SOFTWARE ARCHITECTURAL STYLES

Sean Boyd, Mark D'Adamo, Christopher Horne, Nolan Kelly, David Ryan, Nairn Tsang

SENG 403 - W2013 Paper Project (Group 4)

Table of Contents

Introduction to Software Architectural Styles

Software architectural styles are patterns or frameworks that have been developed as 'general solutions' to common problems that arise in the software development process. They generally specify or define the components and connectors that compose a solution and their relations – usually a high-level specification of the structure of a program.

Message Bus Architecture

The message bus architecture is a message oriented middleware approach to centralizing communication within an application (or between a series of them). Analogous to the hardware bus architecture used to link physical computer resources, the message bus offers routing and message passing support between software systems or subsystems. It is loosely coupled to the components which utilize it, as it operates independent of their implementation so long as they match the interface [22].

The message bus is very much tied to its message passing mechanism. The two topic-based forms of the publish and subscribe architecture – list and broadcast – lend themselves very well to the message bus architecture [21]. List-based pub/sub keeps subscription information on the bus – it maintains a list of subscribers for each topic, and whenever a message is passed to the bus, the bus will copy and send the message to every subscriber of the message topic. Broadcast-based, on the other hand, sees the subscriptions stored on the node side of the relationship. A message sent to the bus is forwarded indiscriminately to ALL connected nodes, and messages of unwanted topics are simply filtered out and ignored by the recipient applications [23].

Service oriented architecture lends itself very well to utilizing a message bus architecture – facilitating the subscription of service providers to service requesters [24]. By providing a service registry and repository, the message bus can permit numerous and highly varied services to serve each other without needing extensive connection logic in the respective service applications.

Pros:

Improved Modifiability [21] – Applications added to or removed from the system no longer affect any other components; so long as they follow the message bus specifications integration with other components is seamless. New messages can be added without breaking old ones. Objects of any type can be encapsulated and delivered via the bus [22].

Reduced Complexity [21] – Message senders now only interact with the bus instead of all message recipients; likewise, the message recipients need only interact with the bus instead of all senders.

Improved Performance [21] – With all routing and message passing handled by the bus, performance is bound to bus capacity and throughput and freed from the constraints of potentially haphazard message passing protocols.

Improved Scalability [21] – Adding additional services or applications to the bus is an O(1) operation, limited only by the capacity of the implemented bus.

Cons:

Increased Complexity [21] – Integration becomes more complex with a message bus because all components accessing the bus must be shaped to fit its integration. Additionally, it is a shared resource and must address concurrency and queuing concerns.

Lowered Modifiability [21] – Changes to the bus interface must maintain backwards compatibility, or every component that utilizes the bus will have to be updated in order to support the new interface.

Lowered Security [21] – A broadcast based message bus offers no privacy without some form of encryption, as messages are delivered indiscriminately to all connected nodes. [23]

