Arquitectura flexible para Métodos de Integración Multitasa

por

Julio César Pérez Sansalvador

Tesis sometida como requisito parcial para obtener el grado de

MAESTRO EN CIENCIAS EN EL ÁREA DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Octubre 2009
Tonantzintla, Puebla

Supervisada por:

Dr. Gustavo Rodríguez Gómez, INAOE
Dr. Saúl Eduardo Pomares Hernández, INAOE

Instituto Nacional de Astrofísica, Óptica y Electrónica

Arquitectura Flexible para Métodos de Integración Multitasa

por
Julio César Pérez Sansalvador

Tesis sometida como requisito parcial para obtener el grado de **Maestro en Ciencias** en el área de **Ciencias Computacionales** en el Instituto Nacional de Astrofísica, Óptica y Electrónica

Supervisada por

Dr. Gustavo Rodríguez Gómez
INAOE
Dr. Saúl Eduardo Pomares Hernández
INAOE

Tonantzintla, Pue.

Octubre 2009

Resumen

En este trabajo se presenta una arquitectura multitasa diseñada con patrones de diseño orientados a objetos. Esta arquitectura permite aproximar soluciones a problemas de valores iniciales que presenten o no diferentes escalas temporales. Con ayuda de los patrones de diseño, se genera una arquitectura flexible que, entre otras cosas, permite agregar nuevos métodos numéricos estándares para crear diferentes configuraciones para los métodos multitasa. La evaluación de la arquitectura se hace con la ayuda de una métrica que mide la capacidad tanto de adaptación al cambio como de extensión. La arquitectura multitasa propuesta puede ser utilizada como un *framework* para el desarrollo de aplicaciones que involucren aproximar soluciones a problemas con valores de inicio.

Agradecimientos

A mis padres, por el apoyo siempre presente durante este proyecto.

A mi corazón de melón Ket-ziquel, por el tiempo sacrificado para realizar este trabajo.

A mis asesores de tesis, Dr. Gustavo Rodríguez Gómez y Dr. Saúl Eduardo Pomares Hernández por su paciencia y orientación en el desarrollo de este trabajo.

A mis amigos por enseñarme que no existen cosas inalcanzables.

A mi perrita *pancha* y a mi perro *trosky* por todas las noches que se desvelaron conmigo para escribir este trabajo.

Al Conacyt por el apoyo otorgado con la beca 212419 para estudios de maestría.

Dedicatorias

A mis padres Margarita y Oscar.

A mi hermana Eva.

A mi corazón de melón Ket-ziquel.

Índice general

Capítulo 1

Introducción

Los modelos utilizados para simular fenómenos físicos generalmente incluyen sistemas de ecuaciones diferenciales ordinarias o parciales. Este tipo de ecuaciones representan las leyes físicas que gobiernan a los fenómenos modelados. La solución de los sistemas diferenciales en muy raras ocasiones se puede dar en forma cerrada, por lo que se recurre a métodos numéricos para aproximar la solución de estos sistemas.

La simulación de estos modelos involucra resolver problemas con valores iniciales cuyas soluciones pueden presentar diferentes dinámicas de tiempo y/o transitorios rápidos. Por ejemplo, en el modelo de una planta de potencia se presenta, por un lado, el conjunto de ecuaciones que representan al turbo generador, y por otro el de la caldera, esta última, al ser modelada como un proceso térmico muestra cambios lentos mientras que el turbo generador, accionado por el vapor producido por la caldera, presenta cambios rápidos al ser modelado como un proceso eléctrico. A este tipo de sistemas se les conoce como sistemas con diferentes escalas de tiempo.

Al utilizar métodos numéricos convencionales para resolver este tipo de sistemas el tiempo de cómputo resulta excesivo ya que se maneja la misma discretización para todas las partes del sistema. Para remediar esta situación es natural dividir el sistema en dos subsistemas, uno que contenga las componentes que varían relativamente rápido, y el otro con las componentes relativamente lentas. Los dos subsistemas pueden ser integrados de manera separada pero coordinada. Las técnicas numéricas

que explotan las diferentes escalas temporales de un sistema dinámico para reducir el costo computacional, se les llama métodos de integración multitasa, *subcycling* o partidos.

El principio de los métodos de integración multitasa es resolver cada subsistema con un método de integración y tamaño de paso acorde con su dinámica. La coordinación de la información entre el subsistema rápido y lento se lleva a cabo mediante interpolación o extrapolación dependiendo del algoritmo multitasa utilizado. El tiempo de cómputo es reducido ya que el método multitasa explota de manera natural las diferentes dinámicas de un proceso. Aunque los métodos multitasa son conceptualmente simples, los problemas aún no resueltos asociados a su teoría e implementación han inhibido su uso práctico [Ske84].

El problema al que nos enfocaremos en este trabajo es el diseño de una arquitectura de software para métodos de integración multitasa que de facilidades para el uso de diferentes esquemas de integración numérica y permita la coordinación de la información entre los subsistemas rápido y lento en forma eficaz.

Durante años, el desarrollo de software científico se ha enfocado en la creación de algoritmos numéricos rápidos y eficientes. Dejan a un lado la flexibilidad, reutilización, mantenimiento y escalabilidad que son las necesidades primordiales en el ámbito de desarrollo de software para negocios y/o administrativo, también, importantes en el ámbito de la computación científica. Lo anterior es originado debido a que la mayoría del software científico es desarrollado en lenguajes procedimentales como C o Fortran [Bun06].

Las técnicas de programación actuales ofrecen facilidades para hacer que la implementación sea clara, simple y eficiente. Por ejemplo, de acuerdo a Langtangen *et al.* [LM01], la programación orientada a objetos (**POO**) ofrece características que mejoran significativamente el diseño, implementación y mantenimiento de códigos grandes para simulación. Autores como Bruaset y Langtangen mencionan que el enfoque de la POO simplifica la tarea de dar flexibilidad, principalmente por las herramientas que ofrece para la fácil combinación de módulos existentes dentro de

nuevos módulos, [BL97].

La POO simplifica el desarrollo de código, incrementa la robustez y legibilidad. Además, los esfuerzos de mantenimiento son minimizados en forma significativa [AMB97]. En la práctica hay un incremento notable en la productividad humana [AMB97]. Estas son algunas de las razones del porqué en este trabajo proponemos su uso para realizar el diseño de la arquitectura de software multitasa.

Aunque la programación orientada a objetos produce mejoras en la implementación de algoritmos numéricos complejos, puede, también, reducir su desempeño [KM99]. Si se abusan de las características de la POO se puede caer en diseños complicados, difíciles de entender. El uso de buenas técnicas de programación como encapsulamiento, herencia y reutlización de software no son suficientes. Es necesario contar con una metodología que oriente el diseño de la arquitectura deseada.

Los patrones de diseño de software, de acuerdo a [Bli02], son metodologías bien entendidas para el diseño de arquitecturas de software orientadas a objetos, han sido utilizados principalmente en el desarrollo de software para negocios. Algunas de las ventajas, identificadas por [GHJV95], de los patrones de diseño de software son las siguientes:

- Capturan la experiencia en el diseño de arquitecturas de software.
- Facilitan la reutilización de diseños de software y arquitecturas.
- Ayudan en la selección de alternativas para hacer un sistema reutilizable, evitando las que comprometan su capacidad de reutilización.
- Logran que el desarrollador de software realice un diseño correcto de manera rápida.

Los patrones de software, en general, han sido empleados en el diseño de software administrativo, de hecho el catálogo de patrones elaborado por Gamma *et al.* [GHJV95] fue obtenido a partir del análisis de este tipo de software. La tarea principal de este trabajo es aplicar estos patrones para el desarrollo de software científico. Por lo tanto, una de las tareas principales es la de identificar un subconjunto de patrones del catálogo de Gamma *et al.* [GHJV95] que sean aplicables en el contexto del desarrollo de software científico.

Dentro de los lenguajes orientados a objetos (**OO**) se encuentra C++ el cual de acuerdo a [KM99] es apropiado para la programación de algoritmos numéricos. Esto es debido a que el lenguaje C++ permite el manejo de los detalles numéricos tanto a bajo como a alto nivel de abstracción. Se evita así, atar las estructuras de datos del programa al diseño de éste.

En este trabajo se propone el uso de técnicas de análisis, diseño y patrones orientados a objetos para diseñar y desarrollar una arquitectura flexible, la cual permita asignar diferentes métodos y pasos de integración numérica a los diferentes subsistemas que conforman el sistema dinámico. La arquitectura dará las facilidades para hacer la sincronización de la información entre las diferentes dinámicas del sistema.

Es importante conocer si la arquitectura a diseñar cumplirá con las características deseadas. Para evaluar su calidad se utilizará la métrica de software propuesta por Martin [Mar03], la cual mide la abstracción y dependencia entre los componentes de la arquitectura.

1.1. Hipótesis

La hipótesis principal de esta investigación es que el empleo de las técnicas de análisis y diseño de los patrones orientados a objetos nos permitirá diseñar y desarrollar una arquitectura con las características mencionadas anteriormente.

1.2. Objetivo

Diseñar y desarrollar una arquitectura multitasa **Pa**trón **O**rientada a **O**bjetos (**PaOO**) para aproximar la solución numérica de problemas de valores iniciales divididos en subsistemas.

1.2.1. Objetivos secundarios

- Crear un esquema general de la arquitectura donde se presenten las entidades más significativas en el diseño de la arquitectura.

- Diseñar una arquitectura de software con ayuda de los patrones de diseño para los métodos numéricos estándares.
- Diseñar una arquitectura de software con ayuda de los patrones de diseño que permita combinar diferentes métodos numéricos estándares bajo un esquema multitasa.

1.3. Metodología

En esta sección se presenta la metodología a utilizar para alcanzar el objetivo de este trabajo.

1.3.1. Definición de requerimientos

Para todo diseño de software es necesario contar con un conjunto de necesidades que harán de guía para la construcción del diseño de software. Los requerimientos serán obtenidos de los trabajos involucrados en el desarrollo de software para resolver problemas con valores de inicio.

En esta etapa se establecen los servicios que la arquitectura debe proveer como una lista de requerimientos especificados en lenguaje natural.

Requerimientos funcionales y no funcionales

La lista de requerimientos debe dividirse en funcionalidades propias de la arquitectura y restricciones para la implementación. La lista obtenida se enfocará en la flexibilidad para utilizar métodos multitasa más que en las restricciones de la etapa de implementación. Ésto se debe a que la arquitectura creada no debe presentar dependencias a plataformas de desarrollo de software específicos.

1.3.2. Diseño de la arquitectura

Para esta etapa se seguirá la metodología propuesta por Sommerville en [Som95] la cual involucra los siguientes pasos.

Estructuración del diseño

A partir de los requerimientos se crea una visión más clara de las características de la arquitectura. Con esta visión se crearán los primeros diagramas donde se identificarán las partes que conforman a la arquitectura.

Sommerville [Som95] menciona que como parte de los requerimientos del sistema y la actividad de diseño, el diseño del sistema a desarrollar debe ser modelado como un conjunto de componentes y relaciones entre los componentes. Este modelo se presentará gráficamente como el modelo general de la arquitectura del sistema.

Estructura de control del diseño

En esta etapa se define la estructura de control para el manejo de los datos generados por la arquitectura. Esta estructura se obtiene a partir de la información que cada subsistema necesita junto con la evaluación de las responsabilidades de éstos.

Descomposición modular

Una vez obtenida la estructura general de la arquitectura, se realiza otro nivel de descomposición. En esta etapa se convierten los subsistemas ya identificados en clases utilizando técnicas de diseño orientadas a objetos y las recomendaciones dadas por Gamma *et al.* en [GHJV95] para la generación de estas clases.

1.3.3. Diseño orientado a objetos

Ya que la complejidad en el diseño de la arquitectura no recae en el tamaño del sistema de software, sino en lo complejo de los algoritmos numéricos, es necesario entender primero, la parte matemática para tomar en cuenta las limitaciones que deben considerarse, y después realizar la abstracción indicada por la etapa de diseño orientado a objetos.

Basado en el conocimiento adquirido en la parte matemática es posible tomar

ventaja al momento de crear los objetos que conformarán a los subsistemas de la arquitectura.

Durante el proceso de diseño de la arquitectura se deben favorecer las siguientes características para lograr un diseño flexible:

- Alta cohesión
- Bajo acoplamiento
- Alto entendimiento
- Fácil adaptación a otros sistemas

Selección de patrones de diseño

Durante la etapa de diseño de clases se tomará en cuenta la metodología presentada por Gamma *et al.* en [GHJV95] para la selección de los patrones de diseño de software. Ésta se presenta a continuación:

1. Considerar la manera en que los patrones de diseño ayudan a resolver problemas de diseño. Sección 1.6 del libro [GHJV95].
2. Buscar por el patrón más relevante al leer la sección *Intent* del catálogo de patrones de diseño presentado en [GHJV95] para relacionar las partes del problema con las del patrón.
3. Estudiar las relaciones entre los diferentes patrones para identificar a un posible grupo de patrones a aplicar.
4. Revisar los patrones clasificados con el mismo *propósito*.
5. Examinar las posibles causas de rediseño. Buscar aquellos que eviten el rediseño.
6. Considerar las partes que puedan cambiar en el diseño. Al encapsular la parte que varia es posible variar aspectos del diseño sin necesidad de rediseñar.

