Z2 Computer Solutions
Wayne L. Atchison
(303)-999-0701
email: Wayne@Z2cs.com
web: www.Z2cs.com

The Snippet Engine Methodology
White Paper
Written November 9, 1997
Edited March 21, 2008
Printed 3/10/2009

Abstract: A New Software Methodology For 21st Century Computers

The Snippet Engine Methodology is a new and innovative approach to multicore software development. The new Object Oriented Methodology and its supporting execution Engine dramatically change the way multicore software and cross-platform Internet systems are designed, managed, developed, tested, and maintained. The new Methodology works within any target framework, including Visual Studio and .NET, to greatly simplify the task of multi-threaded programming. The Snippet Engine Methodology is specifically designed for Thread-to-Thread synchronization between numerous multiprocessor platforms linked by the Internet. Such capabilities are essential for creating worldwide Distributed Database Servers, Command/Control, and Robotics applications. The Snippet Engine greatly reduces the cost of synchronized dual core software development on 21st century computers.

The Snippet Engine is a superior technology for software development. The Snippet Engine is both a software development methodology for multicore programming, and an execution Engine. The Snippet Engine is an innovative extension of the traditional Object Oriented approach, and greatly simplifies complex projects. The result is extremely fast, synchronized, and easily scalable multicore Distributed Database Servers and Client-side Applications.

Background:

In the early 1990s, when the Object Oriented Design and Decomposition (OODD) methodology first became popular, CPU-level execution-threads were only a new buzzword to UNIX and Windows programmers. As a consequence, the OODD methodology does not directly address the design and programming problems of synchronized multithreaded programming on networked platforms having multiple CPUs and multicores. Splitting up an object class so that different parts of it concurrently runs as lowest-level execution-threads on multiple CPUs is essentially left up to the programmers to figure out. However, the Snippet Engine allows projects to be designed so that its Design Objects are isolated as lowest-level execution-threads. Further, each Object can be synchronized with any other Objects running anywhere in the world.

Discovery begins by looking at the same thing as everyone else, but seeing something different.

The information, concepts, and intellectual properties expressed herein
are the sole property of, and are proprietary to,
Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

1.0 The Overview Description:

The Snippet Engine (**SnEn**) is both platform and language independent. The SnEn has several software innovations which extend and enhance the Object Oriented approach to software development. One such innovation is to take a **Thread-to-Thread Centric approach** to both design and implementation. Design-level Components are directly coded into independent and concurrently executing CPU threads. This means that code-level "thread-synchronization" is the same thing as Design Object synchronization. This allows the Designers to decompose the project's complexity much easier. Further, the more CPU-Cores that a platform has, the faster the application will execute. Essentially, the SnEn allows the project to be designed around Objects that are concurrently running as "Self-Aware Self-Managed Execution Threads".

Another innovation is that the SnEn is an OOD approach which produces verb-Objects. The project's nouns are used by the verb-Objects as they need them. But the SnEn's verb-Objects are more than just objects, they are enhanced by two more significant software innovations. **These two innovations expand the definition of "an object" so that SnEn verb-Objects are "self-aware" and are directly tied to the computer's lowest-level execution core.** By enhancing each verb-Object with these two innovations we go way beyond the traditional concepts of "objects" and create an environment for software development that is far superior.

The SnEn verb-Objects are coded using traditional coding techniques. The programmers may implement each verb-Object in C++, C#, Visual Basic, Java, or other languages. Further, designers and programmers may take full advantage of using Visual Studio and the .Net foundation capabilities.

2.0 Explaining The Snippet Engine's Innovations:

The SnEn calls the Design Components created by the design engineers and executed by the SnEn as threads: a "**Node**". The **SnEn Node** is the Methodology's target entity. Each Node encapsulates Designed Purpose, Information Management, Code, Component Aggregation, Inter-Component Command and Control, and Execution-Interrelationships. Each SnEn Node is itself a Design-level Object running as a single CPU-Thread of execution.

The design engineers think of SnEn Nodes as being "Knowledge Centers". Each Node "Knows How To Do Something", some verb that represents some aspect of the project. Each Node is autonomous, asynchronous, and self-directed. Nodes communicate and use each other in order to accomplish application-tasks. Some Nodes are higher-level Knowledge Centers which communicate and use other lower-level Knowledge Centers. Some Nodes may have a one-to-one correspondence to the application's formal Requirements Specification. That is, a Node may be created to be directly responsible for ensuring that one line-item of the Requirements Specification is accomplished. That

The information, concepts, and intellectual properties expressed herein are the sole property of, and are proprietary to,
Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

Node is a Knowledge Center which knows how to do it. Nodes are dynamically created, asynchronously executed, saved on disk, and deleted as required.

The Key is the Software Innovations of the Snippet Engine: the fundamental definition of "an Object" is both expanded and is directly tied to the computer's lowest-level of code-execution. By doing these two "software things" we have created a decomposition tool and multicore software development environment that is far superior.

The SnEn expands and enhances the traditional definition of an Object. This expanded definition allows each SnEn Node to encapsulate a Verb as an Object. Each Node is a verb-Object which is autonomous, and has a "life" all of its own, it is in essence "self-aware". Thus the SnEn's Design Components become execution threads which are independently scheduled, executed concurrently, responsible for ensuring that some key aspect of the project is fully accomplished, and are saved on disk. It is much more than just an expanded 'C++' class-object. A Node is more like a program within a program, but executes as a simple thread by the computer.

