Name

Puppeteer

Problem

Complex simulations often involve coupling multiple software abstractions created by domain experts in disparate disciplines. How does one communicate information between abstractions without exposing the sender's interfaces to the receiver (or vice versa) and thereby creating *software* dependancies between them? How does one construct entities that describe one abtraction's inextricable *mathematical* dependancies on another without exposing the abstractions to each other?

Context

In physics-based simulations, the linking of models from multiple physics subdisciplines is commonly termed *multiphysics modeling.* Weather and climate simulations, for example, link models that describe the dynamics of air, sea, and ice along with submodels for chemistry, radiation, turbulence, cloud formation and other relevant phenomena. As another example, stealth aircraft design combines aerodyamic and structural analyses with electromagnetic signature prediction.

Forces

The driving force is the desire to construct loosely coupled classes by limiting the interface dependancies that bedevil software maintainers when changes to one interface propagate out to affect all classes that rely upon that interface.

One opposing force is the need to pass data between packages. For example, the structural dynamics solver requires boundary pressure and shear stress data from the aerodynamics solver; while the aerodynamics solver requires boundary geometry input from the structural solver. These underlying mathematical and physical dependancies naturally nudge the naive developer towards writing packages that directly reference each other and thereby creating strong software dependancies.

Succumbing to the latter force, a developer might allow the receiving object to request information by invoking methods on the sending object. This, however, would make the receiver's implementation depend on the sender's interface. In an even more tightly coupled scenario, the receiver might pass an instance of itself to the sender, allowing the sender to invoke methods on the receiver to query it for metadata such as the size and layout of the information the receiver desires. This leads to dependancies in two directions.

A second opposing force is the need to describe how one model depends on another. For example, many nonlinear solvers require a Jacobian matrix, the off-diagonal elements of which comprise the partial derivatives of the right-hand side (RHS) of each governing equation with respect to the dependent variables that make up the solution vector. When the relevant governing equation and dependent variable live inside different abstractions, violating the abstractions by allowing each class to expose these private details might seem attractive.

Solution

A Puppeteer encapsulates and controls references to each submodel, or *puppet*, in the simulation. It does so by delegating operations to the puppets and passing return values from the methods of one as arguments to the methods of another. By requiring that each datum so communicated conform to a format independent of the sending and receiving classes, the Puppeteer can obviate the need to expose information about the data structures inside the puppets. For example, the data might be communicated using types intrinsic to the implementation language or they might be encapsulated in an instance of a class commonly known by all puppets.

The Puppeteer encapsulation mechanism is aggregation, and might specifically be composite aggregation (composition) depending on whether the lifetimes of the puppets coincide precisely with that of the Puppeteer. By its nature, aggregation implies that the puppet interfaces are exposed to the Puppeteer but not the reverse. Likewise, a given puppet's interface need not be exposed to the other puppets.

The Puppeteer exploits the regularity and predictability of the inter-abstraction communications defined formally in the terms that couple the governing equations. The developer of each puppet requests the requisite coupling terms by placing them in the argument lists of public methods. The Puppeteer developer fulfills one puppet's requests by calling public accessors on the other puppets.

In order to assemble quantities that describe the cross-coupling of models with private data and private governing equations, the Puppeteer developer examines the list of data it must shuttle between puppets. The developer can determine from this list what data dependancies exist and which can be neglected. For example, to construct a Jacobian matrix, the Puppeteer might need to calculate the partial derivatives of each governing equation's RHS with respect to each dependent variable in the simulation. If a given puppet requires no data from a second puppet, the Puppeteer sets the corresponding derivatives to zero. If the given puppet does require data from a third puppet, The Puppeteer interrogates the given puppet for the desired partial derivatives. Since the rank of the response matrix depends on the number of governing equations and dependent variables, both of which are hidden inside the puppets, the communication must take place in a format that allows for dynamically sizing the stored result. One might employ C++ Standard Template Library (STL)

vectors, for example, or Fortran 2003 allocatable arrays.

