

(**Conocimiento** expresado en un)

***Modelo Computable***

de un ***Dominio de Problema***

de la Realidad



¿Dónde está expresado el modelo?





**Glenn Vandenburg: Real software engineering**

<https://youtu.be/RhdIBHHimeM>



# Interludio





Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.

—*Winston Royce*



**2010**  
**\* rough cut \***



# MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS

*Dr. Winston W. Royce*

## INTRODUCTION

I am going to describe my personal views about managing large software developments. I have had various assignments during the past nine years, mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis. In these assignments I have experienced different degrees of success with respect to arriving at an operational state, on-time, and within costs. I have become prejudiced by my experiences and I am going to relate some of these prejudices in this presentation.



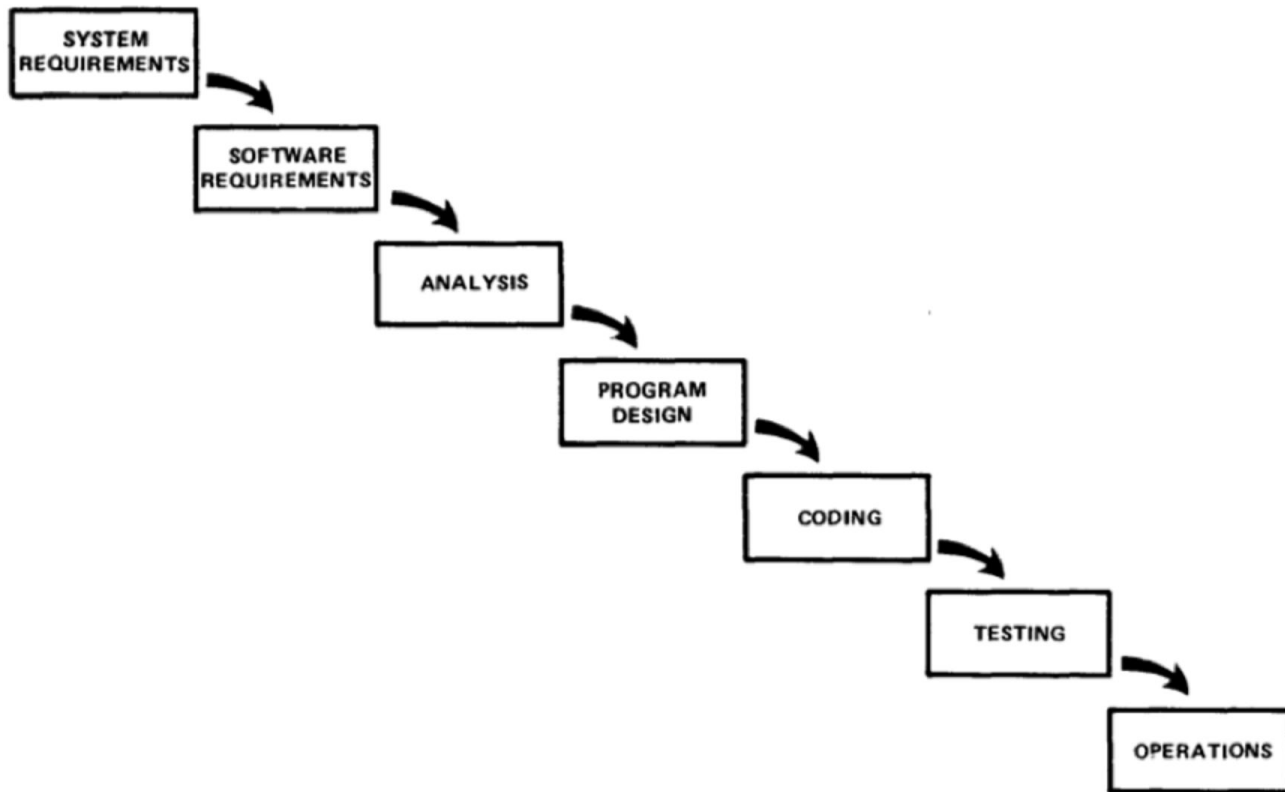


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.





I believe in this concept, but the implementation described above is risky and invites failure. The problem is illustrated in Figure 4. The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.







I believe in this concept, but the implementation described above is risky and invites failure. The problem is illustrated in Figure 4. The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.





I believe in this concept, but the implementation described above is risky and invites failure. The problem is illustrated in Figure 4. The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.