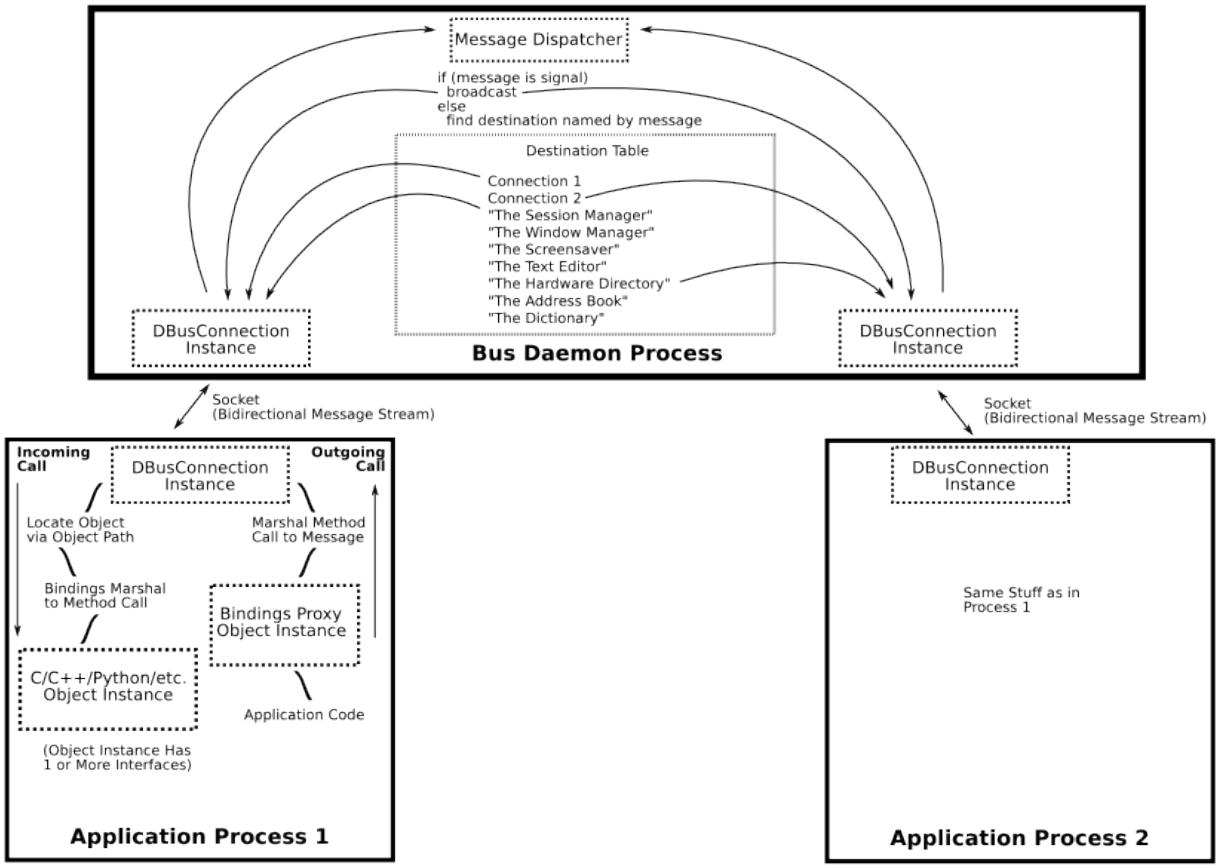
Low Downtown Tolerance [21] – The bus becomes a single point of failure for all communication across the application(s). Logic is not necessarily implemented for applications to manage their own messages while the bus is out of service, which can lead to message loss and failures of data integrity.

Example – D-Bus

D-Bus is an open source, unix-based tool for interprocess communication (IPC) that utilizes the message bus architecture. It consists primarily of: [19]

- The libdbus library, which facilitates the message scheme and one-to-one communication
- The message bus daemon, which can connect to many simultaneous applications and provides routing
- Bindings to facilitate a wider variety of frameworks and systems

In particular, D-Bus was designed to address communication between applications in the same desktop session, and between a desktop session and the operating system. D-Bus uses both the call-and-return and event-based varieties of message passing [19] between applications to facilitate these concerns.



[25] – The D-Bus Concept Diagram

Client / Server Architecture

Client-server architecture is one of the two prevailing models for network programming, the other being peer-to-peer. Almost every application that requires some sort of communication between clients uses one of these two architectures. In the client-server architecture, each entity on the network assumes the role of either a client or a server. Servers are powerful computers or processes dedicated to managing a set of resources ([17], page 18). They are ideally active at all times to service requests made by clients in order to provide a set of resources or functionalities. Servers can service many clients at a time and regulate their access ([18], page 261). A client is a single user host system such as a PC or workstation with user applications that initiate contact with the server in order to make use of its resources ([17], page 10). These resources can include data, CPU's, printers, and data storage devices. Clients never communicate directly with each other; they are only capable of indirect communication through the server. Clients and servers are located physically on different computer systems and require remote access to communicate ([18], page 261). Clients and servers are loosely coupled systems that communicate through a message-passing mechanism. Ideally client-server software is independent of hardware or operating system platforms.