Invariant

When a multiphysics simulation is accomplished by writing time integration expressions (e.g., Runge-Kutta steps) employing overloaded operators on an abstract class (as detailed in the Examples section of this pattern), there exists at most one Puppeteer. In simulations involving only one physics submodel, no Puppeteer is required. In simulations with multiple submodels, the desire to make the time advancment code application-independant suggests its expressions must be agnostic regarding the specific couplings between the state variables. This suggests that all these couplings must be encapsulated in the object being integrated, said object being the Puppeteer by definition. Alternatively, if one builds the coupling details into the time integration abstraction, then that abstraction effectively becomes the Puppeteer.

Examples

The text in this section is adapted from Rouson, Adalsteinsson and Xia [6].

One can describe a broad swath of multiphysics phenomena with equations of the form

$$\frac{\partial}{\partial t}\vec{U} = \vec{\mathcal{F}}(\vec{x},t), \quad \vec{x} \in \Omega, \quad t \in (0,T] \tag{1}$$

where $\vec{U} \equiv \{U_1, U_2, \ldots, U_n\}$ is the problem state vector; $\vec{x}$ and t are coordinates in the space-time domain $\Omega \times (0,T]$; and $\mathcal{F} \equiv \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$ is a vector-valued operator that couples the state vector components via a set of governing ordinary-, partial-, or integro-differential equations. Closing the equation set requires specifying appropriate boundary and initial conditions

$$\vec{B}(\vec{U}) = \vec{C}(\vec{x},t), \quad \vec{x} \in \Gamma \tag{2}$$

$$\vec{U}(\vec{x},0) = \vec{U}_0(\vec{x}), \quad \vec{x} \in \Omega \tag{3}$$

where Γ bounds Ω, $\vec{B}(\vec{U})$ typically represents linear or nonlinear combinations of $\vec{U}$ and its derivatives, and where $\vec{C}$ specifies the values of those combinations on Γ.

A common step in solving equation (1) involves rendering its RHS discrete by projecting the solution onto a finite set of trial basis functions or by replacing all spatial differential operators in by finite difference operators and all spatial integral operators in by numerical quadratures. Often, one also integrates equation (1) against a finite set of test functions. Either process can render the spatial variation of the solution discrete, while retaining its continuous dependence on time. One commonly refers to such schema as *semi-discrete* algorithms. The resulting equations take the form

$$\frac{d}{dt}\vec{V} = \vec{\mathcal{R}}(\vec{V}) \tag{4}$$

where the RHS vector function $\vec{\mathcal{R}}$ contains linear and nonlinear discrete operators and where $\vec{V} \equiv \{V_1, V_2, \ldots, V_q\}^T$ might represent $q \equiv np$ samples of the n elements of $\vec{U}$ on a p-point grid laid over $\Omega \cup \Gamma$.

Alternatively, $\vec{V}$ might contain expansion coefficients, e.g. Fourier coefficients, obtained from projecting the solution onto the aforementioned space of trial functions, e.g. complex exponentials. In any case, we can now assume that the boundary conditions have been incorporated into equation (4) and no longer need to be specified separately as in equation (2). This allows us to focus on equation (4) as a self-contained, continuous dynamical system. For the remainder of this example, we ignore the original spatial dependence in equation (1) and use a specific dynamical system, the Lorenz system [4] as our prototypical multiphysics model:

$$\frac{d}{dt}\left\{\begin{array}{c} v_1 \\ v_2 \\ v_3 \end{array}\right\} = \left\{\begin{array}{c} \sigma(v_2 - v_1) \\ v_1(\rho - v_3) - v_2 \\ v_1 v_2 - \beta v_3 \end{array}\right\}, \tag{5}$$

which represents a reduced-dimension model for weather with parametric dependence on σ, ρ, and β. For certain values of of these parameters, the Lorenz system exhibits chaotic behavior, inclding extreme sensitivity to initial conditions and a strange attractor.