Further, a Node can be sent commands and data by any of the other Nodes, or by any external Internet program running anywhere in the world. This means that any application running anywhere in the world can send or receive commands and data directly with any lowest-level execution thread in the application. This capability is critical to sophisticated Distributed Databases and networked gaming and robotic applications. In doing this a Node retains its own awareness, as if it were "alive". A Node manages itself, schedules itself, receives and sends commands, manages its own data store, and operates both autonomously and asynchronously from all other Nodes, as if it were itself a program within a program.

Still further, a Node is also just like a single brain-cell of the application. The entire application is the sum total of all of the interaction and data retained by all of its Nodes. Thus, "thinking" is the process of one Node interacting with other Nodes, just like one brain-cell interacts with other brain-cells. Thus a SnEn application is a Database, as all of the Nodes combine to become one very smart Database Engine.

It is important to understand how this expansion of an object's definition is so much superior to the OODD approach. The SnEn allows the designers and programmers to put the project's coordinating logic (encapsulate the higher level "brains" that coordinate the Nouns) into autonomous self-aware smart computer-entities. Each SnEn Node is:

- an action-Verb that knows how to do something for the project,
- that manages itself autonomously as it acts and reacts on its own,
- that retains its own data and its own self-awareness,
- that operates like a program within the larger program,
- that functions like a brain-cell within a brain as it interacts with others,
- that coordinates with the other Nodes to do its task,
- that stays alive in case something happens to make it have to redo its task.

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

The point being stressed is that SnEn Nodes encapsulate the higher-level coordinating logic that they need in order to do their job. A foundational construct of all Nodes is that they know how to coordinate with each other. The "smarts" of the application is the interaction between the Nodes.

Conceptually, what this means is that building an application using SnEn Nodes is like being able to create an independent working-crew for each task-portion of the application. Each Node is conceptually just like a separate and specialized working-crew that knows how to do something. Imagine if the application designer could not only list all of the tasks (verbs) of the application, but then could assign a completely autonomous and trustworthy working-crew to do each of those tasks. Then imagine that each working-crew would by themselves dynamically interact with all of the other crews to figure out for themselves when to do their job, and then schedule themselves, and then asynchronously show up and get it done, and then tell the other crews waiting on them to proceed. Conceptually, you cannot execute an application with any more efficiency.

How It Works:

SnEn Design Components are a new innovation designed to fully match the primary direction of software development for the last 35 years: which is to put more emphasis on the design, so that the code is simpler to follow, logically bounded, conceptually encapsulated, absolutely predictable, having a reduction in interface complexity, and is reusable. **SnEn Nodes satisfy all of these goals**.

In the Design Phase the project's design engineers create Design Components that encapsulate areas of expertise. All of the project's Requirement Specifications are decomposed into specific areas of independent expertise. These designed components are independent Knowledge Centers. Knowledge Centers know how to ensure that some key aspect of the application is accomplished. Each Design Component affirms: "I Know How To Do This", and then does it.

Each SnEn Design Component is a unique Knowledge Center that will execute as a simple CPU thread. Even though executed as a thread, SnEn Design Components are "Smart Execution Threads". Each component is independently asserted, scheduled, and coded. Each component has its own encapsulated data store, command and control queue, and is only responsible for accomplishing one aspect of the project. Each component can be coupled to any other components to trigger and to cascade results. Because each Design Component is a single thread of execution, the software coding is both structured, compartmentalized, and thread-safe for concurrency. This has the advantage that each Design Component can be independently developed and tested. The more programmers assigned to the project, the faster the project will be completed.

The Integration Phase:

One of the primary advantages of using the SnEn is that it astronomically reduces the total number of software interfaces. This significant reduction in the number of software interfaces saves considerable time and cost in testing, integration, documentation, and

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

maintenance. As will be explained later, integration of the independently tested Design Components is "automatic". **There is no Integration Phase in a SnEn project**. This means most large software projects will experience a 40% reduction in development cost.

The SnEn is a new innovation in dual core / multicore software development and is vastly superior. **The SnEn is exactly what the software industry needs in the 21st Century**.

3.0 Each Snippet Engine Design Component Is An Innovative Encapsulation:

Using the SnEn the design engineers create components having both purpose and expertise. Snippet Design Components are an extension to traditional objects, and now encapsulate many more software facets and concepts. SnEn components are not complicated, but they may better be understood by not using the words "component" and "object" for awhile. For this reason, the following explanation will refer to SnEn Design Components as "things". Using the generic term "thing" will avoid prior concepts, and the reader may then better accumulate an understanding of exactly what a Snippet Design Component really is.