Client-server architectures can be classified into many different types, including 2-tier, 3-tier, fat client, thin client, etc. The n-tier architecture is a model in which presentation, application processing, and data management functions are logically separated. Fat client refers to clients that run nearly everything involved with the client-server architecture. The client is responsible for a lot of the computations and processes that are needed to perform a particular task while the server is responsible for very little. In contrast, there is a thin client in which the server is responsible for a vast majority of the work and the client has very little responsibility ([18], page 262-265).

Pros ([13])

- **Centralization of data:** All data is on a single server, simplifying the organization and management of data. This also makes it easier to backup data as well as modifying/editing ([17], page 12).
- **Security:** Requests made by clients can be monitored and logged. Validation can be enforced with requests to ensure that only those clients with valid credentials can have access to specific services.

Cons ([13])

- **Congestion:** If an increasing number of clients wish to communicate with the server, this will increase the server's workload, thus reducing network speeds.
- **Cost:** Cost of setup and maintenance are generally high.
- **Not Robust:** If the server fails, the clients will no longer be able to function. Clients will not be able to interact at all until the server is restored.

Example – Microsoft SQL Server

An example of a real work application that makes use of the client-server architecture is Microsoft's SQL server. As the name suggests, SQL server is a database application in which the database resides on a server. The server is a relatively large computer in a centralized location. Clients can connect to the server and make requests for data using a number of different applications. Having the database centralized on a server makes it much easier to maintain an up to date collection of data. Clients do not need to continually update local copies of the database to ensure consistency. SQL server provides security options as well, requiring authentication to access the database or to make changes to the data if enabled ([14], page 1-5).

Object-Oriented Architecture

The object-oriented architecture style is a set of design principles in software development that focuses on breaking down a system into individual and reusable components, or objects. Objects typically consist of data fields (ie. attributes) and procedures (ie. methods). Objects are usually instances of classes, and a program can be considered to be a collection of objects interacting with each other. This is in contrast to the more conventional procedural programming, where a program is more like a list of subroutines. Objects are separate, independent entities that are loosely coupled. They communicate with one another through interfaces, method calls, and sending and receiving messages.

Fundamental Features ([1], Object-Oriented Architecture Style)

- **Inheritance:** Objects can inherit the characteristics of other objects. This means an object can inherit the functionality of another base object, and even override that functionality's behaviour. Changes to a base object are also propagated down to lower objects.
- **Encapsulation:** The internals of an object can be hidden from others so that only that object can manipulate its own state and variables. This protects the integrity of the object and prevents users from setting the internal data of that component to an invalid or inconsistent state.
- **Abstraction:** Abstraction refers to breaking down a system into logical components that can perform some sort of work and communicate with other objects in meaningful ways. It can help reduce complexity.
- **Polymorphism:** Polymorphism refers to giving an object multiple forms. It allows you to make changes to a base object and extend it to different types, while still keeping the base object interchangeable with its other forms.

Pros ([1], Object-Oriented Architectural Style)

- Reusable
- Extensible
- More closely related to how we view world objects
- Increased cohesion
- Easier testing through encapsulation

Cons

- Strong coupling between superclasses and subclasses. Swapping out superclasses can often break subclasses.
- Not all problems can necessarily be broken down into clearly defined objects.
- Not necessarily appropriate for smaller, less complex projects.

Example – Cocoa [20, Model-View-Controller]

A good example of a practical application of object-oriented programming (OOP) is Apple's native API, Cocoa. Cocoa is implemented in Objective-C, an object-oriented programming language. Cocoa also greatly emphasizes the Model-View-Controller (MVC) architecture pattern, which is strongly tied to the object-oriented paradigm.