La metodología presentada será complementada con las recomendaciones para el uso de patrones de diseño que Shalloway expone en [ST04]. Gamma *et al.* señalan en [GHJV95] algunas advertencias sobre el uso excesivo de patrones de diseño, entre estas se encuentra que *no* hay que aplicar patrones a menos que la flexibilidad que ofrecen sea en verdad necesaria, de lo contrario se crearán diseños complicados por el alto nivel de variabilidad introducida por los patrones de diseño.

1.3.4. Medida de la calidad del diseño

Para evaluar la calidad de la arquitectura respecto a la flexibilidad, dependencia de clases y nivel de abstracción, se utilizará la métrica propuesta por Martin en [Mar03] que mide la cohesión, acoplamiento y capacidad de extensión del diseño, entre otras.

1.3.5. Validación y Verificación

En la etapa de validación se comprobará que la arquitectura cumpla con los requerimientos especificados al aplicar la métrica mencionada. Para la etapa de verificación se realizarán pruebas que han sido definidas por Sommerville en [Som95], entre las que destaca el uso de herramientas para verificación de la arquitectura.

1.4. Organización del documento

En el capítulo 2, *Estado del arte*; se presenta un resumen de los trabajos relacionados con el desarrollado de software para aproximar soluciones a ecuaciones diferenciales ordinarias (**EDO**). En el capítulo 3, *Fundamentos en métodos numéricos para aproximar soluciones a EDO's*; se explican de manera breve los conceptos relacionados con el tema de Métodos Numéricos. En el capítulo 4, *Fundamentos en desarrollo de software con Patrones de Diseño*; se presentan los conceptos fundamentales sobre Patrones de Diseño de Software. En el capítulo 5, *Arquitectura propuesta*; se presenta de manera detallada el desarrollo del diseño de la arquitectura para Métodos de Integración Multitasa a través de Patrones de Diseño de Software. En el capítulo 6, *Validación y Verificación*; se presenta la evaluación de la arquitectura propuesta con la métrica mencionada en el mismo capítulo. En el capítulo de *Conclusiones*; se exponen las conclusiones y el posible trabajo futuro a desarrollar.

Capítulo 2

Estado del arte

En este capítulo se describen algunos de los principales trabajos que han contribuido al desarrollo de software científico. Se presentan las ventajas y desventajas de los lenguajes procedimentales y orientados a objetos bajo este contexto. Se explica la influencia del modelo OO en el desarrollo de software científico, y los enfoques más significativos para resolver EDOs numéricamente. Además, se exponen los desarrollos de software que involucran el uso de Métodos de Integración Multitasa, y se pone un énfasis especial en las propuestas que emplean el paradigma OO.

2.1. Enfoques en el desarrollo de software científico

Durante años, el desarrollo de software científico se ha enfocado en la creación de algoritmos numéricos rápidos y eficientes, dejando a un lado la flexibilidad, reutilización, mantenimiento y escalabilidad. Tradicionalmente, el paradigma bajo el cual se ha desarrollado este tipo de software ha sido el de programación procedimental (*procedural programing*), siendo sus representantes más significativos Fortran y C. Como ejemplo del software desarrollado bajo este modelo podemos citar el repositorio de software para cómputo científico *Netlib* el cual contiene un gran número de programas y librerías, entre ellas las famosas LaPack (**L***inear* ***A****lgebra* ***Pack****age*)

[ABB+99] y BLAS (**B***asic* ***L****inear* ***A****lgebra* ***S****ubprograms*) [LHKK79], ambas para operaciones donde se requiera manipulación algebraica.

Debido a la poca flexibilidad de los algoritmos numéricos implementados en Fortran o C los ingenieros y/o científicos han optado por desarrollar su propio software, lo que resulta en sistemas de software fuertemente acoplados al problema o modelo que resuelven. Estos sistemas de software son poco o nada flexibles, no reutilizables y de mantenimiento difícil.

Es deseable contar con paquetes de software numérico flexibles, que permitan realizar pruebas de modelos matemáticos de manera sencilla. Estos paquetes deberían ser fácilmente integrados a otros sistemas en desarrollo, por lo que deben ser escalables y de fácil mantenimiento. El usuario final no debería preocuparse por conocer a detalle la implementación, sino que por medio de estándares o patrones logre integrar, extender y/o reutilizar los paquetes existentes.

El enfoque de POO cubre las necesidades anteriores. Al ocultar la implementación de los algoritmos numéricos por medio de la abstracción, se logra que el usuario no se preocupe por la estructura interna de los objetos que utiliza, sino que explote el comportamiento de estos [Bir93]. Además, la POO facilita el mantenimiento y las extensiones al código desarrollado.

2.1.1. Ventajas y desventajas

A continuación presentamos una lista de las principales ventajas y desventajas de los dos modelos de programación mencionados:

- **Programación Procedimental**
  - Algoritmos numéricos rápidos y eficientes.
  - Programación a bajo nivel que permite el uso de recursos computacionales de manera eficiente.
  - Para el caso de Fortran, su portabilidad se basa en la popularidad del lenguaje dentro del contexto científico o ingenieril, [Bir93].
  - No hay flexibilidad en el código, son difíciles de mantener y/o extender.

- **Programación Orientada a Objetos**

  - Alto nivel de abstracción, evita perderse en el detalle.
  - El mantenimiento y modificación del código existente se facilita al crear nuevos objetos a partir de los existentes por medio de la *herencia*.
  - Las características de *encapsulamiento*, *herencia* y *polimorfismo* son elementos claves al momento de relacionar la implementación con la abstracción matemática, [AMB97].
  - La POO fomenta la implementación computacional de abstracciones matemáticas, [AMB97].
  - En el caso de C++, es posible usar código C de bajo nivel para implementar las funciones que tengan una pesada carga numérica.
  - C++ es capaz de lograr un desempeño incluso mejor que Fortran al utilizar compiladores especializados junto con técnicas de programación genérica, [VJ97].
  - La falta de eficiencia ha sido el argumento más importante contra el uso del modelo de POO para el desarrollo de software numérico, [AMB97].
  - La disponibilidad de compiladores para C++ en el ámbito industrial no asegura la eficiencia de éste, sobretodo en aplicaciones numéricas. Tampoco, la disponibilidad de librerías numéricas estándares escritas en C++, [KM99].

2.2. Patrones en el desarrollo de software

Las ventajas ofrecidas por la POO no serían por sí mismas suficientes para diseñar la arquitectura Multitasa con las características mencionadas. Se necesita de la experiencia del desarrollador en el diseño de software para cumplir con las expectativas.

El inconveniente anterior tiene por fortuna una solución, encontrada en los años 90's: los *patrones de diseño de software*. Ellos, almacenan la experiencia del diseño de software orientado a objetos de los expertos [GHJV95]. En los trabajos de Cuéllar, Masuda y Ouyang se utilizan para diseñar o rediseñar sistemas de software. El resultado son sistemas flexibles, escalables, fáciles de extender y de mantener gracias al empleo de los patrones [CVRG04], [MSU00], [Ouy02].

2.3. Desarrollo de software científico bajo el enfoque OO

El desarrollo de software se ha vuelto cada vez más complicado debido a que las aplicaciones a desarrollar son cada vez más grandes y complejas. En la última década ha habido un incremento en el interés de aplicar el paradigma de POO en el desarrollo de software numérico, [AMB97]. La necesidad de implementar algoritmos numéricos complejos ha despertado un interés potencial en los beneficios ofrecidos por la POO, [KM99].

Ya que la mejora y actualización de los algoritmos numéricos implica desarrollos en el área de cómputo científico, es necesario contar con técnicas que permitan la extensión y reutilización del código de una manera más simple que la ofrecida por el paradigma de programación procedimental. Entre los primeros trabajos que mostraron las ventajas de aplicar el paradigma OO en el desarrollo de software científico se encuentran los siguientes:

El trabajo desarrollado por Birchenhall, [Bir93], *MatClass*, es un proyecto para experimentar en el uso de métodos de POO en el campo de métodos numéricos mediante el lenguaje C++. *MatClass* contiene a una familia de clases que permite trabajar con matrices que combinado con C++ ofrece al usuario un ambiente para desarrollar aplicaciones que impliquen operaciones con matrices.

MatClass es un software abierto, extendible y portable. En el contexto del autor se refiere a que es abierto ya que el código fuente se encuentra al alcance del público, es extensible porque el usuario puede definir sus propias clases de objetos heredando propiedades de las clases ya definidas, y es portable al no presentar dependencias con el sistema operativo.

Mientras *MatLab* ha sido visto como un sistema ideal para implementar algoritmos prototipo antes de ser trasladados a Fortran, *MatClass* pretende eliminar la frontera entre la etapa de probar algoritmos prototipo y desarrollo en C++.

En [KM99] se presenta a *DAE-TK*, una implementación en C++ de la función

daspk desarrollada en Fortran 77, se utiliza para resolver ecuaciones diferenciales algebráicas. En este trabajo se menciona que debido a la necesidad de crear códigos menos complejos algunos científicos e ingenieros han optado por utilizar el enfoque de POO en lugar del paradigma procedimental.

Las metas de *DAE-TK* son destacar las ventajas de implementar software numérico con técnicas de POO, mostrar las consideraciones de diseño útiles en el desarrollo de aplicaciones numéricas complejas y evaluar las consecuencias en el desempeño del estilo de programación de C++. Una de las ventajas del polimorfismo es que evita el uso de sentencias de control para seleccionar la parte de código que debe ser ejecutada.

Algunas de las recomendaciones encontradas con el desarrollo de *DAE-TK* son:

- Evitar copiar y construir objetos de gran tamaño como vectores o matrices de manera innecesaria.
- Usar funciones *inline* para mantener la eficiencia.
- En los argumentos de entrada de las funciones utilizar llamadas por referencia constantes (*const & TYPE*) cuando se pasen objetos grandes y llamadas por valor cuando se trate de objetos pequeños.
- Utilizar funciones de la librería *math.h*.

En el mismo trabajo se concluye que se obtiene una mejora considerable en el desempeño del programa al utilizar las funciones para operaciones matemáticas intrínsecas del lenguaje. El uso de polimorfismo tiene un efecto *insignificante* en el tiempo de ejecución total. En general, las consecuencias del bajo desempeño se deben a la acumulación del mal uso de C++ en los niveles más bajos.

Una lista de paquetes de software numérico desarrollados bajo el paradigma de POO puede ser encontrada en el sitio *http://www.oonumerics.org/*.

Debido a las ventajas ofrecidas por el enfoque OO y la experiencia que los patrones de diseño ofrecen, proponemos el uso conjunto de éstos para cumplir con las características de la arquitectura Multitasa a diseñar.

2.4. Software científico para resolver EDOs

Dos han sido los enfoques importantes para resolver sistemas de EDOs, el primero, consiste en ayudar al usuario a seleccionar un método numérico junto con sus parámetros, el segundo se orienta hacia la creación de paquetes de software que puedan ser utilizados dentro de aplicaciones de propósito más general.

2.4.1. Selección automática del método numérico para resolver EDOs

La selección automática del método numérico apropiado se realiza con base en el estudio de las características del sistema de ecuaciones.

Este enfoque es motivado por los problemas que resultan al seleccionar un método numérico inadecuado. De acuerdo a Bunus en [Bun06], la eficiencia de la simulación numérica está fuertemente influenciada por la selección correcta del método numérico. Las consecuencias de aplicar un método inapropiado dan origen a resultados incorrectos o errores en la aproximación demasiado grandes, además, del alto costo computacional [KEM93].

En el trabajo de Pectu [Pet05] se menciona que la gran cantidad de software numérico disponible para resolver EDOs se ha convertido en un problema. Los usuarios que no están familiarizados con los algoritmos numéricos tienen que enfrentarse con la dificultad de seleccionar un paquete de software numérico por lo que sería de gran ayuda conocer los métodos adecuados para resolver su problema. En el mismo trabajo se presentan las características que debe cumplir un sistema experto para resolver EDOs.

El sistema desarrollado por Kamel en [KEM93] se llama *ODEXPERT*, el cual, a partir de un modelo matemático especificado como un sistema de EDOs selecciona un método numérico apropiado para resolverlo. El sistema se implementó en *OPS*, un lenguaje de desarrollo basado en conocimiento, algunos de los componentes que conforman a *ODEXPERT* han sido escritos en C y Fortran. El sistema interacciona

con *MAPLE* para realizar la manipulación simbólica del modelo de entrada.

ODEXPERT se conforma básicamente de cuatro módulos: el primero, es la interfaz del usuario donde se especifíca en lenguaje Fortran el sistema de EDOs, el segundo, es donde se realizan las pruebas al sistema de EDOs para identificar sus propiedades, el tercero, es la base de conocimientos que, a partir de las propiedades obtenidas en el módulo dos, utiliza un sistema basado en reglas para recomendar el método numérico adecuado, y el cuarto módulo se encarga de presentar al usuario las recomendaciones finales.

Un trabajo similar es desarrollado por Bunus en [Bun06], donde se presenta el sistema *ModSimPack*; plataforma de simulación-decisión desarrollada en *Modelica 2002* [Mod05], selecciona automáticamente el método numérico a utilizar para resolver un modelo matemático dado. *ModSimPack* se divide en dos partes principales: la primera, realiza la manipulación simbólica del modelo para determinar sus características, y la segunda se encarga de evaluar esas características y seleccionar un método numérico apropiado para resolver el modelo. Una de las desventajas de este trabajo es que la base de conocimientos de métodos numéricos contiene un número limitado de éstos.