Each SnEn Design Component Encapsulates:

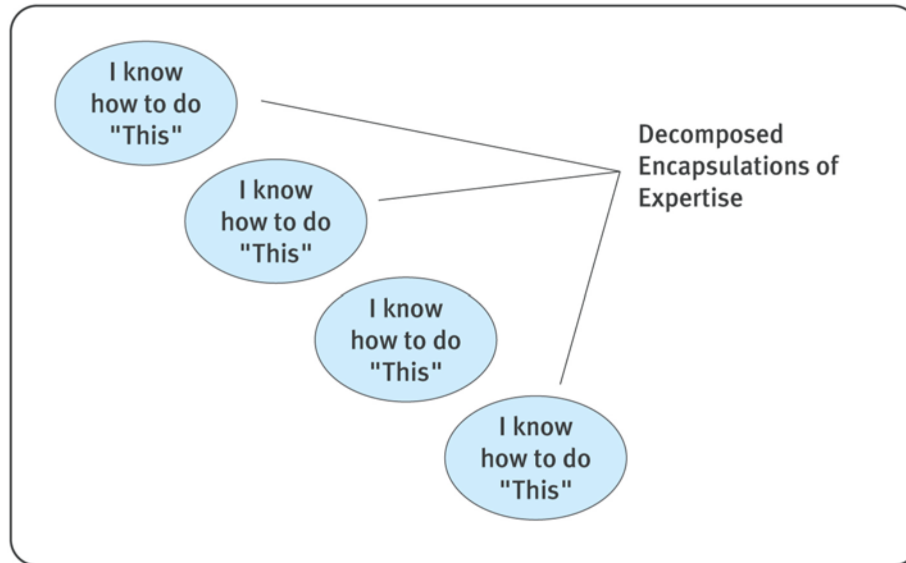
- Designed Purpose
- Component Aggregation
- Information (Data Store) Management
- Execution-Relationships
- Inter-Component Command and Control
- Synchronized Code

These six aspects of multithreaded programming are encapsulated into a single entity, a "thing", which is instantiated and managed by the SnEn.

Designed Purpose:

The SnEn Methodology has the design engineers decompose the problem into "things" that perform a designed purpose. A purpose is not a function or a task. A purpose is a Knowledge Center having expertise and the responsibility to ensure that some aspect of the problem is managed. A Snippet Component is a "thing" which is responsible for ensuring that its designed duty is accomplished.

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

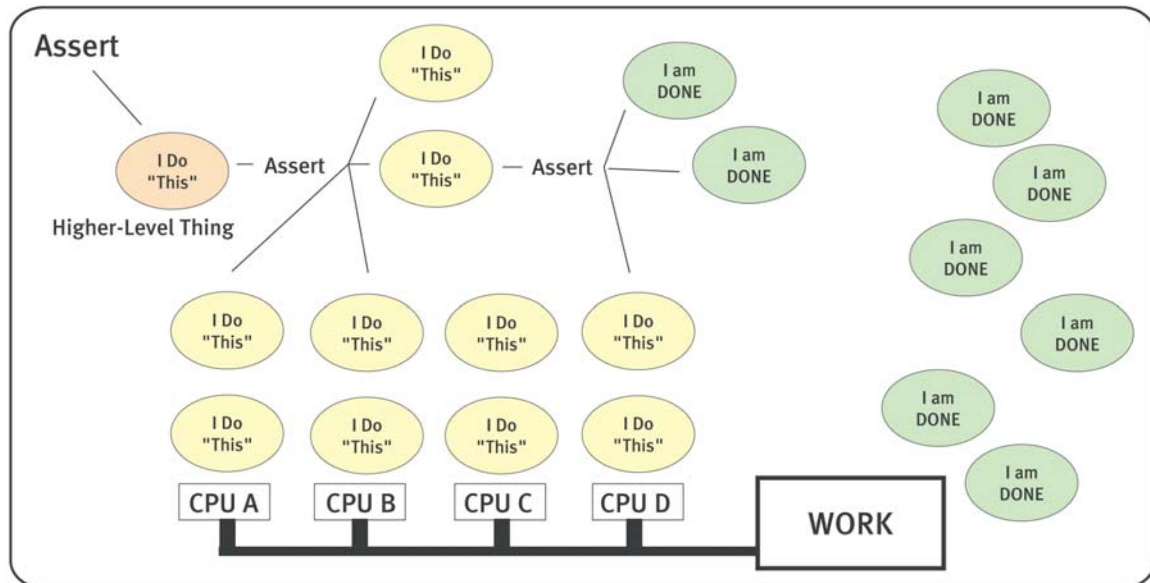


Design Phase Decomposes Into "Things" Having Expertise and Purpose
Such as “I Know How To Manage Sales Receipt Entries”

Each “thing” performs as an independent Knowledge Center, which fully accomplishes some aspect of the customer's requirements. Unlike OODD components, a designed purpose is not limited to encapsulate only a single computer, or a single set of data, or a single capability. For example, a designed purpose may be something very complicated: "Ensure that the histogram of voltage levels are current, and are displayed on monitors in both the space station and in Houston".

This example "thing" will execute asynchronously as its own CPU-thread, and is programmed with all of the expertise required to fetch the current voltage data, create a histogram, and present the histogram on the separate monitors. **However, this "thing's" internal code will not be large or complicated**. Rather its code will be simple coordinating logic, as it accomplishes its own designed purpose by asserting other lower-level "things" having their own designed purpose to accomplish each of the lower-level steps of the task. Thus each “thing” has a designed purpose, and that purpose may be a smaller sub-part of another “thing’s” purpose. Further, the "things" being designed are both design components and code components, there is no abstraction. Each "thing" will execute as a separate CPU-thread within the application, and will perform its designed duties when scheduled. Each designed "thing" will be independently programmed, tested, and certified to work.

