Improving Time to Market for Software Projects

The holy grail of software product development is *time to market.*

Ideally, we should be able to:

- snap software components together to create products
- design software products and communicate the design to developers
- have as few dependencies as possible
- incorporate software assets developed by third parties (and/or in-house developers)
- integrate early
- perform inspection and testing of software assets developed by third parties[1]
- change the design[2] when requirements change, at any point in the development process
- keep Design separated from Code.

Film Production as a Model for Software Development

Ideally, Software should be produced in the same manner as films are produced.[3]

- Storyboard for overall film flow
- A script

---

[1] Called “incoming inspection”.

[2] drastically

[3] Or, Software should be produced in the way that Audio recordings are produced.

- Characters, personalities, what happens if there are too many characters? (Combine multiple characters into one).
- Scenes - detailed storyboards for every component of the film.
- rule of 7 plus minus 2 - keep things simple enough to understand / remember
- Visual assets - scenery and green-screen visual shots (who decides which assets are acceptable / unacceptable?)
- Audio assets - music (who decides what music is used, who decides what kind of music is needed for each scene?)
- Scoring
- Cut / splice points
- Time sequencing
- Final editing - what to leave on the cutting room floor?

Why Not?

- Software programs have too many dependencies[4]
- Software languages don't allow easy shuffling of components
- Time / sequencing is not explicitly handled, even though most business need to sequence interactions with their customers.
- Software programs do not "snap together"[5]
- Software programs cannot be understood by anyone outside of the programming community.
- Testability - programs are hard to test, even with TDD
- Programs are usually not designed in a hierarchical manner[6]
- No language for Design (no blueprints)
- Hidden synchronizations cause problems (CALL / RETURN is a synchronization mechanism)
-

[4] Despite the existence of libraries

[5] APIs are too complicated. Arrowgrams simplifies the concept.

[6] Despite attacks on global variables, structured programming.

Arrowgrams is a Corporate Org Chart for Software

Design and Communicate

Decomposing Software Projects

Divide and Conquer

Top Down

Bottom Up

Snapping Software Components Together

Reduce Dependencies Between Components

Libraries

Frameworks

Encapsulation

The holy grail of software development is *encapsulation*.

OO (object oriented) encapsulates data and attempts to treat software as encapsulated data.

Many other aspects of a Design can be encapsulated, not just data.

Businesses already use *hierarchy* and *encapsulation* without realizing it. For example, a department head owns five divisions where each division is headed by a manager and a group of employees. The employees report directly to the managers, and the managers report directly to the department head. In this way, the department head can control all of the employees without being swamped by talking to too many people. Only the five managers report directly to the department head and they pass down directions from the department head to their employees.

This is, also, the way to design products.[7]

Divide and conquer.

[7] Including software products

Break every part of the Design down into constituent parts so as not to be overwhelmed by all of the design decisions required in the final product.

Arrowgrams encourages such a hierarchical design process. For example, you can Design a product in layers, examining each layer on its own.

Arrowgrams Diagrams show the layers of a product Design via composition of rectangles and show the routing of information via wires[8].

The diagrammatic nature of Arrowgrams immediately shows when a particular layer is too complicated[9]

Encapsulate Everything, Not Just Data

What Can Be Encapsulated

- control flow[10]
- data
- architecture
- data transfer (message sending, parameters in calls, etc.)
- global variables
- local variables
- continuations
- callbacks
- dynamic call chains
- threading
- inter-code dependencies
- macros

---

[8] Arrows between ports.

[9] Which, then, leads to a decision to break the layer down even further.

[10] if/then/else, while, call, return, etc.

- pointers
- abstraction
- polymorphism
- state
- correctness proofs
- types
- namespaces
- garbage collection, memory allocation
- concurrency
- componentization
- overloading of operators
- security
- visualization of software
- looping constructs (LOOP, recursion, etc)
- testing
- project scoping[11]
- multi-cores
- distributed cpus, distributed computers
- timing, sequencing.

Incorporate Assets Developed by Contractors

Incoming Inspection

As assets arrive, e.g. from third parties, we must be able to easily test them for usability and how they fit in with the overall design.

This means that we need to easily / quickly build up and tear down test jigs for incoming inspection of assets.

[11] Is this the same as Architecture or is it something else?

Pipelines and scripting languages were a step in this direction.

Is This The Same As TDD?

Early Integration

When we draw a design on a whiteboard, we are specifying how software components are expected to fit together. Ideally, such drawings would be made rigorous and automatically converted to software.