2.4.2. Software científico para resolver EDOs bajo el enfoque OO

En el trabajo desarrollado por Conway [Con95], se presenta una arquitectura para métodos Runge-Kutta adaptativos. Entre las ventajas obtenidas al utilizar el enfoque OO se listan las siguientes:

- Permite agregar nuevos métodos Runge-Kutta con sólo indicar los coeficientes del nuevo método.
- El método numérico no contiene detalles del sistema de EDOs, lo que facilita el cambio del modelo a resolver.
- La estructura jerárquica de la arquitectura, permite la reutilización de los módulos abstractos para definir nuevas familias de métodos.

La arquitectura desarrollada por el trabajo anterior resulta de uso complicado para el usuario no familiarizado con POO.

Como el trabajo mencionado se encuentra acoplado a los métodos Runge-Kutta adaptativos, es deseable extenderlo para trabajar con otro tipo de métodos. Este trabajo ha servido de inspiración para la realización de esta tesis.

Entre los trabajos que han desarrollado software numérico bajo el enfoque OO para resolver EDOs se encuentra el de Milde *et al.* [Mil98], *ODE++* es una librería escrita en C++ para la representación y solución de EDOs. En este trabajo se menciona que los usuarios que necesitan resolver un problema con EDOs deben especificar hasta veinte variables para utilizar una rutina que resuelva su problema. La reutilización de código se dificulta debido a la falta de modularidad.

La arquitectura propuesta para *ODE++* separa mediante una jerarquía de clases la representación del sistema de EDOs, la representación de los parámetros del problema y la solución del problema. Los objetivos de diseño para *ODE++* son:

- Una interfaz consistente que sea independiente de la representación del problema o el algoritmo numérico utilizado para aproximar su solución.
- Una interfaz que sea fácil de manejar por usuarios no experimentados.
- Flexibilidad para alterar los parámetros de un método, ajustes detallados, si fuera requerido.
- Capacidad para extender la librería agregando nuevas clases con diferentes estilos para representar el sistema de EDOs, además, de nuevos métodos para resolver estos problemas.

En el mismo trabajo, se emplea el enfoque OO de los años 80s, uso exhaustivo de herencia y polimorfismo, ya que se reducen los esfuerzos al momento de probar y depurar el código. Sin embargo, puede originar explosión de clases lo que trae como consecuencia un mantenimiento difícil del código.

Una de las características importantes de *ODE++* es que la consistencia en las interfaces de los objetos es hace posible utilizarla como parte de un proyecto de software más grande. Lamentablemente, este software ya no se encuentra disponible actualmente.

2.5. Software para Métodos de Integración Multitasa

En el trabajo de Gear, [Gea81], se presentan los problemas relacionados con la solución numérica de EDOs que en el año 1981 presentaban retos a los investigadores del área. El camino en la búsqueda de soluciones comienza con la identificación del problema hasta alcanzar el desarrollo de software robusto. El proceso se puede ver como una escalera donde cada escalón representa uno de los siguientes elementos:

1. Identificación del problema.
2. Primeros métodos y códigos preliminares.
3. Análisis matemático del problema.
4. Primeros códigos para producción.
5. Análisis del software para una clase de problemas.
6. Desarrollo de software robusto y de propósito general.

El entendimiento de cada uno de estos elementos va mejorando con el paso del tiempo. La figura 2.1, presenta la idea en forma de escalera; ver [Gea81].

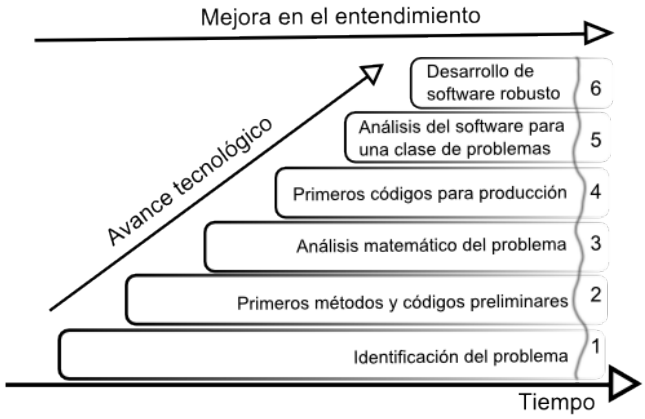


Figura 2.1: Avance tecnológico y mejora en el entendimiento del problema, tomada y traducida de [Gea81].

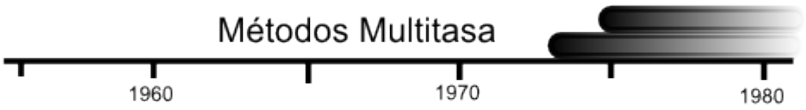


Figura 2.2: Situación de los Métodos Multitasa para 1981, tomada y traducida de [Gea81].

El caso de los métodos multitasa para ese año se presenta en la figura 2.2 donde se observa que ya se había identificado el problema y se contaban con los primeros métodos y códigos preliminares.

Como se puede observar, el estudio y desarrollo de los Métodos Multitasa en esa época apenas comenzaba. Los avances encontrados al día de hoy se presentan en trabajos encaminados al análisis matemático de los métodos multitasa, [RGM04], [And93], [Kvæ00], [CWG84], [Wel82]. En los trabajos de Kværnø[KR99] y Engstler *et al.* [EL97] se presentan códigos, en lenguajes procedimentales, que implementan este tipo de métodos. Éstos se ubican en el punto 4 (*primeros códigos para producción*) de la figura 2.1.

El trabajo realizado en esta tesis, al utilizar un enfoque diferente en el diseño de software que integre a los Métodos Multitasa y al realizar el análisis de las características que éste debe cumplir, logra cubrir los puntos 4 (*preparación de los primeros códigos para producción*) y 5 (*análisis del software para una clase de problemas*) de la figura 2.1. Al no contar con una teoría matemática completa para los Métodos Multitasa, sólo es posible cubrir parcialmente el punto 6 (*desarrollo de software robusto y de propósito general*) de la figura 2.1.

2.5.1. Métodos de Integración Multitasa bajo el enfoque tradicional

En [EL97] se propone un método de integración multitasa basado en la extrapolación de Richardson para aproximar la solución a sistema de EDOs. Las ecuaciones obtenidas al aplicar el método de extrapolación serán las utilizadas para calcular

las aproximaciones a la solución del sistema de EDOs. En contraste con el enfoque tradicional de los métodos multitasa, la componente lenta y rápida son integradas simultáneamente utilizando un macro paso.

La implementación del algoritmo propuesto se hace en lenguaje Fortran. El código se ha escrito de manera que pueda ser aplicado a problemas de un sólo tipo, es decir, las entradas al programa deben cumplir ciertas restricciones. La llamada al método integrador sigue los estándares de los métodos integradores para EDOs implementados en Fortran, los cuales presentan al menos once parámetros.

Debido a que el método reemplaza las funciones del modelo por otras obtenidas mediante la extrapolación, se muestran buenos resultados al momento de disminuir la tolerancia en el error, sin embargo, el costo computacional se ve afectado por los cálculos extras necesarios para el método de extrapolación.

En [KR99] se presentan dos algoritmos, MRKI y MRKII, ambos implementan métodos Runge-Kutta adaptativos bajo un esquema Multitasa. Estos algoritmos son probados con un ejemplo con diferentes escalas de tiempo, a saber, el modelo de un circuito donde la parte análoga representa la parte lenta y la digital la parte rápida.

Los algoritmos son implementados bajo el esquema de programación procedimental, específicamente en el lenguaje Fortran. Agregando a esto la complejidad de los algoritmos propuestos se espera que sea un código de mantenimiento difícil y poco o nada flexible.

2.5.2. Métodos de Integración Multitasa bajo el enfoque OO

El trabajo desarrollado por Matthey [MCH+04], *ProtoMol*, es una plataforma para la simulación de dinámica de moléculas. Permite probar nuevos algoritmos desarrollados para esta tarea y pretende ser utilizado para el entrenamiento de estudiantes. Los programas de software existentes, por su complejidad, no son adecuados para lograr las metas de *ProtoMol*. La flexibilidad de *ProtoMol* es alcanzada gracias al uso de herencia y patrones de diseño.

De acuerdo a los desarrolladores de *ProtoMol*, el uso de técnicas de POO y progra-

mación genérica (*templates*) permite alcanzar la flexibilidad y desempeño deseado. Por un lado, mediante el uso de herencia y patrones de diseño se logra la reutilización de código, no sólo a nivel de algoritmo sino también a nivel de arquitectura. Por otro lado, el uso de *templates* permite generar código eficiente en las secciones críticas para el desempeño.

Los patrones de diseño utilizados en *ProtoMol* son: *Policy, Strategy, Abstract Factory* y *Prototype*. Al combinar las ventajas que estos ofrecen se alcanza la flexibilidad en la arquitectura del sistema, además de su fácil mantenimiento y extensión. En *ProtoMol* se afirma y se muestra evidencia de que la POO y el uso de programación genérica (templates) no necesariamente sacrifican un buen desempeño.

Entre las características interesantes de *ProtoMol* se encuentra el uso de *Multiple Time Step Methods,* métodos numéricos formados por combinaciones de métodos especializados para resolver problemas de dinámica de moléculas. Al igual que los métodos multitasa, dividen el sistema de acuerdo a sus diferentes escalas de tiempo. Los métodos utilizados en *ProtoMol* se utilizan con frecuencia para calcular trayectorias de partículas en la simulación de dinámica de moléculas.

En la exposición de trabajos de esta sección, sólo el de Matthey *et al.* [MCH$^+$04] utiliza patrones de diseño para desarrollar su software.

Respecto a la aplicación de las técnicas de POO para la implementación de Métodos Multitasa a la fecha no se encuentran trabajos por lo que el campo en el que se ubica esta tesis es prácticamente nuevo.

2.6. Resumen del estado del arte

En la tabla 2.1 se presentan los trabajos más importantes en cuanto al desarrollo de software científico para resolver EDO's, además, en último lugar, se presentan las características de la arquitectura propuesta en este trabajo.

Trabajo	Paradigma de programación	Método estándar o Multitasa	Comentarios
ODEXPERT (1993) [KEM93]	**P** (OPS, Fortran, y C)	ME	Selección automática del método. Interacción con *Maple* para la manipulación simbólica y pruebas al sistema de EDO's. Uso de una base de conocimientos para la toma de decisiones.
C++ Integ. C. (1995) [Con95]	**OO** (C++)	ME	Arquitectura de software para métodos Runge Kutta adaptativos.
MExtraPolaM (1997) [EL97]	**P** (Fortran)	MIM	Uso de métodos de extrapolación como alternativa para implementar métodos multitasa.
ODE++ (1998) [Mil98]	**OO** (C++)	ME	Arquitectura de software para la representación y solución de sistemas de EDO's. Ya no se encuentra disponible.
NetLib (1999) [ABB+99]	**P** (Fortran)	ME	Repositorio de software para cómputo científico.
DAE-TK (1999) [KM99]	**OO** (C++)	ME	Implementa la función *daspk* de Fortran para resolver EDA's (ecuaciones diferenciales algebraicas).
MRKI, MRKII (1999) [KR99]	**P** (Fortran)	MIM	Métodos Runge Kutta Multitasa, método multitasa altamente acoplado a los métodos Runge Kutta.
Protomol (2004) [MCH+04]	**OO** (C++)	MIM	Los métodos multitasa implementados en Protomol son formados a partir de métodos específicos para resolver problemas relacionados con la dinámica de moléculas. Uso de patrones de diseño para obtener flexibilidad en el software desarrollado.
ModSimPack (2006) [Bun06]	**P** (Modellica)	ME	Selección automática del método. Base de conocimientos para la toma de decisiones.
Arq. Mult. (2009)	**OO** (C++)	MIM	Arquitectura diseñada con patrones de diseño para lograr flexibilidad al cambio. Métodos multitasa formados a partir de métodos estándares para resolver sistemas de EDO's.

Tabla 2.1: Resumen del estado del arte. **P**: Procedimental; **OO**: Orientado a objetos; ME: Métodos estándar; MIM: Métodos Multitasa

Capítulo 3

Fundamentos en métodos numéricos para aproximar soluciones a EDOs

La ecuaciones diferenciales ordinarias han sido utilizadas desde los tiempos de Isaac Newton para describir el comportamiento de una gran variedad de fenómenos físicos dinámicos, Schilling *et al.* [SH99].

En este capítulo se presentan los conceptos básicos de los métodos numéricos que nos permiten aproximar soluciones a sistemas de ecuaciones diferenciales ordinarias de primer orden con valores de inicio. Se exponen los métodos numéricos más comunes como son el Método de Euler y los Runge Kutta. Se presentan los conceptos básicos sobre los Métodos de Integración Multitasa. Además, se muestra el uso del método de líneas, **MOL** por sus siglas en inglés, para generar sistemas de EDOs a partir de ecuaciones diferenciales parciales (**EDP**s).

3.1. Notación

A lo largo del capítulo los escalares se denotan por medio de y, g, ϕ, etc, y los vectores por medio de negritas, por ejemplo, $\mathbf{y}, \mathbf{g}, \boldsymbol{\phi}$, etc. Los componentes de un

vector s-dimensional se denotan por los elementos $y_i, i = 1, 2, \ldots, s$, y el vector $\mathbf{y}$ es denotado como $\mathbf{y} = [y_1, y_2, \ldots, y_s]^T$ donde el superíndice T denota el vector transpuesto. Se usa $y^{(m)}$ para denotar la m-ésima derivada de la función y.