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.



"Things" May Execute Concurrently, Each As a Thread, On Any CPU
"Things" Assert Other "Things" To Aggregate Work

Component Aggregation:
Each "thing" may couple and be coupled to any other "thing", just like "same-form-building-blocks" (Lego's Blocks) that fit together to form functional toys. Thus, new "things" are formed by aggregating other "things" together to form composite "things". Composite "things" will accomplish more expertise by coordinating the contributions of the lower-level "things" being aggregated.

This allows the designers to encapsulate complexity and expertise into "things" that provide specific capabilities. **Each "thing" is itself the aggregation of other "things", plus itself**. A "thing" collects the expertise of other "things", and then applies its own expertise, a knitting together, to form an even greater expertise. Thus a "thing" aggregates those "things" which directly progress towards its own designed expertise. Each "thing" encapsulates within itself all the "know-how" it requires to accomplish its uniquely designed purpose. Each "thing" executes to ensure that its designed expertise has been successfully accomplished, in the past tense.

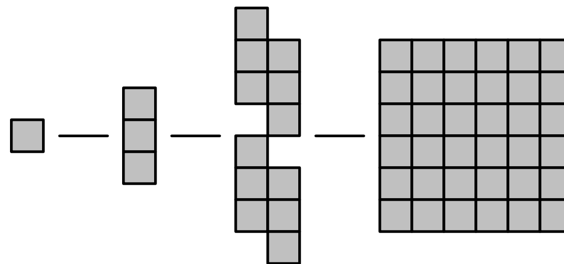
In order to coordinate the contributions of lower-level "things", the coordinating higher-level "thing" may require data-coupling into or with those lower-level "things". Thus, in order to accomplish the aggregation of "things" into higher-level "things", data-coupling between "things" may (this is a design option) be required.

The scope and type of data-coupling between designed "things" is completely within the control of the software design engineers. In some circumstances, the design engineers may decide not to allow any other "thing" to "see inside of" a specific Knowledge Center. In other circumstances they may decide to allow several "things" to each have "read-only-access" directly from the data store of another Knowledge Center. Thus, data-coupling between "things" may be as tight or as loose as deemed prudent by the

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

design engineers. The SnEn Methodology gives the engineers full flexibility and full control over data coupling decisions.

Regardless of the type of data-coupling chosen, the SnEn provides four standardized mechanisms for accomplishing data-coupling, so that all “things” data-couple together using the same standardized methods. This standardization dramatically reduces the number of unique software interfaces that must be separately documented, coded, tested, and maintained. Further, by providing only four standardized mechanisms, it is easy for a programmer to write the few lines of already debugged standardized code which provides the data-coupling to another “thing”.

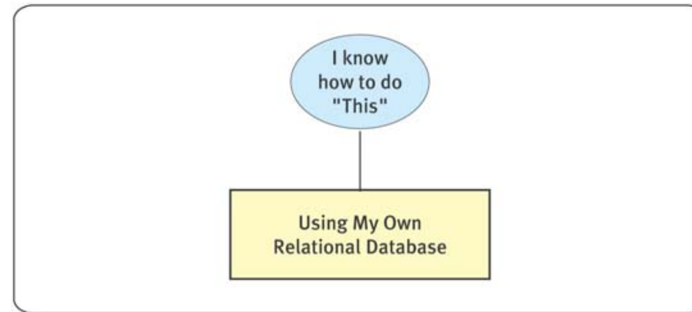


All "Things" Couple Using The Same Methods
New "Things" Are Created By Aggregating Other "Things" Together

This standardized method of data-coupling allows infinite expansion of aggregated expertise. There is no limit to which “things” may be aggregated together. Eventually a single "thing" is created to aggregate with a few other "things", so that, the single new "thing" is in effect the entire target application of the project. Assert this single highest-level "thing", and the entire application executes as fast as the number of CPUs permit.

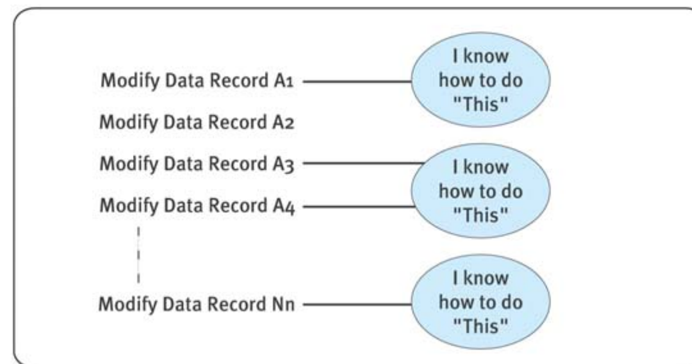
Information Management:
Each "thing" is an information repository having a predictable and consistent interface with all other "things". Each "thing" manages its own data store as a single-key Relational Database of sorted variable length records. There is no design limit to the size of a "thing’s" data store. Each "thing" manages its data store in exactly the same manner as all other "things". The data store of each "thing" is stored in shared RAM. The SnEn enables any "thing" to be able to access and use the data of any other "thing" with full synchronization. Shared RAM is locked and unlocked to avoid concurrency collisions. Just as with data coupling, the SnEn Methodology gives the design engineers full flexibility and full control over allowing or restricting access to each "thing’s" data store.