To recast this as a multiphysics model, let us partition equation (**??**)such that

$$\vec{\mathcal{R}} \equiv \{\vec{a}, \vec{c}, \vec{g}\}^T \tag{6}$$

$$\vec{V} \equiv \{\vec{\alpha}, \vec{\xi}, \vec{\gamma}\}^T \tag{7}$$

which separates the Lorenz system into into three distinct but coupled subsystems with simple one-dimensional state vectors

$$\vec{\alpha} \equiv \{v_1\}, \quad \vec{\chi} \equiv \{v_2\}, \quad \vec{\gamma} \equiv \{v_3\}, \tag{8}$$

and scalar state equations

$$\vec{a} \equiv \{\sigma(v_2 - v_1)\}, \quad \vec{c} \equiv \{v_1(\rho - v_3) - v_2\}, \quad \vec{g} \equiv \{v_1 v_2 - \beta v_3\}. \tag{9}$$

For illustrative purposes, we imagine these submodels to be Air, Cloud, and Ground classes, respectively, in a weather model. In a richer, higher-dimensional example, the Air class might simulate fluid dynamics. The Cloud class might simulate droplet transport. The Ground class might track evolving boundary conditions.

Figure 1 provides a Unified Modeling Language (UML) class diagram depicting relationships between an Atmosphere Puppeteer and its three air, cloud, and ground puppets. The connecting lines denote relationships between the Puppeteer and its puppets. The open diamond adorning the Puppeteer end of the relationships indicates aggregation. In UML, the aggregated classes (the puppets) are *attributes* of the aggregating class (the Puppeteer). The language mechanism that faciliates aggreatation depends on the implementation language. In

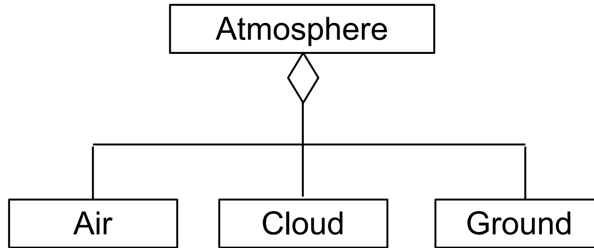

Figure 1: UML class diagram: aggregation of air, cloud and ground puppets into an atmosphere Puppeteer.

C++, one would typically make puppets data members. In Fortran 2003, the analogous mechanism is termed "derived type components."

To illustrate the need for forming joint quantities, consider advancing equation (1) using an implicit integration scheme such as the trapezoidal rule:

$$\vec{V}^{n+1} = \vec{V}^{n} + \frac{\Delta t}{2}\left[\vec{\mathcal{R}}\left(\vec{V}^{n}\right) + \vec{\mathcal{R}}\left(\vec{V}^{n+1}\right)\right] \tag{10}$$

where the presence of $\vec{V}^{n+1}$ on the RHS necessitates iteration (or linearization) when $\vec{\mathcal{R}}$ contains nonlinearities. One generally poses the problem in terms of finding the roots of a residual vector such as

$$\vec{f}\left(\vec{V}^{n+1}\right) = \vec{V}^{n+1} - \left\{\vec{V}^{n} + \frac{\Delta t}{2}\left[\vec{\mathcal{R}}\left(\vec{V}^{n}\right) + \vec{\mathcal{R}}\left(\vec{V}^{n+1}\right)\right]\right\} \tag{11}$$

where $\vec{V}^{n}$ is known.

A dilemma arises in finding the roots of $\vec{f}$ using Jacobian-based iteration methods. Consider Newton's method. Defining $\vec{y}^{m}$ as the mth iterative approximation to $\vec{V}^{n+1}$, Newton's method can be expressed as

$$J\Delta\vec{y}^{m} \equiv -\vec{f}(\vec{y}^{m}) \tag{12}$$

$$\vec{y}^{m+1} \equiv \vec{y}^{m} + \Delta\vec{y}^{m} \tag{13}$$

$$J_{ij} \equiv \left.\frac{\partial f_i}{\partial y_j}\right|_{\vec{y}=\vec{y}^{m}} = \left.\frac{\partial f_i}{\partial V_j^{n+1}}\right|_{\vec{V}^{n+1}=\vec{y}^{m}} = \delta_{ij} - \frac{\Delta t}{2}\left[\frac{\partial \mathcal{R}_i\left(\vec{V}^{n+1}\right)}{\partial V_j^{n+1}}\right]_{\vec{V}^{n+1}=\vec{y}^{m}} \tag{14}$$

where $\mathbf{J}$ is the Jacobian matrix, $\mathcal{R}_i$ is the RHS of the ith governing equation, and δ_{ij} is the Kronecker delta. Equation (12) represents a linear algebraic system. Equation (12) represents vector addition. In both equations, $\vec{f}$ and $\vec{y}$ contain data that is distributed across, and hidden within, multiple class implementations. Hence, equations (12)–(14) define an abstract calculus that is most naturally implemented via overloaded operations on objects [5].