3.2. Problemas de valores iniciales

Una *ecuación diferencial ordinaria* es una ecuación de la forma

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \quad \alpha < t < \beta \tag{3.2.1}$$

donde la $' = d/dt$, $t \in \mathbb{R}$ es la variable independiente, la variable dependiente $\mathbf{y} \in \mathbb{R}^s$ y $\mathbf{f}$ es una función conocida definida en algún subconjunto de $\mathbb{R}^{s+1}$.

Una solución de (3.2.1) es una función $\mathbf{g}(t)$ que satisface la ecuación (3.2.1). Es decir, tiene derivada y satisface $\mathbf{g}' = \mathbf{f}(t, \mathbf{g})$ para toda t en el intervalo (α, β).

Generalmente una ecuación del tipo definido por (3.2.1) dispone de un estado inicial conocido, el cual es llamado *condiciones de inicio*. Este se definen por medio del vector

$$\mathbf{y}(\alpha) = \boldsymbol{\eta} \tag{3.2.2}$$

Un *problema de valores iniciales* está definido por (3.2.1) y (3.2.2). En este trabajo nos enfocaremos a esta clase de problema.

Si la función $\mathbf{f}(t, \mathbf{y})$ se puede escribir como $\mathbf{f}(t, \mathbf{y}) = A(t)\mathbf{y} + \mathbf{B}(t)$ donde $A(t) \in \mathbb{R}^s \times s$ y $\mathbf{B}(t) \in \mathbb{R}^s$, entonces el sistema de EDOs es *lineal*.

3.2.1. Reducción de orden en ecuaciones diferenciales

El orden de la derivada más alta dentro de la ecuación diferencial es llamado el orden de la ecuación diferencial. Esta sección presenta un método para reducir ecuaciones diferenciales ordinarias de orden superior a un sistema de ecuaciones de primer orden. Posteriormente se ejemplifica su uso.

La mayoría de los fenómenos físicos pueden ser modelados utilizando EDOs de orden superior a uno. Generalmente, es posible convertir estas ecuaciones a sistemas de EDOs de primer orden mediante un cambio de variables adecuado.

Una posible estrategia es la definida por Sánchez en [Sán79], dada una ecuación de la forma

$$y^{(k)} = f(t, y, y'', \ldots, y^{(k-1)}) \quad k > 1 \tag{3.2.3}$$

de orden k, se definen las variables $x_1, x_2, \ldots, x_k$ por medio de

$$\begin{array}{rl|rll} x_1 &= y & x_1' &= y' & = x_2 \\ x_2 &= y' & x_2' &= y'' & = x_3 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{k-1} &= y^{(k-2)} & x_{k-1}' &= y^{(k-1)} & = x_k \\ x_k &= y^{(k-1)} & x_k' &= y^{(k)} & = f(t, x_1, x_2, \ldots, x_k) \\ & & & & = f(t, \mathbf{x}). \end{array} \tag{3.2.4}$$

El cambio de variables anterior da origen a un sistema de primer orden

$$\begin{aligned} \mathbf{x}' &= \mathbf{f}(t, \mathbf{x}) \\ \mathbf{x} &= [x_1, x_2, \ldots, x_k]^T \\ \mathbf{f}(t, \mathbf{x}) &= [f_1(t, \mathbf{x}), \ldots, f_k(t, \mathbf{x})]^T \\ &= [x_2, \ldots, x_k, f(t, \mathbf{x})]^T \end{aligned}$$

Ejemplo

Considere el siguiente problema con valores iniciales definido en el intervalo $I = (\alpha, \beta)$

$$ay'' + by' + cy = \mu(t) \quad y(\alpha) = \eta_1 \quad y'(\alpha) = \eta_2 \tag{3.2.5}$$

Definimos las nuevas variables $x_1 = y, x_2 = y'$, entonces:

$$x_1' = x_2 = y' \tag{3.2.6}$$

$$x_2' = y'' \tag{3.2.7}$$

Al sustituir (3.2.6) y (3.2.7) en (3.2.5) se obtiene:

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= -\frac{(bx_2 + cx_1)}{a} + \mu(t) \\ x_1(\alpha) = \eta_1 \quad & x_2(\alpha) = \eta_2 \end{aligned}$$

El sistema anterior puede ser representado en notación matricial de la siguiente manera

$$\mathbf{x}' = -\begin{bmatrix} 0 & 1 \\ \frac{c}{a} & \frac{b}{a} \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ \mu \end{bmatrix} \tag{3.2.8}$$

De acuerdo a [LeV07], en ocasiones es útil eliminar cualquier dependencia explícita de $\mathbf{f}$ en t introduciendo una nueva variable que sustituya a t. En nuestro ejemplo definimos $x_3 = t$, entonces $x_3' = 1$, por lo tanto $x_3(\alpha) = 0$.

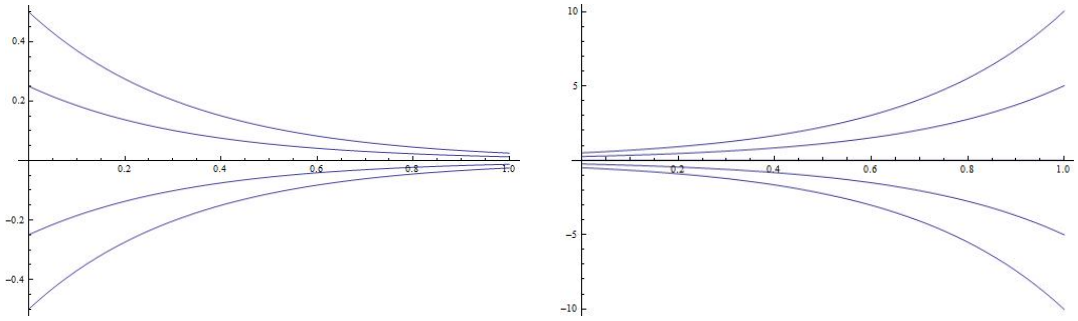
3.2.2. Existencia y unicidad

Antes de resolver un problema utilizando un método numérico se debe comprobar que el problema tiene una solución, de hecho, un problema que tiene una solución se dice que es un problema *bien planteado* a la Jacques Hadamard, ésto significa que el problema tiene una solución única que depende continuamente en los datos usados para definir el problema.

Para garantizar la existencia de una única solución, se requiere que $\mathbf{f}$ sea suave. Es decir, que la función sea Lipschitz continua en $\mathbf{y}$ en algún rango de t e $\mathbf{y}$. Es decir, existe una constante $L > 0$ tal que para toda $(t, \mathbf{y}) \in D$ se cumple

$$|\mathbf{f}(t, \mathbf{y}) - \mathbf{f}(t, \mathbf{y}^*)| \leq L|\mathbf{y} - \mathbf{y}^*| \tag{3.2.9}$$

Esto quiere decir que $|\mathbf{f}(t, \mathbf{y}) - \mathbf{f}(t, \mathbf{y}^*)| \to 0$ conforme $|\mathbf{y} \to \mathbf{y}^*|$. A la constante L se le llama la constante de *Lipschitz* de la función $\mathbf{f}$.



(a) Converge (b) Diverge

Figura 3.1: Curvas solución para $y' = \lambda f(t, y)$, en (a) con $\lambda = -3$ y en (b) con $\lambda = 3$.

Cuando la función $\mathbf{f}$ posee derivada continua respecto a $\mathbf{y}$ para toda $(t, \mathbf{y}) \in D$ la constante de Lipschitz L se puede definir por

$$L = \sup_{(t,\mathbf{y})\in D} \left| \frac{\partial \mathbf{f}(t, \mathbf{y})}{\partial \mathbf{y}} \right| \tag{3.2.10}$$

La constante de Lipschitz mide que tanto $\mathbf{f}(t, \mathbf{y})$ cambia si $\mathbf{y}$ es perturbada. Consideremos la ecuación $\mathbf{y}' = \lambda \mathbf{f}(t, \mathbf{y})$, con condición inicial $\mathbf{y}(\alpha)$, por (3.2.10) $L = |\lambda|$. La solución de la ecuación diferencial está dada por $\mathbf{y} = \mathbf{y}(\alpha)e^{\lambda t}$. Analizaremos los casos para $\lambda < 0$ y $\lambda > 0$. En las figuras 3.1(a) y 3.1(b) se presentan las curvas solución para el caso donde λ es negativa y positiva respectivamente. En ambos casos, la constante de Lipschitz es la misma, $L = |\lambda|$. Sin embargo, ya que $\mathbf{f}(t, \mathbf{y}) = \mathbf{y}'$, la pendiente de la línea tangente a la curva solución, depende de la variación de $\mathbf{y}$, la pendiente variará si perturbamos $\mathbf{y}$. En el primer caso para $\lambda < 0$ la solución $\mathbf{y}$ tiene pendiente negativa. Las soluciones con diferentes condiciones iniciales, perturbaciones en $\mathbf{y}$, convergerán a una misma solución para t muy grande, luego $\mathbf{f}(t, \mathbf{y}) = \mathbf{y}'$ es poco sensible a perturbaciones en $\mathbf{y}$. En el segundo caso $\lambda > 0$, se tiene la situación inversa, luego cualquier cambio en $\mathbf{y}$ afectará fuertemente a $\mathbf{f}(t, \mathbf{y}) = \mathbf{y}'$. Ver [LeV07].

3.3. Métodos para resolver EDOs numéricamente

Sin pérdida de generalidad, a partir de esta sección la función f queda definida como $f : \mathbb{R}^2 \to \mathbb{R}$.

El objetivo de los métodos numéricos es aproximar la solución de una ecuación diferencial de la forma $y' = f(t, y), \alpha < t < \beta$ con condiciones de inicio $y(\alpha) = \eta$. El método numérico debe encontrar una función discreta (o conjunto de puntos) que satisfaga las condiciones iniciales dadas en una región espacio-tiempo.

Una explicación detallada sobre error, estabilidad y convergencia de los métodos numéricos se presenta en el apéndice A.

3.3.1. Algunos métodos numéricos básicos

El método numérico calcula, a partir de ciertos datos iniciales U_0, las aproximaciones $U_1, U_2, \ldots, U_n$ de modo que se satisfaga $U_n = y(t_n)$.

El método más conocido es el *Método de Euler* o también llamado *Forward Euler*. Éste consiste en sustituir y' por la fórmula de diferencias $(U_{n+1} - U_n)/h$. Donde h representa la distancia entre el punto U_{n+1} y U_n, de modo que $t_n = nh$ para $n \geq 0$.

El método de Euler queda definido entonces como

$$U_{n+1} = U_n + hf(t, U_n) \quad n = 0, 1, \ldots \tag{3.3.1}$$

Otro de los métodos más conocidos es el *Backward Euler*, éste queda definido de la siguiente manera

$$U_{n+1} = U_n + hf(t, U_{n+1}) \quad n = 0, 1, \ldots \tag{3.3.2}$$

A los métodos como éste, donde el valor de U_{n+1} se define en función del mismo U_{n+1}, se les conoce como *métodos implícitos*, mientras que los métodos tipo Euler, donde el valor U_{n+1} se define en función de sus valores anteriores, son conocidos como *métodos explícitos*.

Un método implícito también conocido es la *regla trapezoidal*, la cual se define

como

$$U_{n+1} = U_n + \frac{1}{2}h\left[f(t, U_n) + f(t, U_{n+1})\right] \quad n = 0, 1, \ldots \tag{3.3.3}$$

3.3.2. Error de truncamiento

El *error de truncamiento*, se obtiene al reemplazar el valor aproximado U_i, por el valor verdadero de la función $y(t_i)$ en el método numérico.

Por ejemplo, para el caso del Método de Euler a partir de la ecuación (3.3.1) y aplicando una expansión en series de Taylor se obtiene

$$\begin{aligned}
\tau_n &= \frac{y(t_{n+1})}{h} - \frac{y(t_n)}{h} - f(t_n, y_n)) \\
&= \frac{1}{h}\left[y(t_n) + y'(t_n)h + \frac{1}{2}h^2 y''(t_n) + O(h^3)\right] - \frac{y(t_n)}{h} - f(t_n, y_n)) \\
&= \frac{1}{2}hy''(t_n) + O(h^2)
\end{aligned}$$

Por lo que el error de truncamiento para el Método de Euler se comporta como $O(h)$, es decir, es un método con exactitud de primer orden.

Análogamente podemos definir la exactitud de los métodos *Forward Euler* y el *Backward Euler*. En este caso, ambos cuentan con exactitud de primer orden, es decir, el error se comporta como $O(h)$. En el caso de la *regla trapezoidal* la exactitud es de segundo orden, es decir, el error se comporta como $O(h^2)$.

3.3.3. Métodos multipaso lineales

Los métodos numéricos hasta ahora mencionados son miembros de una familia de métodos llamada *Métodos Multipaso Lineales*. Un método de esta familia con r pasos tiene la forma:

$$\sum_{j=0}^{r} \alpha_j U_{n+j} = h \sum_{j=0}^{r} \beta_j f(t_{n+j}, U_{n+j}) \tag{3.3.4}$$

El valor U_{n+r} es calculado utilizando los valores anteriores $U_{n+r-1}, U_{n+r-2}, \ldots, U_n$. Para normalizar la ecuación (3.3.4) se supone a $\alpha_r = 1$. Si $\beta_r = 0$ entonces tenemos un método explícito, de otro modo será un método implícito.