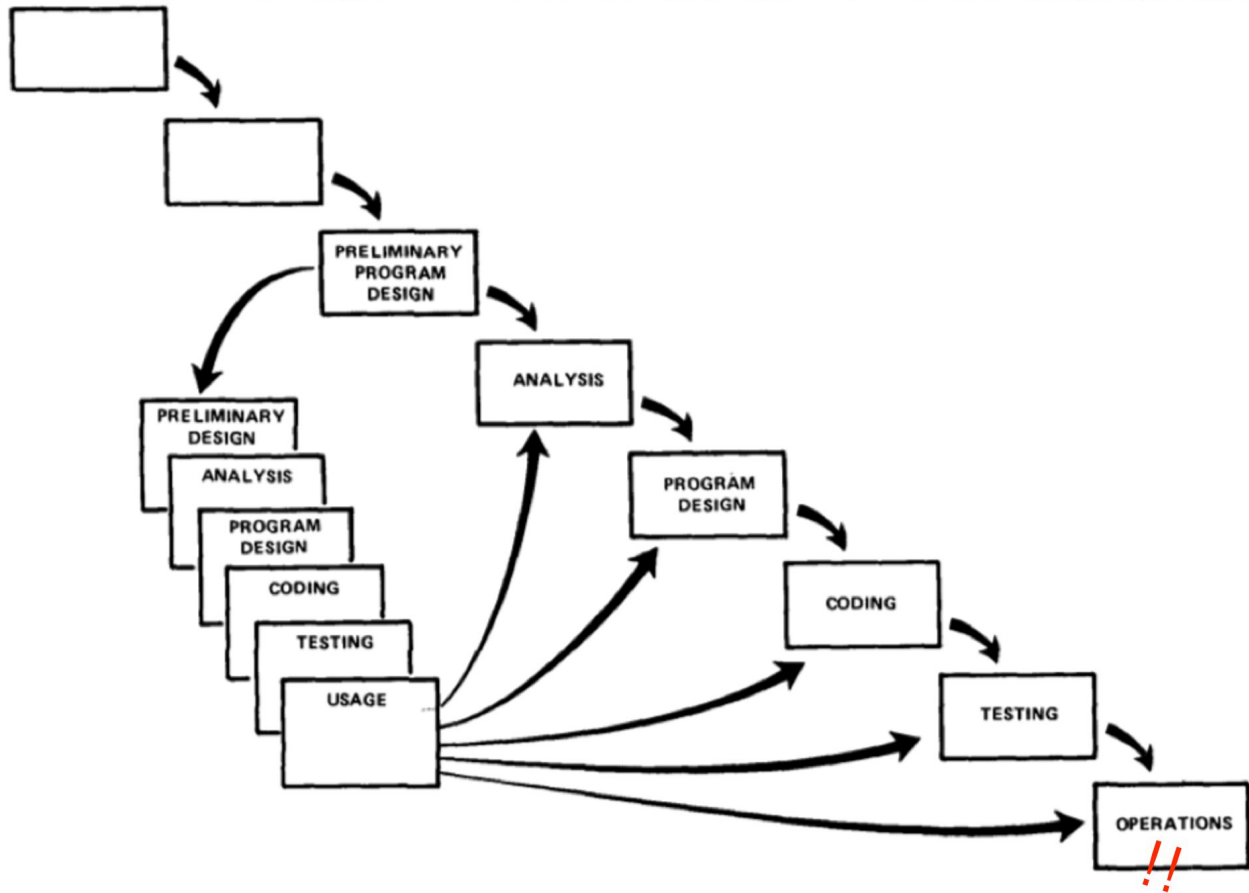


Figure 7. Step 3: Attempt to do the job twice – the first result provides an early simulation of the final product.