Since the appearance of term δ_{ij} and the factor $\frac{\Delta t}{2}$ depend on the chosen quadrature scheme, we focus here on the calculation of the final bracketed

quantity in equation (14), which would appear in any Jacobian-based iteration process. For discussion purposes, we rewrite this quantity using the notation

$$\left[\frac{\partial \mathcal{R}_i\left(\vec{V}^{n+1}\right)}{\partial V_j^{n+1}}\right]_{\vec{V}^{n+1}=\vec{y}^m} \equiv \frac{\partial(\vec{a},\vec{c},\vec{g})}{\partial(\vec{\alpha},\vec{\chi},\vec{\gamma})} \tag{15}$$

The aforementioned dilemma presents itself in the need to calculate cross terms such as $\partial\vec{a}/\partial\vec{\chi}$ when $\vec{a}$ and $\vec{\chi}$ are hidden inside separate classes.

The dialogue that supports the Jacobian assembly can be paraphrased as follows:

> *Puppeteer to Air*: Please pass me a vector containing the partial derivatives of each of your governing equations' RHS with respect to each element in your state vector.
>
> *Air to Puppeteer*: Here is the requested vector ($\partial\vec{a}/\partial\vec{\alpha}$). You can tell the dimension of my state by the size of this vector.
>
> *Puppeteer to Air*: I also know that your governing equations depend on the Cloud state vector because your interface requests information that I retrieved from a Cloud. Please pass me a vector analogous to the previous one but differentiated with respect to the Cloud state information I passed to you.
>
> *Puppeteer note to self*: Since I did not pass any information to the Air object from the Ground object, I will set the cross-terms corresponding to $\partial\vec{a}/\partial\vec{g}$ to zero. I'll determine the number of zero elements by multiplying the dimensionality of $\partial\vec{a}/\partial\vec{\alpha}$ by the dimensionality of $\partial\vec{g}/\partial\vec{\gamma}$ after I receive the latter from my Ground puppet.

To complete the assembly of $\partial\vec{\mathcal{R}}/\partial\vec{V}$, the Puppeteer then holds analogous dialogues with its Cloud and Ground puppets. In each dialogue, the Puppeteer developer employs information available in the public interfaces of each puppet. See Rouson, Adalsteinsson, and Xia for a UML sequence diagram for a more detailed narrative of the dialogue and Appendices A.3 and B.3 of that article for reference C++ and Fortran 2003 implementations.

Known Uses

The Puppeteer pattern has been used throughout the collection of packages produced by the Multiphysics Object-oriented Fluid Environment for Unified Simulations (Morfeus) project[1] at Sandia National Laboratories and the City University of New York. It has also been used by Adalsteinsson et al. [1]. Langtangen and Munthe [?] described a similar idea, although not in the idiom of design patterns and not in the context of writing an abstract calculus.

[1] http://public.ca.sandia.gov/csit/research/scalable/morfeus.php

Related Patterns

- OPL : Layered
- Gamma et al. [2]: Mediator
- Rouson, Adalsteinsson and Xia [6]: Semi-Discrete

Author

Damian Rouson

References

[1] Adalsteinsson, H., B. J. Debusschere, K. R. Long, H. N. Najm Components for atomistic-to-continuum multiscale modeling of flow in micro- and nanofluidic systems *Scientific Programming* **16**:4, 297–313 (2008).

[2] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[3] H. P. Langtangen and O. Munthe. Solving systems of partial differential equations using object-oriented programming techniques with coupled heat and fluid flow as example *ACM Transactions on Mathematical Software* **27**:1,1–26 (2001)

[4] Lorenz, E. N. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences* **20**:2, 130–141 (1963).

[5] Rouson, D. Towards analysis-driven scientific software architecture: The case for abstract data type calculus. *Scientific Programming* **16**:4, 329–339 (2008).

[6] Rouson, D., H. Adalsteinsson and J. Xia. Design patterns for multiphysics modeling in Fortran 2003 and C++. *ACM Transactions on Mathematical Software* **37**:1 (2010).