Por ejemplo, para derivar el Método de Euler definimos los valores $r = 1, \alpha_0 = -1, \beta_0 = 1$ y como es un método explícito entonces $\beta_1 = 0$.

Al sustituir en la ecuación (3.3.4) obtenemos el Método de Euler

$$\begin{aligned} \alpha_0 U_n + \alpha_1 U_{n+1} &= h\left[\beta_0 f(t_n, U_n) + \beta_1 f(t_{n+1}, U_{n+1})\right] \\ \alpha_0 U_n + \alpha_1 U_{n+1} &= h\beta_0 f(t_n, U_n) \\ U_{n+1} &= U_n + h f(t_n, U_n) \end{aligned}$$

3.3.4. Métodos Runge Kutta

Existen métodos de un solo paso capaces de obtener exactitudes mayores a las de primer orden. Para lograrlo calculan un conjunto de estados intermedios entre U_n y U_{n+1} por lo que son también conocidos como *métodos multi-estado*. El ejemplo más claro de ellos son los métodos Runge Kutta explícitos.

Tomando la notación de Lambert [Lam73], un método Runge Kutta de R estados queda definido de la siguiente manera:

$$\begin{aligned} y_{n+1} - y_n &= h\phi(t_n, y_n, h) \\ \phi(t_n, y_n, h) &= \sum_{r=1}^{R} c_r k_r \\ k_1 &= f(t_n, y_n) \\ k_r &= f(t_n + h a_r, y_n + h \sum_{s=1}^{r-1} b_{rs} k_s), \quad r = 2, 3, \ldots, R \\ a_r &= \sum_{s=1}^{r-1} b_{rs} \quad r = 2, 3, \ldots, R \end{aligned}$$

Se cumple que $\sum_{r=1}^{R} c_r = 1$. Dependiendo de la selección de los coeficientes para los vectores $\mathbf{a}, \mathbf{c}$ y la matriz $\mathbf{b}$ se obtiene un Runge Kutta particular.

La representación de los coeficientes a manera de matriz es la siguiente:

$$
\begin{array}{c|cccc}
a_1 & b_{11} & & & \\
a_2 & b_{21} & b_{22} & & \\
\vdots & \vdots & \ddots & & \\
a_r & b_{r1} & b_{r2} & \dots & b_{rr} \\
\hline
 & c_1 & c_2 & \dots & c_r
\end{array}
\tag{3.3.5}
$$

Entre los métodos Runge Kutta más conocidos se encuentran el Runge Kutta 2 y el Runge Kutta 4, cuyos coeficientes se muestran a continuación.

Coeficientes Runge - Kutta 2

$$
\begin{array}{c|cc}
1 & \frac{1}{2} & \\
\hline
 & \frac{1}{2} & \frac{1}{2}
\end{array}
\tag{3.3.6}
$$

Coeficientes Runge - Kutta 4

$$
\begin{array}{c|ccc}
\frac{1}{2} & \frac{1}{2} & & \\
\frac{1}{2} & 0 & \frac{1}{2} & \\
1 & 0 & 0 & 1 \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}
\tag{3.3.7}
$$

3.4. Métodos Multitasa

En el modelado de sistemas físicos dinámicos mediante ecuaciones diferenciales es posible encontrar componentes de la solución que varían rápidamente, es decir, que presenten altas frecuencia de oscilación y/o transitorios rápidos, mientras que las componentes restantes de la solución presentan variaciones relativamente lentas. A este tipo de sistemas se les conoce como **sistemas con dos escalas temporales**, Rodríguez [RG98].

El uso de técnicas convencionales o métodos como los presentados en las secciones anteriores implica que la parte con variación rápida imponga el tamaño del paso de integración h, generalmente pequeño respecto a las componentes lentas, resultando en un costo computacional alto para aproximar la solución de estas últimas.

Para obtener una precisión requerida, existe un valor máximo para h el cual está dado por las componentes cuya solución varía rápidamente. La idea detrás de un Método Multitasa es explotar los diferentes tipos de variación presentes en la solución: pocas variaciones rápidas y muchas variaciones lentas. El acoplamiento entre los dos tipos de variación se mantiene por medio de interpolar las componentes que presenten variación lenta, Skelboe [Ske84].

Ejemplos de la clase de problemas que dan lugar a este tipo de sistemas se encuentran al modelar la interacción de dos sistemas físicos diferentes. Por decir algo, un sistema de control y un sistema a ser controlado, donde el primero representa el sub-sistema rápido y el segundo el sub-sistema lento. Otro caso es la simulación de órbitas celestiales donde cuerpos con una alta frecuencia de variación angular orbitan un cuerpo central.

El problema de la imposición del paso por parte de la componente rápida resulta en un alto costo computacional para la parte lenta. La solución a este problema se encuentra *partiendo* al sistema de EDOs en dos sub-sistemas, la parte con variación rápida y la parte con variación lenta.

La integración de los subsistemas se realiza utilizando un paso relativamente grande para la parte lenta y un paso pequeño para la parte rápida. A esta técnica de integración se le conoce como *Método de Integración Multitasa*, Gear [Gea81], Rodríguez [RG98]. Un método de integración multitasa es un algoritmo que permite integrar cada ecuación en un sistema de EDOs con un paso y un método adecuado a su comportamiento.

Consideremos el siguiente problema con valores de inicio con n ecuaciones

$$\mathbf{y}' = \mathbf{q}(t, \mathbf{y}) \quad \mathbf{y}(\alpha) = \boldsymbol{\eta} \tag{3.4.1}$$

supongamos que (3.4.1) puede dividirse en dos subsistemas, sea entonces:

$$\mathbf{y}' = \begin{pmatrix} \boldsymbol{\mu}' \\ \boldsymbol{\nu}' \end{pmatrix} = \begin{pmatrix} \mathbf{f}(t, \boldsymbol{\mu}, \boldsymbol{\nu}) \\ \mathbf{g}(t, \boldsymbol{\mu}, \boldsymbol{\nu}) \end{pmatrix} \quad \mathbf{y}(\alpha) = \begin{pmatrix} \boldsymbol{\eta_f} \\ \boldsymbol{\eta_g} \end{pmatrix} \tag{3.4.2}$$

donde $\mathbf{f} : \mathbb{R} \times \mathbb{R}^r \times \mathbb{R}^s \to \mathbb{R}^r$ y $\mathbf{g} : \mathbb{R} \times \mathbb{R}^r \times \mathbb{R}^s \to \mathbb{R}^s$. Se cumple que $r + s = n$. El subsistema $\mathbf{f}(t, \boldsymbol{\mu}, \boldsymbol{\nu})$ representa a la parte rápida y $\mathbf{g}(t, \boldsymbol{\mu}, \boldsymbol{\nu})$ a la parte lenta.

Al aplicar un método multipaso lineal al método multitasa y hacer la sustitución $\lambda = k(n+1)$ tenemos:

$$\mu_\lambda = -\sum_{j=0}^{k} \alpha_{kj} \mu_{\lambda-j-1} + h\beta_{kj} f(t_{\lambda-j-1}, \mu_{\lambda-j-1}, \hat{\nu}_{\lambda-j-1}) \tag{3.4.3}$$

$$\nu_{i\lambda} = -\sum_{j=0}^{ki} \alpha_{ki} \nu_{i\lambda-j-1} + h\beta_{ki} g(t_{i\lambda-j-1}, \mu_{i\lambda-j-1}, \nu_{i\lambda-j-1}) \tag{3.4.4}$$

Para (3.4.4), $i = 1, \ldots, r$, donde r es el número de pasos del método multipaso lineal. Para cada paso de integración $H = ih$ del método lento $\mathbf{g}(t, \boldsymbol{\mu}, \boldsymbol{\nu})$, tendremos i pasos de integración h del método rápido $\mathbf{f}(t, \boldsymbol{\mu}, \boldsymbol{\nu})$. Esto se ejemplifica en la figura 3.2.

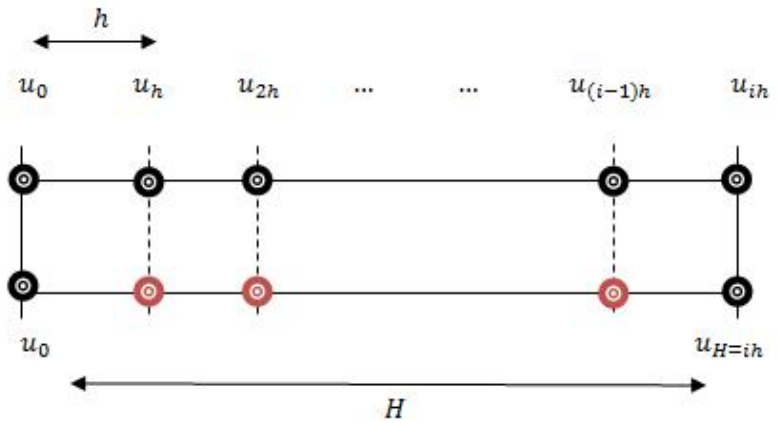


Figura 3.2: Método multitasa, la parte rápida se integra $i - veces$ contra una de la parte lenta.

El uso de los métodos multitasa se justifica cuando el costo de evaluación de la función $\mathbf{g}(t, \boldsymbol{\mu}, \boldsymbol{\nu})$ es alto o la dependencia entre las componentes rápidas y lentas es poca. En general, el segundo caso se presenta cuando el sistema de EDOs se obtiene al discretizar un problema de EDPs por medio de alguna técnica como el método de líneas, Gear [Gea81].

3.4.1. Algoritmos Multitasa

Un Método de Integración Multitasa puede ser implementado bajo tres esquemas diferentes: *interpolación*, *información adelantada* o *información atrasada*. El algoritmo utilizado para aproximar una solución a (3.4.2) dictará la manera en que se calcula el valor para $\hat{\boldsymbol{\nu}}$ de la ecuación (3.4.3).

- *Interpolación*, $\hat{\boldsymbol{\nu}}$ se calcula a través de interpolación.
- *Información adelantada*, $\hat{\boldsymbol{\nu}}$ se toma como el valor calculado para $\boldsymbol{\nu}_{(i)\lambda}$.
- *Información atrasada*, $\hat{\boldsymbol{\nu}}$ se toma como el valor calculado para $\boldsymbol{\nu}_{(i-1)\lambda}$.

3.4.2. Estabilidad en los métodos multitasa

En lo que se refiere a la teoría que defina las condiciones bajo las cuales los métodos multitasa sean estables muy pocos trabajos se han desarrollado.

En el trabajo de Rodríguez [RG02], se muestra un análisis de los trabajos realizados en este ámbito además de presentar las condiciones necesarias y suficientes que garantizan la estabilidad absoluta de los métodos multitasa lineales semi-implícitos.

En la mayoría de los trabajos revisados se utilizan sistemas de dos ecuaciones donde el acoplamiento entre éstas es generalmente débil. Los valores de la componente lenta utilizados por la componente rápida son calculados mediante interpolación. Las matrices que definen al sistema de ecuaciones son de tipo triangular además de que en ninguno se encuentran criterios generales para definir la estabilidad absoluta de los métodos multitasa.

En particular, en el trabajo de Wells [Wel82] se presentan recomendaciones como la de integrar las componentes lentas primero y predecir el valor de las componentes rápidas mediante extrapolación. En el trabajo de Kvaernø[Kvæ00] se presenta el análisis realizado a métodos Runge Kutta bajo el esquema Multitasa. Sin embargo, al igual que los otros trabajos, este análisis se realiza en sistemas donde el acoplamiento es débil.

3.5. Método de Líneas

El método de líneas es una técnica para resolver EDPs por medio de discretización en una dimensión y permite integrar el problema semi-discretizado como un sistema de EDOs. Al discretizar en el espacio se obtiene un sistema de EDOs donde cada componente corresponde a la solución en algún punto del intervalo espacial de la EDP.

Entre las ventajas de obtener un sistema de EDOs se encuentra la de utilizar software de propósito general para resolver el sistema obtenido.

Ejemplo

A continuación aplicaremos el método de líneas a la ecuación de calor

$$\begin{aligned} \mu_t &= \mu_{xx} \\ \mu(x,0) &= \eta(x) \\ \mu(0,t) &= g_0(t) \quad t > 0 \\ \mu(1,t) &= g_1(t) \quad t > 0 \end{aligned}$$

Al discretizar en el espacio con intervalo $\Delta x = x_{i+1} - x_i$ obtenemos

$$U_i' = \frac{1}{h^2}(U_{i-1(t)} - 2U_i(t) + U_{i+1}(t)), \quad i = 1, 2, \ldots, m \tag{3.5.1}$$

Con esta nueva interpretación obtenemos un sistema de m EDOs para las variables $U_i(t)$, ver figura 3.3.