As the name suggests, in MVC there are three main objects: Model, View, and Controller. The Model consists of the application data and logic, and the View requests information from the Model and displays it as some visual representation (ie. GUI). The Controller acts as an intermediary object and passes information between the Model and the View. Each object is independent from the others, which gives the system a lot of flexibility and makes the components highly reusable, which is one of OOP's biggest advantages. Changes to one component do not necessitate changes to another. A simple example would be using different GUIs for the same system. In this case you could reuse the same Model and Controller, and only have to make changes to the View, saving a lot of time and effort. Virtually all the apps developed for Apple's app stores strongly adhere to this idea of modularity and reusability.

Layered Architecture

The Layered Architecture Style is focused around dividing software functionality into distinct layers that interact vertically. It is dependent on message passing between layers and clearly

defined functional layers. Each layer can only send or receive messages to the layer directly above or below it.

Pros ([1], Layered Architectural Style)

- **Abstraction**: Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.
- **Encapsulation**: No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.
- **Clearly defined functional layers**: The separation between functionality in each layer is clear.
- Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.
- **High cohesion**: Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.
- **Reusable**: Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.
- **Loose coupling**: Communication between layers is based on abstraction and events to provide loose coupling between layers.
- **Design Pattern Support:** The layered architecture is supported by many software design patterns including MVC and other Separated Presentation patterns.

Cons

The only major disadvantage involved in layered architecture involve difficulties structuring systems as layered systems. Specifically, restricting communication to adjacent layers, and keeping coupling between layers reduced. ([2], page 12) If software systems can be successfully designed as layered systems, there are no disadvantages.

Example

An example of a layered software system is this process control for chemical production processes. The left side represents the software side, and the right represents the hardware side.