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.



Each "Thing" encapsulates its own Database Records

The SnEn also allows automatic triggering upon data record modification. Each individual data record, within a "thing's" data store, may be associated with any other "thing". When such a data record is modified, the record's associated "thing" is automatically asserted by the SnEn. This allows each data record to have its own intelligent change-manager, which knows how to handle the ramifications and manage the ripples each time that data record is modified. Allowing an interrelationship between any data-record and any other "thing" is part of the normal construct.



Each Database Record may have its own "Thing" to manage changes and rippling

What this means is that each data record can have its own unique execution path to both ripple its changes and to create new lower level data records. Thus, each data record may have its own unique intelligence, in which it has another "thing's" unique code, associations, and interrelationships with any other data records residing in any other "things" executing in any other computers anywhere in the world. This provides the design engineers infinite flexibility to create any level of intricate data relationships.

Thus, cascading data changes throughout complicated interrelationships with many other data stores located across networked computers is accomplished "automatically" by the SnEn. The programmer does not have to worry about how ramifications are rippled to other parts of the network. It is the SnEn that handles the details of creating and monitoring the designed associations and interrelationships. If one data field changes in one "thing", the SnEn will ripple that event to effect actions in any number of other "things" on many different computers anywhere in the world. The programmer does not have to code these complicated interrelationships, they are defined and setup during the

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

Design Phase. **It is this ability to cascade data changes through many other data stores that allows the application to exhibit superior intelligence**. **Imagine the value of being able to change a single Database record and automatically have that change reflected in hundreds of other Databases on computers around the world**.

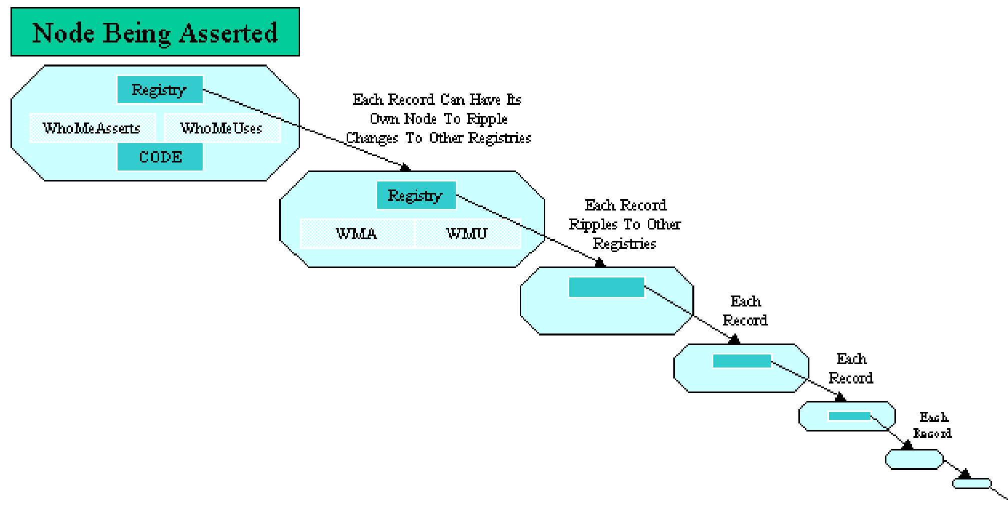
Notice that each associated "thing" may itself be another level of data store, and also have its own associated "things" cascade changes. Thus, the level of depth of data storage, and the level of intelligent ramification management within each "thing", is infinite.

This infinite capability enables Cybernetic Wit. Cybernetic Wit is achieved because the software can absorb new data, placing it into newly created levels of data store, and associate the new data with its own intelligence and cascading-interrelationships, resulting in an increase in the application's appearance of knowledge and intelligence.

As a quick summary, all "things" have **infinite expansion capabilities** in two directions:

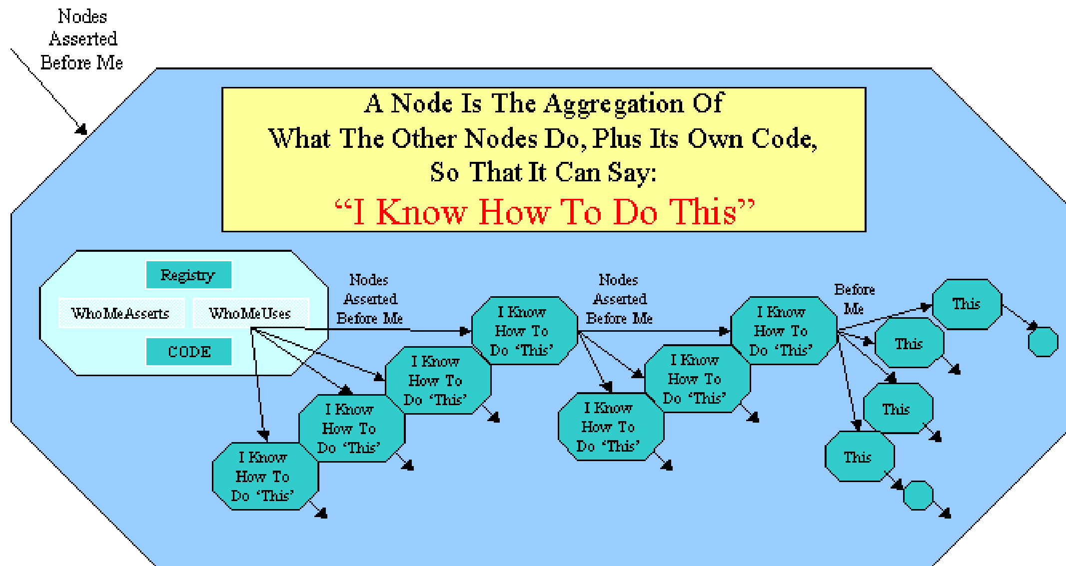
- **Inward depth within itself through data record associated "things"**
- **Outward expansion by aggregating with any other "things"**