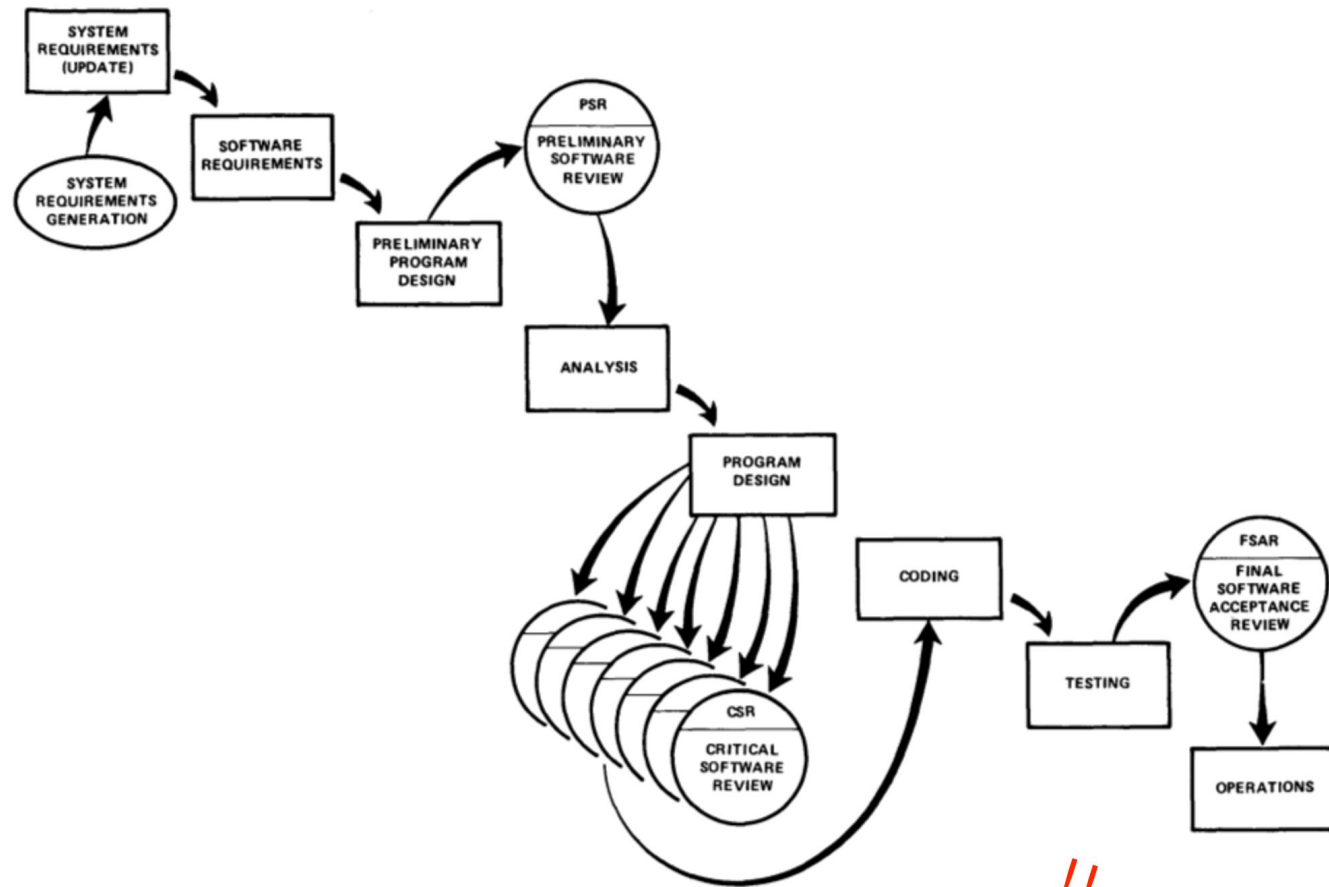


Figure 9. Step 5: Involve the customer – the involvement should be formal, in-depth, and continuing.



# Fin del Interludio



Ahora sí escuchemos a Glenn Vanderburg



## What is Software Design?

by

Jack W. Reeves  
©1992 C++ Journal

Object oriented techniques, and C++ in particular, seem to be taking the software world by storm. Numerous articles and books have appeared describing how to apply the new techniques. In general, the questions of whether O-O techniques are just hype have been replaced by questions of how to get the benefits with the least amount of pain. Object oriented techniques have been around for some time, but this exploding popularity seems a bit unusual. Why the sudden interest? All kinds of explanations have been offered. In truth, there is probably no single reason. Probably, a combination of factors has finally reached critical mass and things are taking off. Nevertheless, it seems that C++ itself is a major factor in this latest phase of the software revolution. Again, there are probably a number of reasons why, but I want to suggest an answer from a slightly different perspective: C++ has become popular because it makes it easier to design software and program at the same time.

<https://wiki.c2.com/?WhatIsSoftwareDesign>



¿Qué es Diseño?





