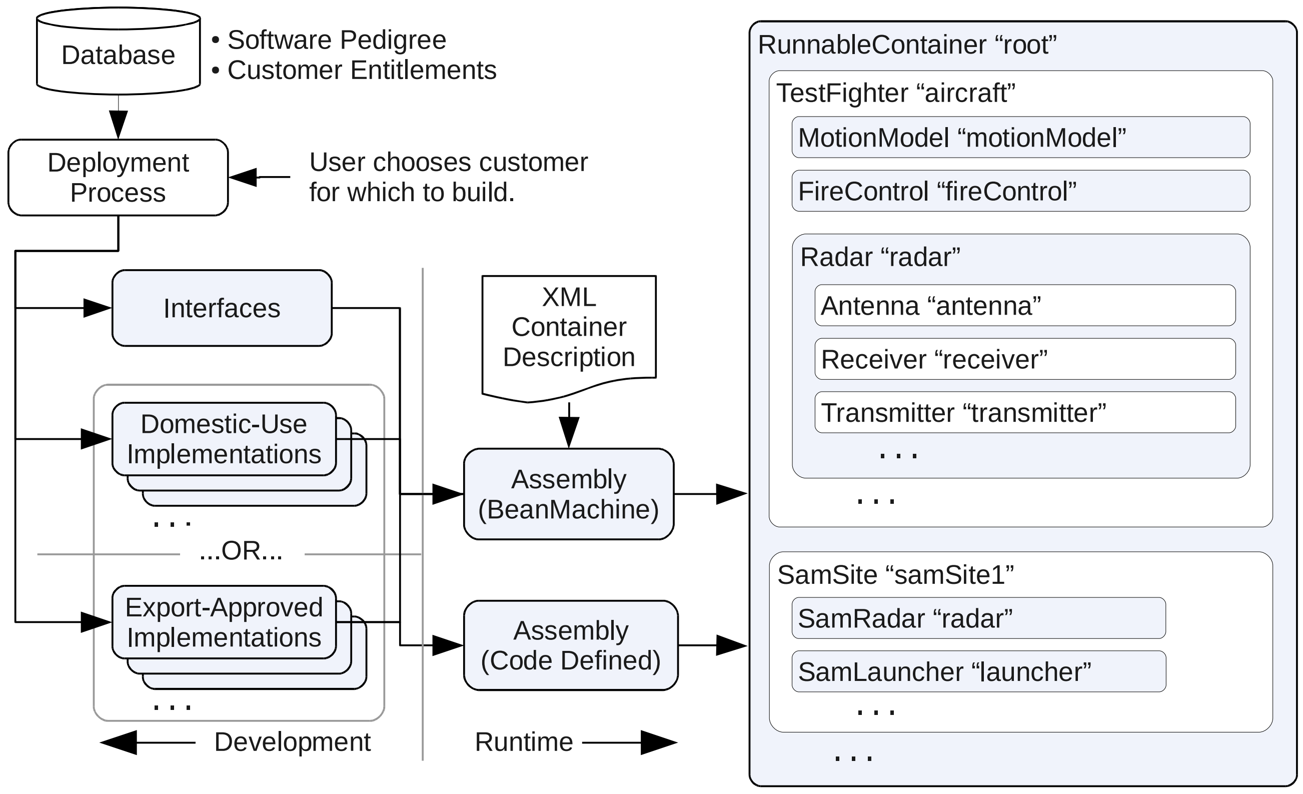
Accommodating Export Restrictions

One creative potential use of Fractal Containment® is to drive deployments using a database that tracks software pedigree and customer entitlements.

For example, suppose you have a simulator that supports a certain set of capabilities that can be deployed anywhere in the U.S., but some components of the system cannot be taken to certain countries. The common approach to preparing the software set for overseas use is to have software developers remove the non-compliant components and splice in suitable substitutes where necessary. This typically is a time-consuming manual task that also risks the accidental inclusion of software that cannot go outside the country.

A system that could quickly assemble an export-approved software set would save labor and time, and reduce the risk of mistakes since the process is repeatable and only relies on the occasional maintenance of the data that describes the software pedigree and customer entitlements.