Nodes Can Cascade To The Infinitely Small



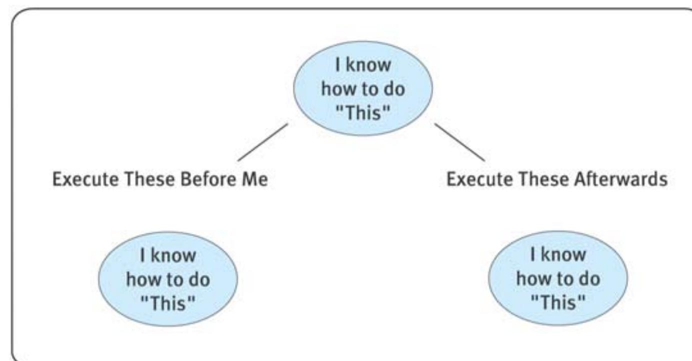
For Example: A Parts List To Build A Ship
Change the Assembly of any Part and that change automatically Ripples to change the Assemblies of everything based upon that Part

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.



Nearly all interrelationships are defined during the Design Phase, and **<u>are described using an easy to modify text script file</u>**. By keeping all of the design interrelationships described in a simple text file, even major design changes can be easily accomplished at any point in the project's development life cycle.

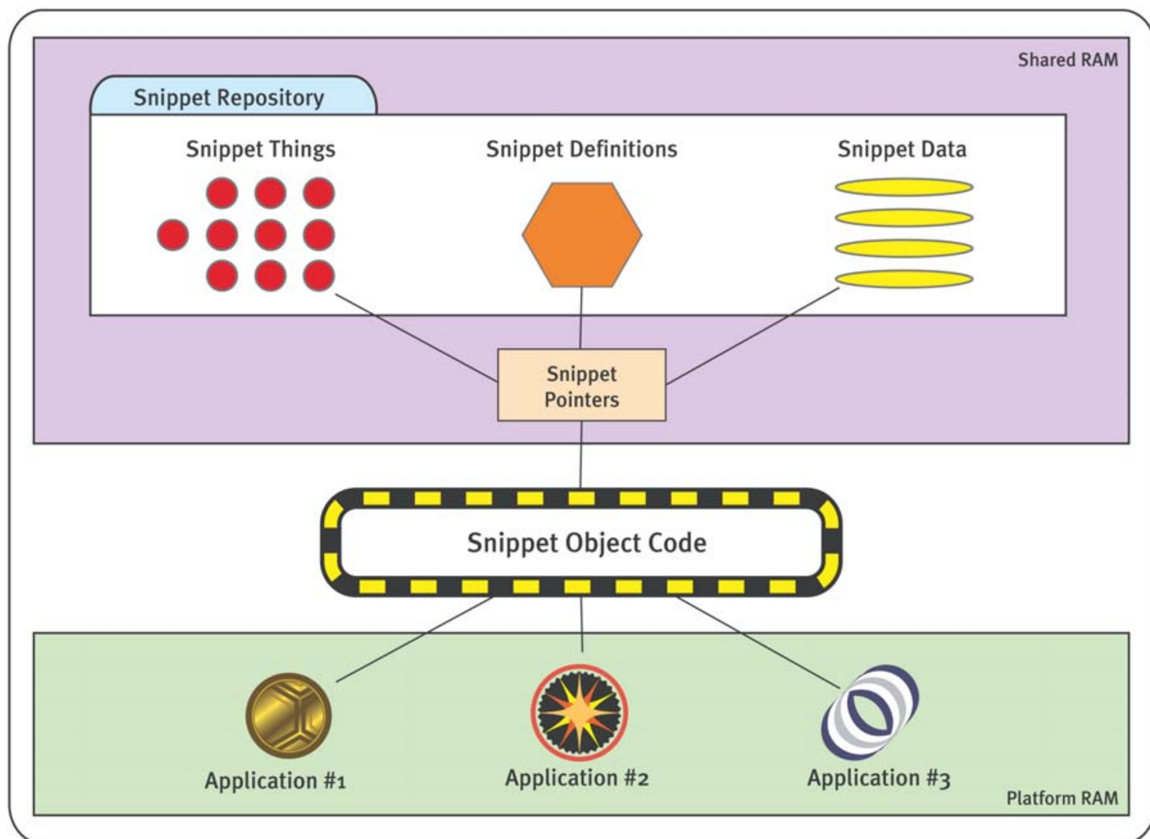
<u>Execution-Relationships</u>:
The SnEn creates, destroys, manages the state, and the scheduling of all "things". Some "things" are defined to be executed at intervals. Some "things" upon receiving a command or data. Some "things" are defined to be executed before other "things". Some "things" are defined to be executed after other "things". Any "thing" may be asserted by any other "thing". All "things" will automatically wait for the other "things" it depends upon to finish before it performs its own duties.



Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

A “Thing” does not “see the CPU” until its dependant “Things” have finished
Other “Things” are automatically Asserted when something “Changes”

The SnEn also manages the state and access to shared RAM. Each "thing" may allocate, read, lock, and write RAM that is shared by all other "things". All "things" may execute while using the same shared RAM concurrently, especially on platforms having multiple processors. So execution speeds are maximized, the SnEn manages shared RAM in a manner, which at the microprocessor level, avoids the operating system's Kernel. This saves at least 600 CPU cycles per event. All "things" may share data, application defined global areas, and contexts. **<u>How the shared RAM is used is completely up to the design engineers</u>**. The design engineers may decide to restrict some "things" from having any shared access.



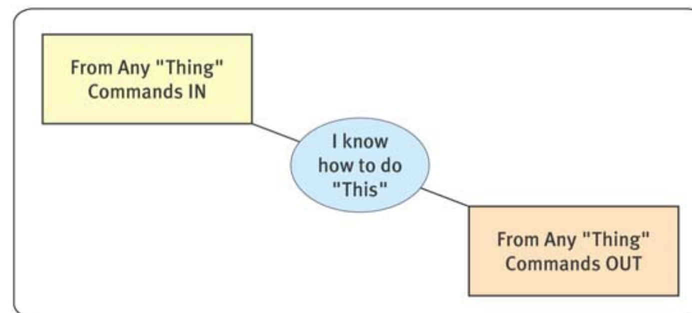
All “Things” Concurrently Share The Same RAM Environment

<u>This capability also means</u> that two or more applications may use the exact same shared RAM. This enables any Snippet Application to reference, use, command, and couple with any of the "things" from any other concurrently executing Snippet Application.

<u>Inter-Component Command and Control</u>:
Each "thing" is also a command and control center, having a predictable and consistent interface with all other "things", using exactly the same mechanism. Any "thing" can send commands and data to any other "thing", even across the Internet to “things”

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

running on networked computers anywhere in the world. Likewise any "thing" can receive commands and data from any other "thing" running anywhere in the world. Thus, **threads talk directly to threads running anywhere in the world**. The SnEn enables and monitors all communication between "things", so that any "thing" may asynchronously schedule and be scheduled by any other "thing". Essential for remote gaming and robotics, **any execution thread may synchronize commands and data with any other execution thread on any remote computer running anywhere in the world.** Sending commands and data is all handled by the SnEn, the programmer only needs to include a few lines of standardized code.



All "Things" receive and send commands and data in exactly the same way

Synchronized Code:
Each "thing" may optionally have separate code which executes during its creation, its deletion, prior to its dependencies, and after its conclusion. A "thing" can be programmed in any language, such as C++, and the programmers may use Object Oriented Programming class abstractions to program "things" in the project.

Each "thing" may synchronize its own execution with any other "thing" on any other computer. All "things" synchronize execution with each other using the exact same method. Thus, any "thing" may be designed to execute in a synchronized "lock-step-fashion" with any other "things".

For example an application may have a requirement to: ensure that all threads on all computers dealing with moving the game pieces finish first, then allow all threads on all computers which find the battles to execute next, then allow only the threads which perform the battles to go next, and so forth. Synchronization is defined during the Design Phase, and is implemented by a programmer within 15 lines of standardized (already debugged) code. Thus, even extremely complicated gaming is made easy.

A "thing's" code allows it to establish and remove associations and interrelationships with other "things", to process command and control messages, to manage its own Data Store, to perform its designed duties, to coordinate the aggregation of other "things", and to use the application's User Interface. Thus, even the lowest level "things" can use the GUI to show status, ask questions, and display diagnostics. For example, the board-level

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

"drivers" can tell the user that the internet card has failed, or that is has no protocol binding for communications.

4.0 Benefits Of The Snippet Engine Methodology:

Following is a partial list of the advantages of using the SnEn:

A. There is no abstraction between Design Components and Code Components. Thus, no time is wasted to convert the designer's class-abstractions into thread-level concurrent code-segments. This also means that "inter-Thread-synchronization" is the same thing as SnEn Node-synchronization which is also the same thing as Design Object-synchronization.

B. The SnEn automatically instantiates all Nodes when they are needed the first time. The programmer does not have to worry about how or when to get their Node into existence. Nodes may delete themselves when finished.

C. Each Node accomplishes the same result, regardless of who, when, or why it is asserted. All Nodes accomplish their designed responsibilities, or else they are not finished yet.

D. Programmers are focused on only generating their Node's code. They are coding a single CPU-thread, independent and autonomous, to ensure a designed purpose is accomplished. Dependencies on other Nodes which must execute first are automatically handled by the SnEn. When the Node is finished with its own execution, those Nodes which need to execute afterwards are also automatically asserted by the SnEn. Thus, the programmer's attention is directly focused on only the lines of code within their own execution thread.