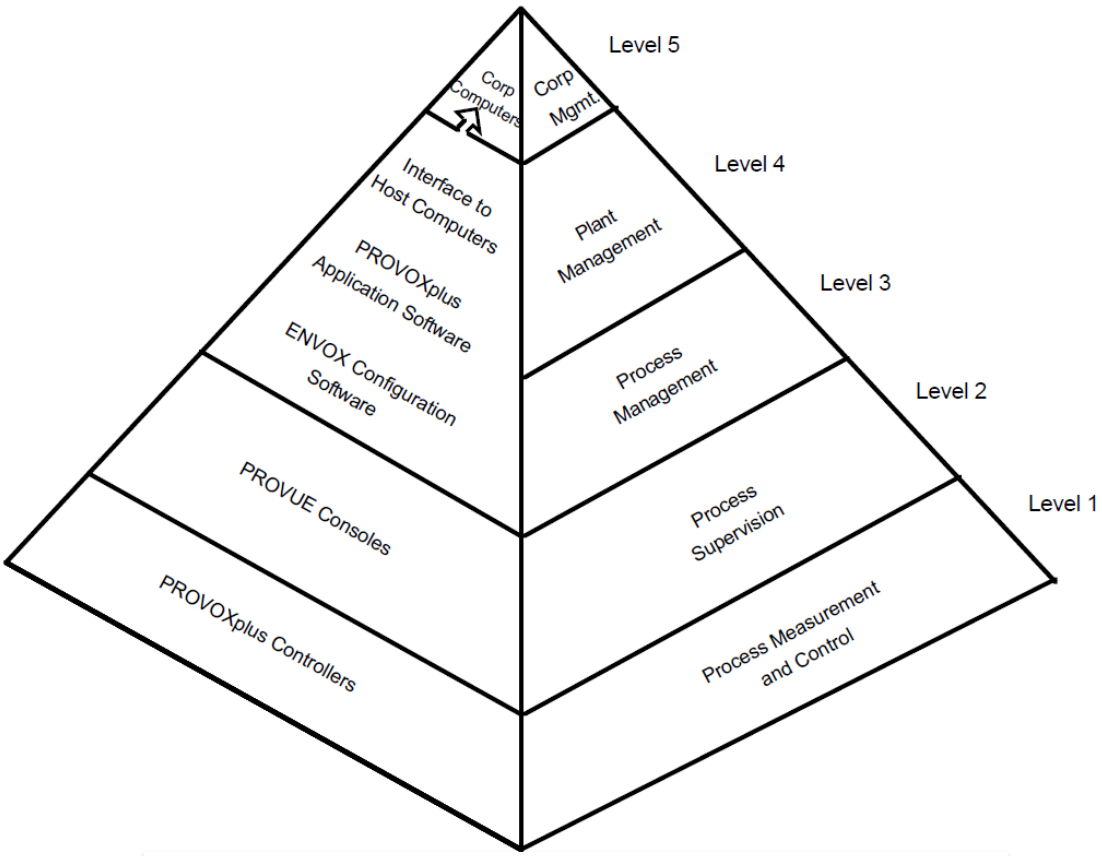


Image Source: [2], page 28

• Level 1: Process measurement and control: direct adjustment of final control elements.
• Level 2: Process supervision: operations console for monitoring and controlling Level 1.
• Level 3: Process management: computer-based plant automation, including management reports, optimization strategies, and guidance to operations console.
• Levels 4 and 5: Plant and corporate management: higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

([2], page 28-29)

N-Tier Architecture

N-Tier architectural style is the separation of the functionality of software into different tiers. Each tier can be located on a separate physical computer. These systems are component-oriented, and usually use platform specific communication methods instead of message passing. Each tier in the system is completely independent from all tiers except those immediately above and below it. Communication between the tiers is not synchronous because this allows the addition or removal of tiers. ([1], N-Tier/3-Tier Architectural Style)

The N-Tier architectural style may also be known as the 3-Tier architectural style because in its most common incarnation there are three tiers, each located on a separate server.([1], N-Tier/3-Tier Architectural Style) This architectural style has close ties to the Layered architectural style, and they may be used at the same time in which case a tier may host a particular layer. The N-Tier architectural style is used if the layer in question requires an amount of resources that could lead to decreased performance of other components of the system if it were to be run on the same machine. It is also used if a layer needs access to sensitive information which is not needed in other layers, as it allows the data to be stored in a place where no other layers can possibly access it. ([3], The N-Tier Model)

Pros ([1], N-Tier/3-Tier Architectural Style)

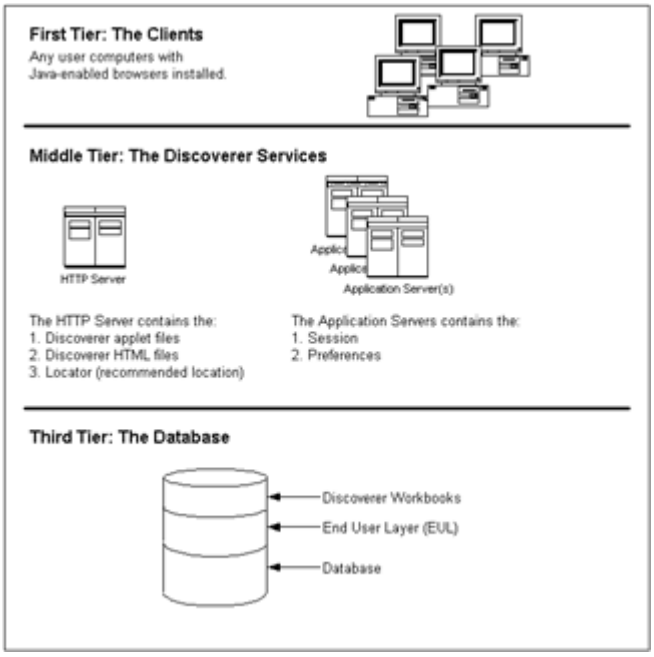
- **Maintainability**
  Each tier is independent of the other tiers, which allows changes and patches to be performed on a part of the system without affecting the application as a whole. Maintainability is an extremely important consideration in any software project, so this is a large benefit.
- **Scalability**
  Because tiers are designed based on the application's layers, scaling tiers is a simple process. The n-tier system is modular, allowing for easy scaling. This is another important benefit.
- **Flexibility**
  Since each tier is mostly independent, they can be managed or changed independently, allowing for a flexible application.
- **Availability**
  Since a tier is responsible for one component, it is easy for the application to increase the amount of a certain component, allowing higher availability.