With Arrowgrams, you can draw diagrams that compile directly to software.

The Arrowgrams Component Diagram Editor lets you draw software components as boxes[12] with ports. Boxes are connected to other boxes by drawing lines between ports.

Component boxes are *loosely connected* and do not create dependencies between components.[13]

A Design can be created as an Arrowgrams Diagram before any code is written. Components can be integrated before any code is written.[14]

The Arrowgrams StateChart Diagram Editor[15] lets you draw the internal workings of components as simple state machines.

---

[12] reactangles

[13] This is crucial for design refactoring.

[14] This is not a feature of most other software techniques, including the use of Libraries and Git.

[15] niy

Refactoring Designs (Stealing Good Ideas)

Designs can be refactored by moving boxes and wires around on an Arrowgrams Diagram, using the Arrowgrams Diagram Editor.

Good ideas can be lifted from other Designs using simple COPY/PASTE operations of the Arrowgrams Diagram Editor.

Reusing Designs

Patterns

Reusability does not stop with Object-Oriented Software.

Arrowgrams lets us reuse Designs.

Arrowgrams separates Design from Code.

Arrowgrams Designs can be reused[16].

Arrowgrams is a Language for Software Design

In designing Arrowgrams, we put emphasis on Design. There are many languages for creating code, but precious few for expressing Software Design.

We, also, put an emphasis on compiling the designs into software.

To make a change to an Arrowgrams Design, you use the Arrowgrams Diagram editor. That way, designs are never out-of-date[17]. There is no round-

[16] With simple operations like COPY & PASTE.
[17] Like comments.

trip processing[18] in Arrowgrams, because round-tripping allows software developers to change Designs simply by changing code. There is no way to make a "last minute hack" that is not reflected in the Arrowgrams Diagrams. Arrowgrams Diagrams never become stale and out-of-synch with the code.

Ports vs. APIs

Ports and wires are like a *plugboard* used in old-fashioned telephone exchanges, electronics and some musical synthesizers.

Ports are like mini-APIs. The only question to be asked is "can I plug this wire into this Component?".

There are *input* and *output* ports.

APIs tend to come in one, asymmetrical option - *input apis*. And, APIs are much more complicated, with the user needing to know the type of every parameter.

Arrowgrams breaks *types* into two flavors. (1) Ports, and (2) the *type* of the data[19] that flows on the wires into and out of the ports. It turns out that *types* become less important in Arrowgrams

Only One Way to Communicate with Components

Arrowgrams simplifies communication between Components.

You can only SEND data from a port to another Port.

---

[18] Converting Diagrams to code, then converting code back into diagrams.
[19] event

Instead of RETURNing data to the caller, you SEND the data.

In fact, you can SEND many values[20] to another Component.

A Component can SEND data to many other Components, not just the caller.

Exceptions

There are no Exceptions in Arrowgrams.

When an error occurs, you SEND data out a port.

You can designate as many ports as you need for error handling.

Message Sending

Message Sending can only be done between components that are truly distinct.[21]

Pipelines

Distributed Processing, Multi-Core

The combination of features in Arrowgrams make it ideal for distributed applications[22].

[20] Return values
[21] e.g. Concurrent.
[22] Internet, IoT, etc.

Arrowgrams Components cannot share memory, so message sending is simple and easy.

Message Sending remains simple in Arrowgrams. Messages are routed from Component to Component by their enclosing Schematic.[23]

Diagrams instantly show where messages and wires are tangled and have become too complicated.[24] Designers can untangle wires using the hierarchical rules[25] of Arrowgrams.

The *model* for Arrowgrams Designs is shown in Fig. 1[26]

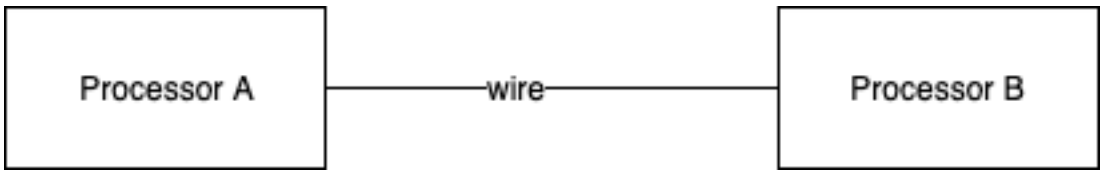

Fig. 1

All Arrowgrams Components "run to completion" and cannot be interrupted.

Loops, Unbounded Recursion

Loops are the exception, not the rule, in Arrowgrams.

Loops are frowned upon.