**Descripción, gráfico o bosquejo en papel**  
relacionado con la cosa que se está diseñando,  
**destinado a su construcción**





```
class Assessment
  attr_accessor :assessmentid, :dateSubmitted, :mark, :comment
end

class Attempt
  attr_accessor :mark, :dateSubmitted, :comment
end

class Student
  attr_accessor :studentid, :dateSubmitted, :comment
end

Assessment *-- Attempt
Attempt 0..* -- 1 Student : submits

If the ass...
impleme...
assess...
assess...
mark...
alloc...

If only one attempt is allowed
per assessment, then the
association here would be 0..1
```

```
def examples
  @tc_examples ||= ExamplesCollection.new
end

def method_added(id)
  name = id.to_s
  caller_lines[name] = caller
  if test_method?(name)
    test_method_metadata[name] = @_metadata
    @_metadata_for_next = nil
  end
end

def test_info(options)
  @_metadata_for_next = options
end

def update(options)
  @_metadata_for_next.update(options)
end
```

**2010**  
\* rough cut \*



¿Qué es Diseño?



**Diseño** es la  
**definición** que usamos para **construir**



A collage of programming language logos and a code snippet. The logos include Clojure, Ruby, Scala, Python, Java, Erlang, and Microsoft .NET. The code snippet is in Ruby and shows a class definition for ExamplesCollector with methods for metadata and test cases.

```
class ExamplesCollector
  def initialize
    @examples = []
  end

  def add_example(example)
    @examples.push(example)
  end

  def test_examples
    @examples.each do |example|
      test_example(example)
    end
  end

  def test_example(example)
    test_case = TestCases.new(example)
    test_case.run
  end

  def test_case_info(options)
    test_case = TestCases.new(example_group).update(options)
    test_case.run
  end

  def test_info(options)
    test_case = TestCases.new(example_group).update(options)
    test_case.run
  end

  def method_added(id)
    name = id.to_s
    caller_lines[name] = caller
    if test_method?(name)
      test_method_metadata[name] = @_mr
      @metadata_for_next = nil
    end
  end

  def examples
    @tc_examples ||= ExamplesCollector.new
  end
end
```

**2010**  
\* rough cut \*



¿Dónde está expresado el  
Diseño de Software?



¿Qué es lo que construimos?



**¡Software ejecutable!**





¿Qué es lo que usamos para construirlo?  
¿Qué nos permite obtener Software ejecutable?



# ¡Código Fuente!



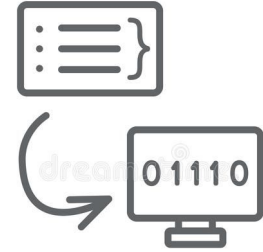
**Código Fuente = Diseño de Software**



 **Programar es Diseñar** 

**Diseñar es Programar**





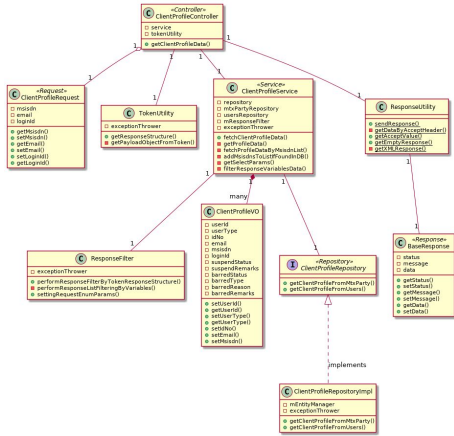
COMPILER

Nuestros “Constructores” son los Compiladores

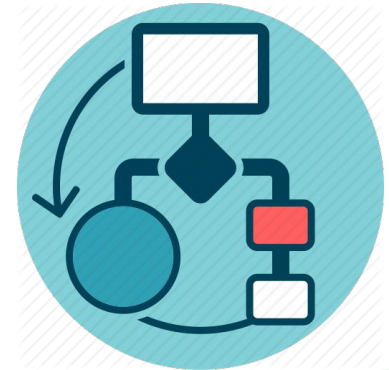


**Testear** y **Debuggear** son actividades de **Diseño**  
(validación y refinamiento de un diseño)





¿Qué papel juegan los “diagramas de diseño”?  
 (Diagrama de Clases, de secuencia, etc)



A close-up photograph of a person's hands framing a landscape. The hands are positioned at the top and bottom, with fingers pointing towards the center, creating a rectangular frame. Through this frame, a paved road stretches from the foreground into the distance, leading towards a body of water under a cloudy sky. The road is flanked by dry, yellowish-brown vegetation. The water is a pale blue-grey color, and the sky is overcast with soft, grey clouds. The overall scene is a metaphor for visualizing a design concept without being able to interact with it.

**Visualizar el Diseño  
Pero no son el Diseño**





¿Por qué nos cuesta estimar?



## ¿Por qué es difícil estimar?

- Descubrir vs. Entregar
- Estimar es difícil porque no estimamos **Cómo Construir** sino **Cómo Diseñar**



¿Que etapa cuesta más?