Cons ([4], Goals of a Good N-Tier Application)

- **Bloat**
  Since communication between tiers is needed and they are on different machines, programs can become much larger than they otherwise would be.
- **Inflexibility**
  Although in most cases the use of the N-Tier architectural style increases flexibility, there are some instances in which having to communicate between multiple layers necessitates data be accessed in a specific way which might not be what is wanted.

Example – Oracle Discoverer Tool ([5], 1.2.1)

A real example of an N-Tier application can be found in Oracle's Discoverer tool. It is an application for analyzing data in a database and as show below it is a 3-tiered application.



([5], Figure 1-1 Three tier architecture of Discoverer Plus)

Component Based Architecture

Component Based Architecture (CBA) is an architecture that focuses on decomposing software designs into functional or logical components with their own methods, events and properties. The components are loosely coupled and reusable to provide modular programs that can be tailored to fit any need. As such the CBA can provide a level of abstraction higher than that of Object Oriented Programming Architecture (OOPA) and does not focus on issues such as communication protocols and shared state.

Components:

A component can be any software package, web service or module that encapsulates a related set of functions or data ([6], Software Component Architecture). The idea is to have the component fully encapsulated to the point that one can make full use of said component without requiring knowledge of its implementation. It provides this functionality to the rest of the system by use of a *provided interface*. A provided interface specifies what a component can provide to other components in the system. An example of this are user-interface components such as buttons and grids. The button can provide a related set of functionality such as event-on-mouse-click, event-on-mouse-hover, etc. It does this without the user needing to know how exactly a button functions.

The idea of all this is to develop the software components to the point of being independent or *loosely coupled* with other components in the system. In an ideal scenario the only thing a component should require to operate is a system mechanism that provides an environment in which the component(s) can execute. While this ideal, it is not always possible and as such components can depend on other components to execute by adopting a *used interface*. A used interface specifies what a components needs to operate.

Pros ([7], What are the advantages and disadvantages of plug-in based architecture?)

- **Ease of deployment**. As new compatible versions become available, you can replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost**. The use of third-party components allows you to spread the cost of development and maintenance.

- **Ease of development**. Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- **Reusable**. The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- **Mitigation of technical complexity**. Components mitigate complexity through the use of a component container and its services. Example component services include component activation, lifetime management, method queuing, eventing, and transactions.

Cons

- **Message Handling** needs to be predefined for the components. Possibility it could be a limiting factor.
- **Reliance on Third Parties**. If your components come from a third party of some sort then you will be at their mercy for updates and changes to the component.
- **Complexity.** While it is designed to reduce complexity of systems it introduces a different type of complexity in terms of component-to-component interactions. Ex. some problems may only surface with certain combinations of components.
- **Testing.** Can be difficult if the component doesn't come with it's own execution environment. (More of a third party component problem)
- **Second System Syndrome.** Depending on how complex the components themselves are you can end up with a platform within a platform within a platform type problem.

Example – Gaming

Then and Now:

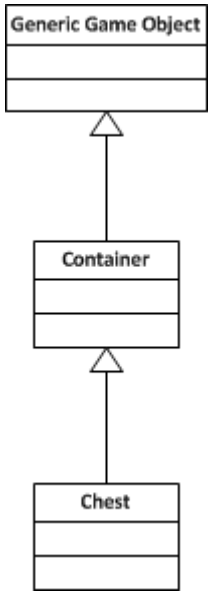
Doom and Quake. You know these names. I know you do. Back in the day you weren't a person unless you had experienced the majesty that is Doom and Quake. While not being known for having a lot of "content depth" but they were known for having many things to kill. They were however known for being at the very forefront of game programming technology. Adaptive tile refresh, raycasting, binary space partitioning (which *Doom* became the first game to use), and surface caching (which was invented for *Quake)* were all programming techniques that were either first pioneered or even invented for these games. Both games were even programmed in Objective-C ([8], Why Objective-C is Cool) which was a language that nobody else was using at the time (everything else was C/C++ or some sort of ASM). These games were paragons of their kind