---

[23] Mother part.

[24] This is the same problem as "global variables" in software.

[25] Components are arranged in layers.

[26] The places where the *wire* touches the rectangles are Arrowgrams Ports. For simplicity, we do not show whether the ports are *input* or *output* in Fig. 1, although this is always required in a true Arrowgrams Diagram.

Infinite loops will cause system failure.[27]

Recursion is allowed only *inside* of code Components. Arrowgrams is a notation for designing systems using Components that are plugged together by wires. The concept of recursion does not apply to the description of Components that are plugged together by wires.

Event Loop Model

Arrowgrams Components are based on the Event Loop Model.[28]

A Component spins around watching its input ports for an event.

When an event arrives, the Component reacts to it and may generate output events[29].

RPC (Remote Procedure Calls)

Arrowgrams does not support Remote Procedure Calls (RPC) directly.

All Arrowgrams components are concurrent and asynchronous[30].

It is possible to explicitly create RPC if the design requires it. This is done by a process called *handshaking*. A Component can RPC to a downstream Component by sending it a request, then entering a state where it waits for a

---

[27] Infinite loops cause system failure in every software language.

[28] This is a variant of the Mutual Multi-Tasking model. This model was tried in Windows 3, and discredited. This model is not suitable for time-sharing systems (like Windows or MacOS), but is quite suitable for creating software products.

[29] None, one, or many output events, possibly on different output pins.

[30] Although, components composed of other components must wait until all inner components have subsided.

response from the downstream Component. See Fig. 2.

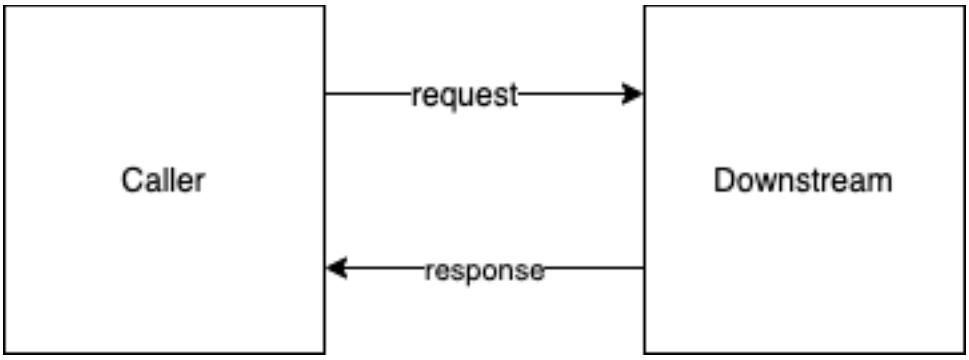

Fig. 2

The advantage of breaking RPC down this way is that the tables can be turned - the Downstream component can request data from the Caller[31].

Micro-Services

The strategy used in Fig. 2 can be used to create a micro-service based architecture.

Every Component Makes Sense

Every Component in an Arrowgrams design can be fully understood on its own.

Composite[32] Components can be made using other Components, in layers. Each layer makes sense. There is no way for a Component at a lower layer to

[31] In this case, the Downstream Component is a Client and the Upstream Component is a Server

[32] schematic

override the operation of a Component in a layer above it[33]

The Arrowgrams Model of Distributed Components

In Arrowgrams, we design Components as if they each have their own computer[34].

This philosophy simplifies Arrowgrams Designs and mimics the real world[35].

Race Conditions

Many of the traditional problems of multi-tasking are not present in Arrowgrams designs.

In the following, we discuss some of these problems and the Arrowgrams take on them.

Memory Sharing

Arrowgrams Components do not share memory[36]

---

[33] Contrast this with OO (object oriented) technology, where *inheritance* allows a component to override the operations of a component in a layer above it.

[34] Cpu + RAM, etc.

[35] IoT, multi-core, internet, etc.

[36] This rule can be broken when Components share the computer. This practice is not recommended.

Thread Safety

Thread safety is no longer an issue because Arrowgrams Components do not share memory.

Full Preemption

Full preemption is not supported by Arrowgrams.

Many of the problems[37] commonly associated with multi-tasking simply "go away" in Arrowgrams, because full preemption is not allowed.

All Arrowgrams Components "run to completion" and cannot be interrupted. Components cannot trip over their own feet.

Fairness

No longer an issue.

Every Component spins waiting for an event on any of its input pins.

If fair scheduling matters to a Design, it must be explicitly designed-in by the designer using control wires and ports.

[37] e.g. priority inversion