A collage of images and text. On the left is a photograph of a server room with rows of server racks. In the center is a snippet of Ruby code:

```
def test_info(options)
  @metadata_for_next = options
end

def method_added(id)
  name = id.to_s
  caller_lines[name] = caller
  if test_method?(name)
    test_method_metadata[name] = @_mr
    @metadata_for_next = nil
  end
end

def examples
  @tc_examples ||= ExamplesCollector.new
end
```

On the right are logos for Clojure, Ruby, Scala, Python, Java, Microsoft .NET, and Erlang.

**2010**  
\* rough cut \*



# Característica fundamental: Paso del Tiempo en el Modelo



System Browser: Object

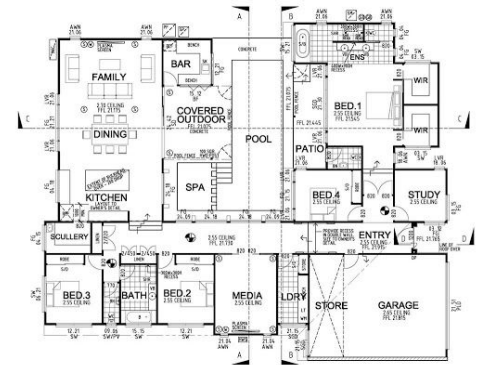
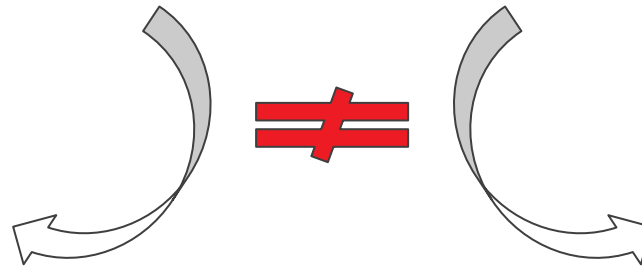
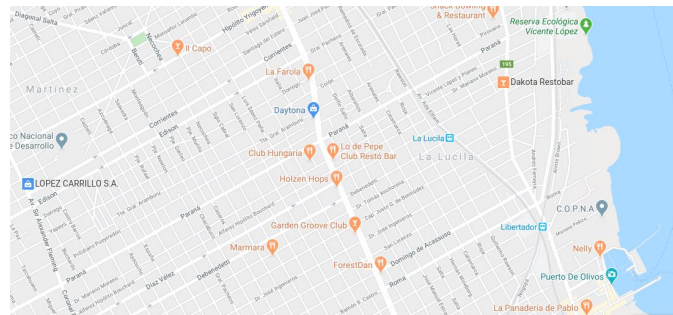
Kernel-Objects	ProtoObject	casing	=
Kernel-Classes	Object	class membership	~=
Kernel-Magnitudes	ActiveModel	comparing	~hash
Kernel-Numbers	Boolean	converting	literalEqual:
Kernel-Text	False	copying	
Kernel-Chronology	True	events-old protocol	
Kernel-Methods	instance	error handling	

no timeStamp ° comparing ° part of base system (i.e. not in a package) ° in no change set °

browse senders implementors versions inheritance hierarchy inst vars class vars show...

**anObject**  
*"Answer whether the receiver and the argument represent the same object. If is is redefined in any subclass, consider also redefining the message hash."*

**↑self == anObject**



# Es Dinámico - El tiempo transcurre

t 0

**printOn:** *aStream*

*"Append to the argument, aStream, a sequence of characters that identifies the receiver."*

.

.

.

.

.

| *title* |

*title* ← self class name.

*aStream*

nextPutAll: *title* aOrAnPrefix;

space;

nextPutAll: *title*

t n



Aquello representado por el modelo cambia

Plano de Arquitectura = Una casa

Software = Qué es una casa





# Conclusiones

- El modelo está expresado en el código fuente
  - Los diagramas no son el modelo computable
  - La documentación no son el modelo computable



# Conclusiones

- No es fácil estimar el “diseño”
  - Tenemos que estimar el diseño
  - Implica estimar “descubrir”
  - No estimamos construir



# Conclusiones

- La “construcción” no cuesta nada, lo costoso es el diseño
  - Crear el “ejecutable” a partir del diseño es lo que hacen los compiladores



# Conclusiones

- Un “modelo computable” implica “ejecución”
- Ejecución implica “paso del tiempo”
- Hay que tener mucho cuidado al comparar nuestros modelos con los de otras profesiones