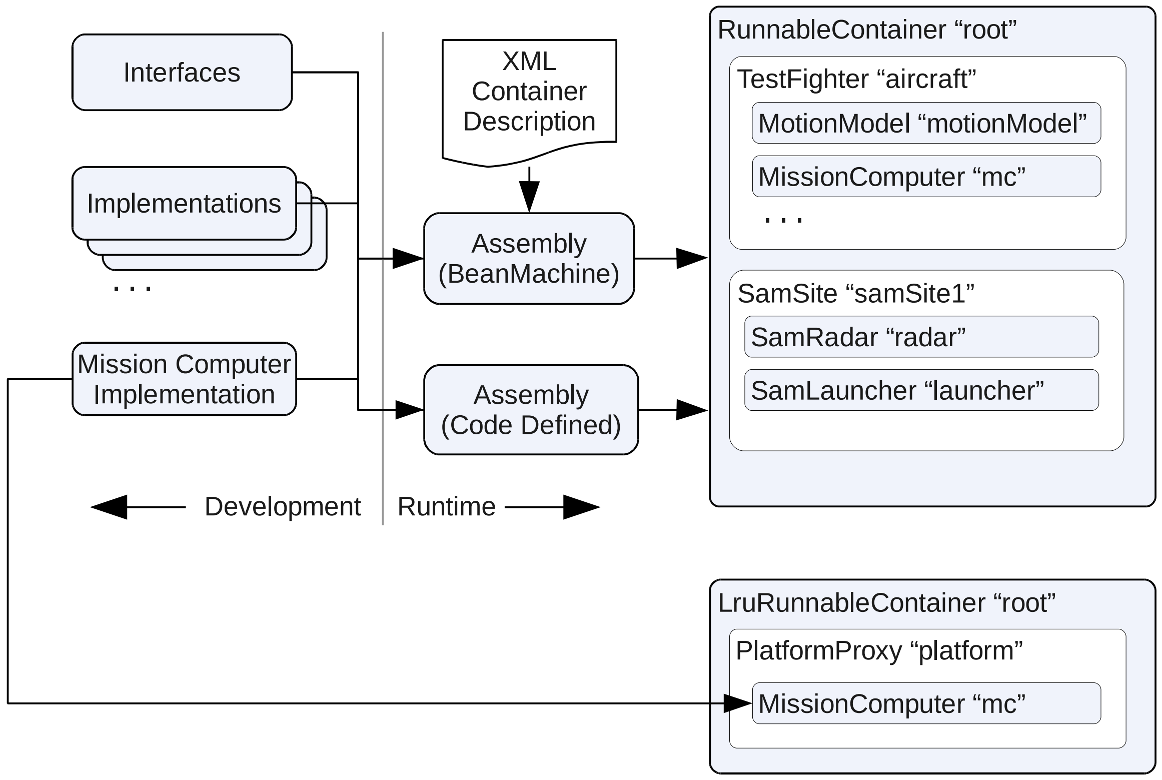
In this example a database-driven assembly process decides which implementations to use based on what a particular customer is allowed to receive. The implementations in this example are assumed to implement common interfaces and therefore be interchangeable. Furthermore we assume the domestic-use and export-approved classes use identical class names, which means that all the process needs to do is choose which components to deploy and the runtime assembly process doesn't know the difference.



Supporting End-To-End Software Flow

A holy grail of software systems used on military platforms is the ability to develop OFP software in the simulator and have it flow all the way down to the LRU on the physical platform. Setting aside for a moment all the non-technical issues that might stop this from working, Fractal Containment® would provide a viable technical solution for this problem because a properly developed component could be run in any container that could provide a comprehensive execution context.

In this example a Fractal Containment® runtime is available in a simulator and an LRU. In practice a Systems Integration Lab would also be involved, and would use a combination of these runtimes. Each of these would host a container somewhere in the hierarchy such that a proper execution environment for a hypothetical Mission Computer could be established. Provided the execution environments are comprehensive the same Mission Computer could be plugged into each of the environments without modification. The inclusion of the OFP code in the simulator is specified declaratively and assembled at runtime, per entity, per scenario. Some entities may choose to use the OFP, while others may not. The LRU would always perform a repeatable load of the OFP.



Summary

CBA's Fractal Containment® is a regular object assembly and execution life-cycle model that allows properly constructed objects to be assembled in a flexible declarative way. Object implementation is separated from the assembly through well-defined interfaces, allowing different implementations of a given interface to be exchanged freely to support security concerns, export restrictions, general experimentation, or other activities that require a flexible infrastructure. Components can be specified per-subsystem, per-entity, per-scenario, and potentially to even finer levels depending on the degree to which Fractal Containment® has been used.

Fractal Containment® is currently in use in military and civilian simulators supported by ZedaSoft, and has demonstrated all benefits envisioned during initial capabilities and design discussions.