Flash forward to today's gaming era and we find we are in a very different place. Games are now massive, extremely resource intensive programs. They can measure on the

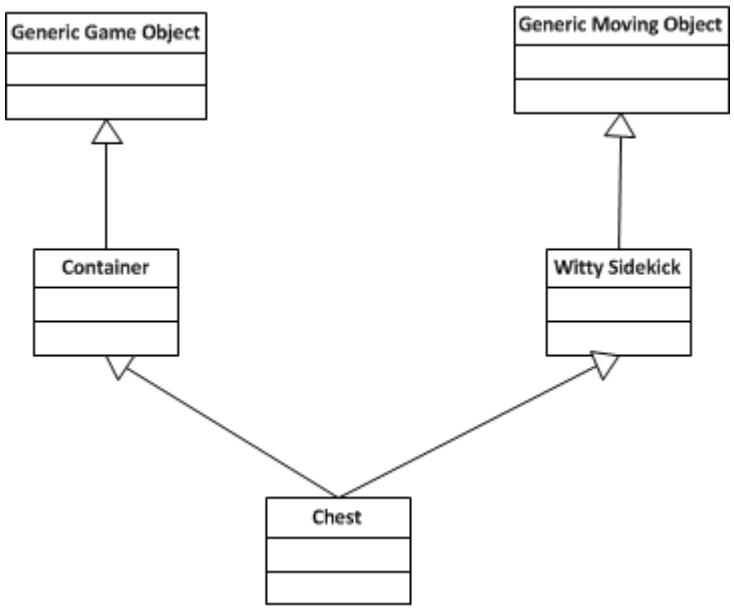
scale of 25GB for a single title and can offer a seemingly endless pit of content, depth and even more things to kill. Suffice it to say games today are complex. Very complex. And therein is our problem. A lot of the OO programming techniques used by games back in the day are no longer feasible anymore. The biggest of these offenders is Inheritance Based Programming ([9], Prefer Composition over Inheritance [Online]).

Inheritance Based Programming:

What is IBP? Well it's like this. Let's say you have a chest object in a game. The chest is, let's say, a derived class of a Container which is a derived class of a Generic Game Object.



This is pretty straightforward and would have yielded a perfectly good chest to use and back in the day this would have been awesome. Not today. Today gamers will not be satisfied with a simple chest that simply opens and closes all simple chest like. No, they want that chest to attack them or follow them around like a sidekick making witty remarks every now and again. The problem is that none of the objects in our chest hierarchy have any functions, methods or parameters for making the chest an entity or movable. So we need to add it by attaching another hierarchy.



So now we have a chest with multiple inheritance so it can be a chest that moves. Simple on paper, very hard to implement, and impossible to debug (handcrafting). So how do we fix this?

CBA to the Rescue:

We can fix it with a proper application of CBA. If we use CBA we can alleviate by adding proper concessions into the Generic Game Object class ([10], Component Based Game Architecture) to accept new components to let the chest move without requiring multiple inheritance. This lets us in essence create a very complex game without using very complex methods. The game engine itself becomes easier to manage because it was built with our loosely coupled, modular parts.

Conclusion

As discussed above, there are many architecture styles that can be used when designing software. The type of architecture that is best for a given program depends on the type of software that is being developed and the style that developers are most confident in using.