El sistema puede ser reescrito como

$$\mathbf{U}'(t) = \mathbf{A}\mathbf{U}(t) + \mathbf{g}(t) \tag{3.5.2}$$

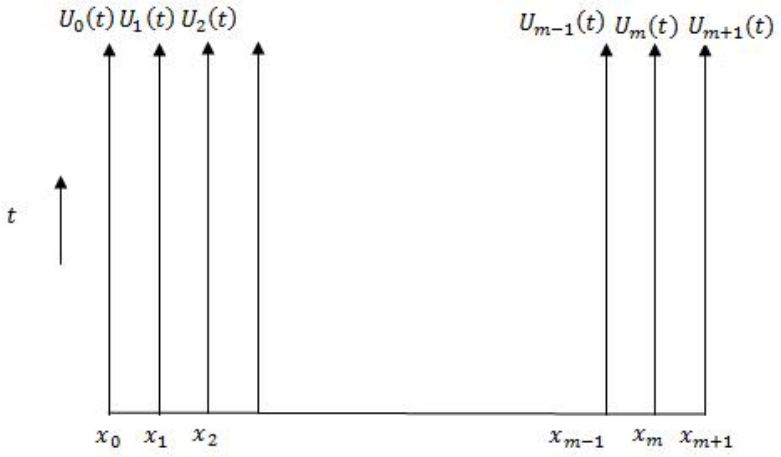


Figura 3.3: Interpretación del Método de Líneas. Cada punto x_i en la malla representa una EDO.

donde $\mathbf{A}$ y $\mathbf{g}(t)$ están definidos como:

$$A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 \end{bmatrix} \qquad g(t) = \frac{1}{h^2}\begin{bmatrix} g_0(t) \\ 0 \\ 0 \\ \vdots \\ 0 \\ g_1(t) \end{bmatrix} \tag{3.5.3}$$

Por cada punto x_i en la malla tenemos una EDO.

En esta sección se han presentado los conceptos fundamentales en lo que respecta a métodos numéricos, tanto estándares como multitasa. La mayoría de estos algoritmos son aún implementados bajo el enfoque procedimental por lo que su reutilización resulta complicada.

Capítulo 4

Fundamentos en desarrollo de software con Patrones de Diseño

En este capítulo se presentan conceptos básicos relacionados con el diseño de software orientado a objetos y el uso de patrones para esta tarea. Se exponen los elementos que definen a un patrón y los beneficios obtenidos al aplicarlos en la tarea de diseño de software.

4.1. Diseño de software orientado a objetos

El paradigma de diseño orientado a objetos surge de la necesidad de enfrentar las dificultades presentadas en otros modelos de diseño. Por mencionar algunas: integración de cambios o nuevas características al sistema, alta dependencia entre los componentes, dificultad en el mantenimiento, sistemas *ad hoc* sin posibilidad a reutilización, entre otros.

El diseño de software orientado a objetos ayuda a crear sistemas reutilizables, de fácil mantenimiento y con baja dependencia entre sus componentes lo que facilita tanto la integración de nuevas características como la modificación del sistema de software. Esto lo logra al crear sistemas formados por objetos los cuales se comunican para ejecutar tareas complejas.

De acuerdo a Shalloway *et al.* [ST04], el enfoque tradicional define a un objeto como *datos y métodos*. Sin embargo, esta perspectiva es limitada ya que se puede pensar al objeto como una *entidad con responsabilidades*. Este enfoque amplía las posibilidades en el diseño de sistemas de software al tener la ventaja de poder definir entidades que sean responsables de sí mismas.

4.1.1. Una perspectiva diferente en el desarrollo de software orientado a objetos

Entre las tareas en el proceso de desarrollo de software se encuentra la del diseño de software, la cual, de acuerdo a Gamma *et al.* [GHJV95], es una tarea difícil que tiende a complicarse cuando se desea diseñar software reutilizable. El uso de las técnicas ofrecidas por el paradigma orientado a objetos no garantiza la generación de un diseño reutilizable, por lo que es necesario utilizar nuevas técnicas como las ofrecidas por los patrones de diseño.

En el trabajo de Shalloway *et al.* [ST04] se presenta una perspectiva de diseño orientada a objetos que supera a la utilizada en los 90's para el mismo propósito. Ésta se muestra en la tabla 4.1.

Perspectiva	*Descripción*
Conceptual	Conceptos en el dominio del problema. Bajo esta perspectiva no se menciona la manera en que el software debe ser implementado.
Especificación	Relacionada con el software pero a nivel de interfaces no de implementación.
Implementación	Bajo esta perspectiva se habla del código.

Tabla 4.1: Perspectivas en el proceso de desarrollo de software.

Podemos definir a un *objeto*, elemento clave en el diseño de software orientado a objetos, en cada uno de los niveles presentados de la siguiente manera:

- A nivel *conceptual* como un conjunto de responsabilidades.

- A nivel de *especificación* como un conjunto de métodos que pueden ser utilizados por él mismo o por otro objeto.
- A nivel de *implementación* como código que define funciones y datos.

Paradigma orientado a objetos bajo la perspectiva presentada

Para complementar la perspectiva presentada y lograr una visión diferente en el desarrollo de software orientado a objetos, se muestra a continuación una serie de definiciones generadas a partir del enfoque expuesto para las características más representativas del paradigma de programación mencionado.

Se entiende por una *clase* a un repositorio de métodos junto con la definición de los datos miembro para los objetos que la instancían, es decir, para los objetos que son creados a partir de la clase.

A nivel conceptual, una *clase abstracta* es una clase sustituta para otras clases. Proveen de un nombre común para un conjunto de clases relacionadas, esto permite tratarlas como un solo concepto. Una clase es una *clase concreta* cuando representa una implementación específica o no modificable de un concepto.

A la relación donde la **clase B** es una implementación específica de la **clase A** se le conoce como una relación tipo *es un* y se conoce formalmente como *herencia*. Otros nombres conocidos para esta relación son, la **clase B** *deriva*, *especializa* o es una *subclase* de la **clase A**.

En el paradigma orientado a objetos de los 90's, el *encapsulamiento* es conocido como el ocultamiento de datos. Al utilizar la perspectiva presentada por Shalloway *et al.* [ST04] es posible redefinirlo como cualquier tipo de ocultamiento. Por ejemplo, al contar con una clase abstracta, **CAbs**, y una clase concreta, **CCncrt** que derive o sea una subclase de **CAbs**, se oculta a la clase concreta cuando se utiliza a la clase **CAbs** para referirse a **CCncrt**.

El hecho de que oculte no sólo información, sino también implementaciones implica que el usuario no tendrá que preocuparse sobre los detalles de éstas. Otra de las ventajas que presenta el encapsulamiento bajo el enfoque en que los objetos son

responsables de ellos mismos, es que, la única manera de afectar a un objeto es a través de los métodos que presenta. Por lo tanto, ayuda a prevenir el acceso no controlado a los datos o valores miembro de los objetos, evitando así, los tan no queridos *efectos colaterales*.

Entre los conceptos importantes del paradigma orientado a objetos se encuentra el *polimorfismo*. Comúnmente, se define como la capacidad de un objeto para comportarse de manera diferente en función de la clase de la que fue instanciado. Con la perspectiva presentada se entenderá al polimorfismo como la habilidad de *objetos relacionados* para implementar métodos específicos para su tipo.

4.1.2. Desarrollo de software flexible

Entre los principales problemas en la tarea de desarrollo de software se encuentran los de crear sistemas con posibilidades a extensión, capacidad de adaptación a nuevas tecnologías, bajo costo de mantenimiento, etc. Estas capacidades se pueden resumir en una sola, *adaptación al cambio*.

A partir de los requerimientos de un sistema de software, que son la base para el diseño e implementación de éste, debería ser posible identificar los posibles cambios y necesidades futuras del sistema desarrollado, sin embargo, y de acuerdo a Shalloway [ST04], a partir de esta etapa se presentan problemas para los desarrolladores del software ya que, generalmente, los requerimientos están incompletos, contienen errores, son mal entendidos y no presentan una perspectiva general del problema. Por lo tanto, y ya que en general no es posible identificar los posibles cambios a través de los requerimientos, es necesario diseñar sistemas que permitan la modificación o integración de nuevas características, requerimientos futuros, con un mínimo de rediseño posible.

Crear un diseño flexible no es una tarea sencilla para desarrolladores no experimentados, sólo los diseñadores con experiencia pueden lograr buenos diseños al primer intento. Los diseñadores expertos saben que no deben resolver un problema desde cero sino reutilizar soluciones que hayan funcionado en trabajos pasados. Una

vez que se encuentra una buena solución se utiliza una y otra vez. En estas soluciones se encuentran patrones de clases y/o comunicación entre objetos. Estos patrones resuelven problemas de diseño específicos y hacen de un diseño orientado a objetos más flexible, comprensible y reutilizable.

Los patrones de diseño de software, de acuerdo a Gamma *et al.* [GHJV95], ayudan en la tarea de crear arquitecturas orientadas a objetos reutilizables, primero, al seleccionar alternativas de diseño que generen sistemas reutilizables, y segundo, al evitar aquellas que comprometen la capacidad de reutilización del diseño. Los patrones de diseño y las técnicas de diseño orientado a objetos se fortalecen entre sí, ya que es necesario conocer las bases del diseño orientado a objetos para utilizar los patrones de diseño y, por el otro lado, al entender la esencia de los patrones de diseño se amplía la visión en el diseño de software orientado a objetos.

4.2. Patrones de Diseño

Christopher Alexander, arquitecto reconocido por muchos de sus diseños y padre del lenguajes de patrones en el ámbito de ciencias computacionales, se pregunta en su libro *The timeless way of building*, [Ale79] si es posible identificar a los factores que hacen clasificar a un diseño arquitectural como bueno o malo. Él responde que sí, debido a que se puede describir la belleza a través de bases objetivas que pueden ser medidas. La creencia de Alexander en que la calidad en un diseño es objetiva, nos permitiría identificar los elementos que hacen de un diseño ser bueno o malo.

Al realizar observaciones en las construcciones, Alexander descubrió que las creaciones arquitecturales consideradas como buenas presentaban características en común, y entendió que existen ciertas estructuras que no pueden ser separadas del problema que intentan resolver. Al buscar por las estructuras que fueron diseñadas para resolver un mismo problema, encontró similaridades entre los diseños que hacían considerarlos de alta calidad, a estas similaridades les llamó *patrones*.

En el libro *A pattern languaje* [AIS77], Christopher Alexander *et al.* definen a

un patrón como: «*Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente junto con la esencia de la solución a ese problema, de manera que pueda ser utilizada más de un millón de veces sin ser implementada dos veces de la misma manera*». Aunque en esta definición, Alexander se refiere a los patrones en el ámbito de la construcción, también aplica para los patrones en el diseño orientados a objetos. Para el caso del diseño de sistemas de software, las soluciones son expresadas en términos de la organización y comunicación entre objetos, además de proponer la especificación de las interfaces de los objetos involucrados. La parte esencial de un patrón es la solución que propone para un problema en un contexto dado.

En la época de los 90's, algunos desarrolladores de software que conocieron el trabajo de Alexander se preguntaban si era posible aplicar las ideas de patrones para el diseño de software. En particular querían saber: primero si ¿existen problemas de diseño de software, que ocurran una y otra vez, donde las soluciones empleadas presentan elementos similares? y segundo, si ¿es posible diseñar software en términos de patrones? La respuesta a estas preguntas fue afirmativa y con el trabajo de Gamma *et al.* [GHJV95] se aplica la idea de los patrones de diseño, en el ámbito de la Arquitectura, al diseño de sistemas de software.

En el trabajo mencionado, [GHJV95], solamente se identifican los patrones que ya existían en la comunidad de diseño de software. Estos patrones reflejan lo aprendido en la tarea del diseño de software de alta calidad. La idea detrás de los patrones de diseño es que la calidad de un sistema de software pueda ser medida de manera objetiva, [ST04].

Con base en lo expuesto por Gamma *et al.* en [GHJV95], un patrón presenta cuatro elementos fundamentales:

1. Un *nombre* que, de ser posible, describa el problema de diseño, la solución, y las consecuencias de su uso.
2. Un *problema*, el cual describe en que circunstancias aplicar el patrón.
3. La *solución* que detalla los elementos que conforman el patrón de diseño, sus relaciones, responsabilidades y colaboraciones. El patrón provee una descripción abstracta del problema de diseño y como la colaboración de diferentes elementos lo resuelven.

4. Las *consecuencias*, resultados y desventajas de aplicar el patrón, así como los elementos a considerar para su integración al diseño. En el caso de patrones de diseño de software, este punto se refiere al espacio y tiempo utilizados por el patrón, además de mencionar el impacto en la flexibilidad, extensibilidad y portabilidad del sistema.

En resumen, un patrón debe tener siempre un nombre y un propósito. Es importante presentar de manera explícita el problema a resolver, el cual es la razón de la existencia del patrón. Cada patrón presenta una solución simplificada para el problema expuesto. La importancia principal de un patrón es que ayuda a los diseñadores novatos a tomar ventaja de la experiencia de otros. Al utilizar un patrón se incluyen los elementos que hacen considerar a un diseño como bueno y se evitan aquellos que harían considerarlo como malo.

4.2.1. Patrones en Ingeniería de Software

Los patrones de diseño están relacionados con la etapa de análisis y diseño de software, ayudan a afrontar los problemas en estas etapas definiendo las interfaces y las comunicaciones entre los objetos. Además, algunos libros relacionados con el tema presentan sugerencias para la etapa de implementación de los patrones para diferentes lenguajes de programación, tal es el caso de [Coo00] y [Hol06], por mencionar algunos.