E. All Nodes use the exact same Command Interface and Data Store access methods. All that the programmer needs to know is the format of the command packets and data records, which are defined during the Design Phase.

F. Nodes have no calling sequence like a function. All Nodes are asserted into execution using the exact same mechanism. Asserting a Node to do its job is simple and is always done in the same way. This simplicity dramatically reduces the number of software interfaces that must be defined, programmed, tested, and documented.

G. Nodes are not called like a section of code, rather they are told to accomplish their task asynchronously. Nodes are asserted because their designed purpose is required, and they may be asserted in any order. Interdependencies between Nodes are automatically handled by the SnEn.

H. Nodes do not pass arguments to each other like a function. Variables are passed between Nodes by either sending commands, or by setting up a designed shared RAM context. Sending a command is often preferred as the receiving Node is immediately asserted in order to handle the incoming command packet. Thus the programmer is relieved of having to understand inter-Node dependency and ramification complications when key variables change. All a programmer needs to do is to "post" the changes to the Node which knows how to handle the changes. Since the receiving Node is executed as a separate thread, the programmer can literally "post and forget".

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

I. Nodes which need to fetch data from other Nodes may do so directly, without posting commands. Data Stores reside in Shared RAM. Because all Nodes both store and retrieve data in exactly the same way, the programmer uses a standardized code-template (already debugged) to directly fetch data from any other Node. If necessary, a Node's Data Store may be locked and then unlocked by any other Node. This type of coupling is very fast.

J. All Nodes have only one State-variable, and this variable can have only five different values. This dramatically reduces the number of software interfaces, which dramatically reduces the number of test cases, which dramatically reduces the volume of documentation required in the project's life cycle.

K. Each Node can be independently tested, and most Nodes may even be certified to not have any bugs. The concept of code having hidden bugs is a result of not being able to test all software conditions. However with Snippet Nodes, because they are narrow scopes of autonomous logic with their own data store, it is possible to isolate a Node's execution-context in order to conduct stand-alone-testing which will test all software conditions. Nodes may be certified “bug-free”.

L. Nodes are self contained, which helps in maintenance and future enhancements.

M. Most Nodes are immediately reusable in future projects. Thus, the cost of future software development projects is dramatically reduced.

N. Nodes may execute on any available microprocessor. Since each Node is its own thread of execution, the entire application may be massively solved in parallel by executing on computers with large numbers of microprocessors. The more available microprocessors, the faster the program will run.

O. A new Node's integration into the application is “automatic”. There is no need for a separate Integration Phase in the project's life cycle. In order to thoroughly test a Node, the programmer will create an environment which includes all of the other relevant Nodes needed. Programming is best performed starting with the lowest level Nodes, and working upwards. Testing of higher level Nodes do not need to mimic lower level Nodes, but may simply include them as real code. Thus, the application is being integrated together as the project progresses, and is being fully integration-tested each time a stand-alone-test is conducted.

P. Because the project is developed from the bottom-up and each Node is independently stand-alone-tested, the more programmers assigned the faster the project will be finished. This allows the software managers to more effectively manage the project’s cost and delivery schedule.

Q. Any Node may use the application’s User Interface to interact with the user. Thus even low-level code may have its own debugging, diagnostics, error reporting, and context help GUI.

R. Any Node may talk to and synchronize with any other Node running anywhere in the world. To the programmer, sending a command or data packet directly to a Node running on another computer is done the same way as sending the packet to a “local Node”. Thus, any execution thread may directly communicate with any other execution thread, even on a completely different computer. Creating applications that share synchronized data and capabilities between networked computers is very easy.

S. Nodes **Are The Database Server**. Each client-browser can communicate directly with its own individual Node, so that, each client-submenu can individually and

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.

concurrently work directly with only the small portion of the whole Database it is using. Thus, each client-access-transaction is automatically distributed and isolated simply by the manner in which the SnEn works. This dramatically reduces the chance of concurrent data access.

T. **Scalability**: Because of the way that the SnEn works internally, a system can be easily scaled in two ways:
   1.) By just adding more CPUs to the existing Host platform: the SnEn automatically uses more CPUs as soon as they are online.
   2.) By adding more Host platforms: each SnEn EXE can communicate with other SnEn EXEs to distribute the Database across as many platforms as required.

5.0 Conclusion:

The Snippet Engine Methodology creates solutions which are specifically targeted to take full advantage of 21st Century multithreaded multiprocessor networked platforms synchronizing over the worldwide Internet. Projects are designed faster, coded faster, and tested faster, resulting in a significant reduction in development cost. Snippet Nodes encapsulate expertise, so that most Nodes are reusable in future projects. Both maintenance and future enhancements are easily accomplished.

The Snippet Engine is not theoretical, it is now being used to develop commercial applications. The benefits of using the Snippet Engine are astronomical. **Anyone concerned about the high cost of software development should feel compelled to further investigate this innovative new technology.** The Snippet Engine is available to use on your very next multicore software project.

Property of Z2 Computer Solutions, © 1997 - 2009. All rights reserved.