References

[1] Microsoft. (2013). *Chapter 3: Architectural Patterns and Styles* [Online]. Available: http://msdn.microsoft.com/en-us/library/ee658117.aspx

[2] D. Garlan and M. Shaw. (1994). An Introduction to Software Architecture [Online]. Available Pdf: http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf

[3] J. Booth, Building Middle Tier Objects in Visual Foxpro [Online]. Available: http://www.jamesbooth.com/n-tier.htm

[4] P. D. Sheriff, Building an N-Tier Application in .NET [Online]. Available: http://msdn.microsoft.com/en-us/library/ms973279.aspx

[5] Oracle Corporation, Overview of Oracle9iAS Discoverer [Online]. Available: http://docs.oracle.com/cd/A97335_02/busint.102/a90288/overview.htm

[6] Rainer Niekamp. Software Component Architecture [PDF]. Available: http://congress.cimne.upc.es/cfsi/frontal/doc/ppt/11.pdf

[7] RP (2012, May 12). What are the advantages and disadvantages of plug-in based architecture? [Online]. Available: http://stackoverflow.com/questions/2818415/what-are-the-advantages-and-disadvantages-of-plug-in-based-architecture

[8] Adam Smith (2007, September 12) Why Objective-C is Cool [Online]. Available: http://loadcode.blogspot.ca/2007/09/why-objective-c-is-cool.html

[9] Steffen Itterheim (2010, June 11). Prefer Composition over Inheritance [Online]. Available: http://www.learn-cocos2d.com/2010/06/prefer-composition-inheritance/

[10] Jordan Springett (2012, Feb 16). Component Based Game Architecture [Online]. Available: https://sites.google.com/site/jspringettblog/news/componentbasedgamearchitecture

[11] Rave Uno. (2011, June 18). Peer-to-Peer Vs. Client Server Networks [Online]. Available: http://www.buzzle.com/articles/peer-to-peer-vs-client-server-networks.html

[12] Lexus. (2011, March 20). Pros And Cons of Distributed Client/server Architectures [Online]. Available: http://www.bukisa.com/articles/471444_pros-and-cons-of-distributed-clientserver-architectures

[13] Microsoft. Chapter 7 - Client/Server Architecture [Online]. Available: http://technet.microsoft.com/en-us/library/cc917543.aspx

[14] Microsoft. SQL Server Overview [Online]. Available: http://www.microsoft.com/en-us/sqlserver/product-info/overview-capabilities.aspx#tab2

[15] Microsoft. Chapter 7 - Fundamentals of SQL Server Architecture [Online]. Available: http://technet.microsoft.com/en-us/library/cc917541.aspx

[16] M. I. Jawid Nazir. Client Server Computing Introduction [PDF]. Available: http://www.manipalitdubai.com/material/Lecture_Notes/STM304/module1.pdf

[17] S. H. Kaisler. (2005). Software Paradigms [PDF]. Available: www.zenisoft.cn:10010/get/pdf/1004

[18] H. Pennington *et al.,* (2005). D-Bus Tutorial [Online]. Available: http://dbus.freedesktop.org/doc/dbus-tutorial.html

[19] Apple. (2012). *Concepts in Objective-C Programming* [Online]. Available: https://developer.apple.com/library/mac/#documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html

[20] Microsoft – Message Bus [Online]. Available: http://msdn.microsoft.com/en-us/library/ms978583.aspx

[21] Tody, Doug. (1998). Message Bus and Distributed Object Technology [Online]. Available: http://adass.org/adass/proceedings/adass97/todyd2.html

[22] Microsoft – Publish / Subscribe [Online]. Available: http://msdn.microsoft.com/en-us/library/ff649664.aspx

[23] Greg Flurry and Kim J. Clark. (2001). The Enterprise Service Bus, re-examined [Online]. Available: http://www.ibm.com/developerworks/websphere/techjournal/1105_flurry/1105_flurry.html

[24] H. Pennington *et al.,* (2005). D-Bus Concept Diagram [PNG]. Available: http://dbus.freedesktop.org/doc/diagram.png