Entre las ventajas obtenidas al utilizar patrones podemos mencionar las más sobresalientes, primera, la nueva perspectiva ofrecida por los patrones libera a los diseñadores de software de preocuparse por los detalles del diseño en las primeras etapas del proceso de desarrollo de software. Los patrones nos ayudan a ver el bosque entre los árboles, [ST04]. Segunda, el uso de patrones permite reutilizar soluciones diseñadas por expertos, es decir, ayudan a evitar el rediseño de soluciones para problemas comunes. Y tercera, los patrones ayudan a establecer una *terminología*, *lingua franca*, en la comunidad de diseño de software, de modo que proveen un punto de referencia en las etapas de análisis y diseño de un sistema de software.

En general, en el área de Ingeniería de Software se obtienen beneficios al utilizar patrones para el diseño de sistemas de software ya que éstos, además de las ventajas

presentadas, proponen una metodología de diseño al identificar las partes que conforman al sistema de software. Por medio del uso de patrones es posible identificar las relaciones que guardan los elementos participantes en el sistema de software.

De acuerdo a Gamma *et al.* [GHJV95], los patrones de diseño son *descripciones de comunicación entre objetos y clases* que son especializadas al momento de resolver un problema de diseño general para un contexto particular. Un patrón de diseño nombra, abstrae e identifica los aspectos clave de una estructura de diseño común que la hacen útil para crear diseños orientados a objetos reutilizables. Los patrones de diseño identifican las clases e instancias participantes, su papel y colaboración, además de la distribución de responsabilidades.

Patrones de arquitectura

En el nivel más alto de abstracción se encuentran los patrones de arquitectura, éstos, ayudan a especificar la estructura fundamental de la aplicación por lo que las actividades de desarrollo posteriores serán gobernadas por esta estructura. Los patrones de arquitectura pueden ser considerados como plantillas para arquitecturas de software concretas, [CC08].

La selección de un patrón de arquitectura debe ser encaminada por las propiedades generales de la aplicación a desarrollar. Es necesario, entonces, saber si se desarrollará un sistema interactivo o estático, si estará distribuido en una red de computadoras u operará en una sola, entre otras. Debido a que la mayoría de los sistemas de software no pueden ser construidos a partir de un solo patrón de arquitectura, en ocasiones será necesario combinarlos para lograr los objetivos del sistema.

Uno de los patrones de arquitectura más conocido es el definido por la triada Modelo/Vista/Controlador (**MVC**), este patrón es generalmente utilizado para el diseño de interfaces de usuario. Consiste de tres subsistemas: el modelo, la vista y el controlador. El modelo contiene los datos con los que se trabaja, la vista se refiere a la forma de presentar los datos y el controlador define la forma en que la interfaz de usuario responde a las entradas. Al hacer esta separación, se incrementa

la flexibilidad y reutilización ya que el acoplamiento entre los elementos es débil.

4.2.2. Clasificación de los Patrones de diseño

En el libro de Gamma *et al.* [GHJV95] se presentan 23 patrones de diseño de software, éstos se listan a manera de catálogo mostrando puntos clave como *nombre*, *problema que resuelven*, *aplicabilidad*, *estructura* y *consecuencias*, entre otros.

Los patrones de diseño son clasificados de acuerdo a dos criterios: *propósito* y *alcance*.

El primer criterio, *propósito*, refleja lo que hace el patrón. Los patrones pueden tener diferentes propósitos, éstos se dividen en tres: de *creación*, *estructural* o de *comportamiento*. Los patrones de creación involucran procesos de creación de objetos. Los estructurales manejan la composición de clases y objetos. Los de comportamiento son responsables de la manera en que interactuan y se distribuyen responsabilidades entre las clases y objetos.

El segundo criterio, *alcance*, especifica si el patrón aplica a clases u objetos. Cuando se aplica a clases se refiere a la relación entre las clases y sus subclases, la cual es establecida mediante la herencia. Cuando el alcance aplica a objetos se refiere a las relaciones que hay entre los objetos. Estas relaciones pueden ser cambiadas en tiempo de ejecución.

Los *patrones de creación* con *alcance* a clases transfieren la responsabilidad de creación de objetos a las subclases mientras que con *alcance* a objetos asignan la responsabilidad a otro objeto.

Los *patrones estructurales* con *alcance* a clases utilizan herencia para formar las clases, mientras que con *alcance* a objetos describen la forma de ensamblar objetos.

Los *patrones de comportamiento* con *alcance* a clases usan herencia para describir el algoritmo y el flujo de control, mientras que con *alcance* a objetos describen la cooperación de un grupo de objetos para desempeñar una tarea.

4.2.3. Diseñando con patrones de diseño

El enfoque utilizado para el diseño de software con patrones, en general, es opuesto al enfoque tradicional. La manera de hacerlo con patrones es enfocarse en las *relaciones de alto nivel* entre los elementos que conforman un sistema de software.

El enfoque de Alexander en [Ale79], no es simplemente el de identificar patrones y describirlos, sino de utilizarlos como un nuevo paradigma de diseño.

Para Alexander [Ale79], el diseño es frecuentemente pensado como un proceso de *síntesis*, un proceso de poner todas las partes juntas, un proceso de combinación. El mismo autor dice «*Un buen diseño de software no puede ser alcanzado al simplemente unir partes que han sido desarrolladas independientemente*». Este enfoque, sin duda puede desarrollar software reutilizable, sin embargo, la meta es conseguir software altamente flexible y robusto.

La técnica del diseño de sistemas con patrones es diseñar entidades, objetos y clases, junto con el contexto donde serán utilizadas de modo que se generen sistemas robustos y flexibles. Además, es necesario no perder de vista la perspectiva general del sistema diseñado. El diseño de sistemas se hace a través de un proceso de *análisis* de manera que se enfrenta al problema de la manera más simple, para luego agregar características por medio de *distinciones*, haciendo el proceso de diseño cada vez más complejo.

Las reglas de diseño de arquitecturas con patrones establecidas por Alexander en [Ale79] se presentan a continuación:

- *Un patrón a la vez*, los patrones son aplicados en secuencia.
- *El contexto primero*, aplica aquellos patrones que generen el contexto para los otros.

Sin embargo, y de acuerdo a Shalloway *et al.* [ST04], estas reglas no pueden ser aplicadas directamente al diseño de software ya que fueron creadas para el diseño de arquitecturas en el ámbito de la construcción. Pero pueden ser utilizadas como *principios* para la orientación en la tarea del diseño de software.

4.3. Tipos de software

La flexibilidad necesaria para un diseño depende en gran medida del tipo de software que se esté construyendo. En el libro de Gamma *et al.* [GHJV95] se presentan tres clases principales de software: *programas de aplicación*, *toolkits* y *frameworks*.

En los *programas de aplicación* se da prioridad a la facilidad de mantenimiento, reutilización interna de código y capacidad de extensión. Al utilizar patrones de diseño se logran las características mencionadas de la siguiente manera:

1. Se facilita el mantenimiento al limitar las dependencias a la plataforma de implementación.
2. Se reducen las dependencias entre clases por lo que se incrementa la reutilización interna de código.
3. Se mejora la capacidad de extensión al explotar la composición de objetos. Además de que se reduce el acoplamiento, lo que mejora la capacidad de extensión.

Un *toolkit* es un conjunto de clases reutilizables que han sido diseñadas para proveer funcionalidades de propósito general. Los *toolkits no imponen un diseño particular* para la aplicación que los utiliza, sólo proveen funcionalidades que pueden ayudar a realizar una tarea necesaria para otra aplicación. Los *toolkits* ponen énfasis en la reutilización de código. Al utilizarlos se evita volver a codificar funcionalidades comunes. El diseño de un *toolkit* es mucho más difícil que el diseño de una aplicación ya que éste debe poder ser utilizado en diferentes aplicaciones para que sea útil.

Y por último, un *framework* es un conjunto de clases que cooperan para crear un diseño reutilizable para una clase de software específica. Al utilizar un *framework* como base para el desarrollo de una aplicación particular se generan subclases a partir de las clases abstractas que el *framework* proporciona, por lo tanto, el *framework impone una arquitectura* para la aplicación a desarrollar, por lo que el énfasis en el diseño de un *framework* se encuentra en la reutilización del diseño más que en la reutilización del código.

Cuando se utiliza un *toolkit*, se escribe el programa principal y se reutilizan las funciones ofrecidas por el *toolkit*. Cuando se utiliza un *framework*, se reutiliza el pro-

grama principal ofrecido por el *framework* y se escriben las funciones que se llamarán. Esto permite desarrollar aplicaciones más rápidamente y mantener estructuras similares en las aplicaciones desarrolladas.

El diseño de un *framework* es la tarea más difícil entre las tres mencionadas ya que su diseñador debe asegurar que tal diseño pueda ser utilizado para el desarrollo de aplicaciones para las que fue diseñado. La contribución principal del *framework* a una aplicación es la arquitectura que éste define, por lo que es sumamente importante que un *framework* sea tan flexible y extensible como le sea posible.

Los patrones de diseño ayudan a que la arquitectura de un framework sea apropiada para muchas aplicaciones diferentes sin necesidad de rediseñarlo, además de que el conocimiento de los patrones de diseño ayuda a entender la estructura del *framework* más rápidamente.

A continuación se presentan las diferencias entre un *framework* y los patrones de diseño:

1. Los patrones de diseño se encuentran en un nivel de abstracción superior al de los *frameworks*.
2. La arquitectura definida por un patrón de diseño es más pequeña comparada con la de un *framework*.
3. Los patrones de diseño no son tan especializados como los *frameworks*.

En este capítulo se introdujeron conceptos básicos de patrones de diseño, se presentó la clasificación de Gamma *et al.* [GHJV95] y se expusieron algunas de las ventajas obtenidas al ser aplicados en la tarea de diseño de software.

Entre otras cosas, se presentó la perspectiva de Shalloway *et al.* [ST04] la cual considera a un objeto como una entidad con responsabilidades bien definidas, ésto permite generar un diseño a partir de la identificación de responsabilidades.

Por último, la idea detrás de los patrones es, de acuerdo a Shalloway *et al.* [ST04], identificar las fuerzas, motivaciones y relaciones de un problema particular y proveer de una manera eficiente para su manejo.

Capítulo 5

Arquitectura propuesta

En este capítulo se presenta de manera detallada el proceso de desarrollo de la arquitectura propuesta. Primero se presentan los requerimientos para la arquitectura y se procede a un análisis de las partes que conforman la solución para obtener los subsistemas de la arquitectura. Los subsistemas obtenidos se descomponen en partes para obtener los elementos que los componen y con la ayuda de los Patrones de Diseño se definen las interfaces, servicios y comunicación entre los objetos involucrados en la arquitectura diseñada. Al final del capítulo se presenta la arquitectura Patrón Orientada a Objetos con la identificación de los patrones utilizados en su diseño.

5.1. Desarrollos de software anteriores

El lenguaje más utilizado para la implementación de métodos que aproximen soluciones a EDOs es Fortran debido al periodo en que fueron desarrollados estos métodos, Petcu [Pet05]. La mayoría de estos códigos son de dominio público por lo que son utilizados con frecuencia en el desarrollo de sistemas de software que involucren aproximar soluciones a EDOs.

La implementación de métodos numéricos para resolver este tipo de problemas era reducida a una subrutina, ejemplo de ello son la colección de rutinas de ODE-Pack [Hin83], VODE [PNB89], LSODE [RH93], entre otros. Ejemplos actuales de

este enfoque en la implementación son los códigos de Shampine *et al.* [SR97] para MatLab y Shampine *et al.* [SC00] para Maple. Entre los problemas de este tipo de implementaciones, y de acuerdo al trabajo de Milde [Mil98], nos encontramos que cuando se desea resolver un problema concreto utilizando las rutinas implementadas en Fortran, se debe especificar una lista de parámetros de entrada que en ocasiones puede llegar a tener más de 20 elementos.

Aunque estas implementaciones resultan eficientes en cuanto a desempeño, presentan grandes problemas en manejo y extensibilidad, entendiendo ésta última como la capacidad de agregar nuevas funciones con un mínimo de esfuerzo e impacto en las funcionalidades existentes. Los problemas que presentan estas implementaciones se deben en gran parte a la falta de modularidad, lo que dificulta la reutilización del código. De acuerdo al trabajo de Kees *et al.* [KM99], al implementar algoritmos numéricos en un estilo completamente procedimental, lenguaje Fortran, se oscurece la estructura conceptual de alto nivel tanto del algoritmo numérico como del código.

En lo que respecta a implementaciones de Métodos de Integración Multitasa, en los trabajos de Engstler, [EL97] y Kværnø *et al.* [KR99] se presentan implementaciones, en lenguajes procedimentales, de algoritmos multitasa para resolver un problema específico. Al ser implementados bajo este esquema se dificulta la reutilización y mantenimiento del código.

5.2. Requerimientos del diseño

La tarea de definición de los requerimientos se basa en la obtención de las descripciones del cliente orientadas a lo que el sistema debe ser capaz de hacer. Sommerville explica que una definición de requerimientos debe ser una descripción abstracta tanto de los servicios que el sistema debería proveer como de las restricciones bajo las cuales debería operar, [Som95]. La definición de los requerimientos debe entonces especificar únicamente el comportamiento externo del sistema.

Los requerimientos del sistema pueden ser funcionales o no funcionales, con fun-

cionales nos referimos a los servicios que el diseño debe facilitar y con no funcionales a las restricciones en los servicios o funciones ofrecidas por el sistema, por ejemplo: tiempo de respuesta, lenguaje de programación, plataforma de desarrollo utilizada, uso de estándares, entre otros.

Los requerimientos deben ser completos y consistentes. Por completos se entiende que cubran las necesidades del usuario y por consistentes que no existan conflictos o contradicciones entre ellos.

En este trabajo de tesis se diseña una arquitectura de software, por lo tanto los requerimientos se enfocarán en especificar las facilidades y restricciones de la arquitectura.

5.2.1. Obtención de los requerimientos

La arquitectura propuesta esta dirigida a un público con interés en realizar pruebas numéricas utilizando diferentes combinaciones de métodos bajo distintas estrategias multitasa. A partir del análisis de los trabajos relacionados con el desarrollo de software orientado a objetos para aproximar soluciones a problemas con valores de inicio, entre los que se encuentran [Mil98], [Con95] y [MCH+04], y junto con los que mencionan las características que debe cumplir este tipo de software, como son [Gea82] y [Gea81], se obtienen los siguientes requerimientos.

- Interfaz consistente independiente del problema y del método numérico utilizado para aproximar la solución.
- Interfaz intuitiva para usuarios no experimentados, por ésto se entiende que las entidades que conforman a la arquitectura, así como las tareas que realizan sean claramente identificables por medio de su interfaz.
- Capacidad para incorporar nuevos métodos numéricos.
- Capacidad para incorporar nuevos algoritmos de interpolación.
- Capacidad para agregar nuevos modelos de manera que únicamente sea necesario definir el nuevo modelo para poder simularlo.
- Capacidad para seleccionar un método numérico particular de manera sencilla.

- Capacidad para alterar los parámetros de un método numérico para realizar ajustes finos.
- Capacidad para seleccionar diferentes estrategias de Integración Multitasa.
- Permitir al usuario generar sus propios Métodos Multitasa a partir de los métodos numéricos disponibles.
- Independencia de las estrategias multitasa de los métodos numéricos.
- Software de propósito general y robusto.

De acuerdo a la metodología presentada en el capítulo de Introducción, primero se construye un esquema general a manera de diagrama donde se presentan los posibles componentes del sistema y las relaciones que existen entre ellos. Segundo, la estrategia utilizada para el diseño de la arquitectura es la orientada a objetos donde cada objeto es responsable de sí mismo. La comunicación entre ellos se realiza mediante el llamado a los procedimientos del objeto, mejor conocido como *paso de mensajes*. Con ayuda de esta estrategia, en principio, se obtiene un diseño con alta cohesión y bajo acoplamiento, características deseables en el diseño de sistemas flexibles.

En la siguiente sección se presenta un ejemplo del uso de la arquitectura diseñada implementada bajo C++.

5.2.2. Ejemplo

La arquitectura propuesta es capaz de resolver sistemas de EDO's con valores de inicio que tengan o no diferentes escalas de tiempo. En esta sección se muestra un ejemplo que presenta dos escalas de tiempo. El objetivo de este ejemplo es crear una perspectiva general de la arquitectura presentada en las secciones siguientes y ayudar al lector a identificar las partes que conforman a la arquitectura.

Suponiendo que se desea resolver el siguiente problema con valores de inicio.

$$
\begin{aligned}
y_1'(t) &= 3y_2 - 5y_1 + \sin(1/20t) \\
y_2'(t) &= -25y_2 + \sin(20t) \\
y(0) &= 1 \qquad 0 \leq t \leq 2
\end{aligned}
$$

Donde la ecuación $y_1'(t)$ representa la *parte lenta* y la ecuación $y_2'(t)$ representa la *parte rápida* del sistema. Se desea integrar la parte lenta con un método de baja precisión, por ejemplo un *Euler* con un tamaño de paso $H = 1/16 = 0.0625$. Para la parte rápida se utilizará un método de mayor precisión, en este caso un *Runge-Kutta 4* con tamaño de paso $h = H/2$.

A continuación se presentan los pasos para dar solución a este problema. Como información adicional se muestran las líneas de código en C++ para cada una de las etapas presentadas.

Definición del problema

El primer paso para dar solución a un problema es definirlo como un nuevo elemento dentro de la arquitectura, es decir, crear la *clase* que lo representará. La definición de esta clase se presenta a continuación:

```
// >> Clase CCncrtEjemplo que define al problema presentado, esta clase hereda
// >> de la clase abstracta 'CAbsModelo'
class CCncrtEjemplo : public CAbsModelo {

  CCncrtEjemplo();           // >> Constructor
  virtual ~CCncrtEjemplo();  // >> Destructor

  void setCndInc();    // >> Establece las condiciones de inicio
  void setFunctions(); // >> Genera el sistema de ecuaciones

  // >> Función que representa la ecuación y'_1(t)
  static void f0(const double _tn, const double *_y, double *_dy);
  // >> Función que representa la ecuación y'_2(t)
  static void f1(const double _tn, const double *_y, double *_dy);
};
```

Implementación

Una vez definida la clase, se procede a implementar las funciones definidas por su interfaz. Se especifican las condiciones de inicio, el orden de ejecución de las ecuaciones y las ecuaciones propias del sistema de ecuaciones. La tarea correspondiente en C++ se presenta a continuación.

```
#include "CCncrtEjemplo.h"
// >> El constructor especifica el número de ecuaciones del sistema, éste
// >> se especifica en el llamado al constructor 'CAbsModelo'
CCncrtEjemplo::CCncrtEjemplo() : CAbsModelo(2) {
  cout << "CCncrtEjemplo::CCncrtEjemplo() - OK" << endl;
}
```

```
CCncrtEjemplo::~CCncrtEjemplo() {
  cout << "CCncrtEjemplo::~CCncrtEjemplo() - OK" << endl;
}

// >> Establece las condiciones iniciales o valores de inicio
void CCncrtEjemplo::setCndInc() {
  y[0] = 1.0;
  y[1] = 1.0;
  cout << "CCncrtEjemplo::setCndInc() - OK" << endl;
}

// >> Se especifica el orden de las ecuaciones en el sistema de ecuaciones
void CCncrtEjemplo::setFunctions() {
  f[0] = &f0;
  f[1] = &f1;
  cout << "CCncrtEjemplo::setFunctions() - OK" << endl;
}

// >> Implementación de las ecuaciones particulares del problema
void CCncrtEjemplo::f0(const double _tn, const double *_y, double *_dy) {
  _dy[0] = 3.0*(_y[1]) - 5.0*(_y[0]) + sin(0.05 * _tn);
}

// >> Implementación de las ecuaciones particulares del problema
void CCncrtEjemplo::f1(const double _tn, const double *_y, double *_dy) {
  _dy[1] = - 25.0*(_y[1]) + sin(20.0 * _tn);
}
```

Programa principal

A partir de la definición e implementación del problema se genera el Método Multitasa como una combinación de dos métodos estándares, en este caso corresponde a los métodos *Euler* y *Runge Kutta 4*. Una vez generado el Método Multitasa se da inicio al proceso de aproximación de la solución. Las líneas de código en C++ para esta tarea se presentan a continuación.

```
CAbsModelo *modelo;                  // >> Objeto que representa el problema a resolver

CAbsIntegratorFactory *IntFac;       // >> Se encarga de crear los métodos numéricos
CAbsIntegrator *MultirateIntegrator; // >> Método Multitasa
CAbsIntegrator *SlowIntegrator;      // >> Método numérico para la parte lenta
CAbsIntegrator *FastIntegrator;      // >> Método numérico para la parte rápida

// >> Se crea la instancia del modelo del problema
modelo = new CCncrtEjemplo();

// >> Se prepara la fábrica de métodos
IntFac = = new CCncrtIntegratorFactory(modelo);

// >> Se crean los métodos numéricos estándares
SlowIntegrator = IntFac->create("Simple", "Euler", 0.0625, 0, 0);
FastIntegrator = IntFac->create("Simple", "RungeKutta4", 0.0625 / 2.0, 1, 1);

// >> Se crea el método Multitasa combinando los métodos numéricos estándares
MultirateIntegrator = IntFac->create("MultirateForward", SlowIntegrator, FastIntegrator);

// >> <<<<<<<<<<< Etapa de aproximación a la solución >>>>>>>>>>>> <<
// >> Intervalo de la solución
```

```
double a = 0.0;
double b = 2.0;
double t = a;

modelo->evaluady();
// >> Integración y aproximación de la solución
while (t < b) {
  MultirateIntegrator->integra(t, modelo->y, modelo->dy);
  t+= 0.0625;
}

cout << "Resultado en t = 2.0" << endl;
cout << "y = " << modelo->y[0] << endl;
cout << "x = " << modelo->y[1] << endl;
```

5.3. Diseño de la arquitectura

Un buen diseño es la clave de un proceso de ingeniería efectivo. Sommerville menciona en [Som95], que el proceso de diseño es un proceso creativo que requiere de ingenio e intuición por parte del diseñador, lo cual se obtiene a través del estudio de los sistemas existentes. En general, el proceso de diseño puede ser dividido en tres partes principales:

1. Estudio y entendimiento del problema.
2. Identificación de las características principales de una posible solución.
3. Descripción de las abstracciones usadas en la solución.

El punto número 1 se abordó en los capítulos sobre *Fundamentos*, los puntos 2 y 3 se desarrollarán en este capítulo.

Con base en lo presentado por Sommerville en [Som95], la primer tarea en el proceso de diseño es identificar los subsistemas y establecer una base para el control y comunicación de éstos. A este proceso se le conoce como el *diseño de la arquitectura*. Durante este proceso hay que tener en cuenta que el modelo de un sistema es una *abstracción del sistema* estudiado más que una *alternativa para su representación*.

Como parte del diseño de la arquitectura es recomendable cubrir las siguientes actividades:

1. *Estructuración del sistema*, estructurar el sistema como un conjunto de subsistemas. En esta etapa se identifican las comunicaciones entre los subsistemas.

2. *Modelado del control*, se establece un modelo de control entre las partes del sistema.

3. *Descomposición modular*, cada subsistema se descompone en módulos.

Al final del proceso de diseño se obtiene una serie de diagramas que muestran la descomposición en subsistemas junto con los módulos que los conforman. De acuerdo a Sommerville [Som95], se entiende por *subsistema* a una entidad del sistema que *no depende* de las operaciones o servicios de otros subsistemas. Un *módulo* es un componente de un sistema que *provee uno o más servicios* a otros módulos.

Es importante hacer notar que el diseño obtenido influirá en el desempeño, mantenimiento, robustez y portabilidad del sistema.

5.3.1. Estructuración del sistema

La primera etapa en la tarea de diseño de la arquitectura es la descomposición del sistema en subsistemas. En los diagramas presentados se utiliza una flecha para indicar transferencia de datos o de control de un subsistema a otro.

Con base en el problema a resolver se identifican al menos dos partes principales:

1. Un *problema con valores de inicio* a resolver.

2. Un *método integrador* para obtener una aproximación a la solución.

En la figura 5.1 se presentan estas dos partes a manera de subsistemas.

Debido a que el diseño corresponde a una arquitectura multitasa, el método integrador será un Método Multitasa. Por cuestiones de claridad nos referimos a un Método Multitasa como *método integrador* y a los métodos numéricos estándares, tales como el método de Euler o métodos Runge Kutta, como *método numérico*.

Este diseño parcial refleja que el problema con valores de inicio, en adelante llamado *modelo del problema*, envía información al *método integrador* el cual se encarga de calcular una aproximación de la solución del problema. Los resultados son enviados y almacenados en el subsistema *modelo del problema*.

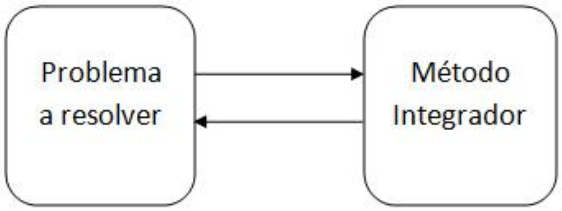


Figura 5.1: Primera arquitectura Multitasa.

El subsistema *modelo del problema* actúa como *elemento de entrada* ya que proporciona el problema a resolver al subsistema *método integrador*.

De acuerdo a Buschmann [BMR+96], el patrón de arquitectura *Modelo-Vista-Controlador* o *MVC*, por sus siglas en inglés, identifica tres elementos en un diseño de software.

- *Modelo*, funcionalidad principal del sistema y manejo de datos. Encargado del *procesamiento* de datos.
- *Vista*, encargado de mostrar la información al usuario, *salidas* del sistema.
- *Controlador*, encargado de manejar los datos de entrada, *entradas* del usuario.

Tanto el *controlador* como la *vista* se refieren a factores externos al sistema de procesamiento.

Al aplicar este patrón, podemos identificar las partes del diseño obtenido con los elementos que conforman el patrón *MVC*, por ejemplo, el *modelo del problema* se identifica con el elemento *controlador* del patrón *MVC* ya que éste proporciona los datos de entrada al *método integrador*.

Al aplicar el principio *Model-View separation*, presentado por Larma en [Lar01] el cual menciona que el modelo no debería tener conocimiento de como se presentan sus datos, generamos un nuevo subsistema encargado del manejo y presentación de los datos de salida. La nueva arquitectura se presenta en la figura 5.2.

La arquitectura obtenida se ha presentado como la comunicación y organización de los subsistemas que la conforman. A partir de este punto se trabajará únicamente con el subsistema *método integrador* ya que los demás no pueden ser descompuestos

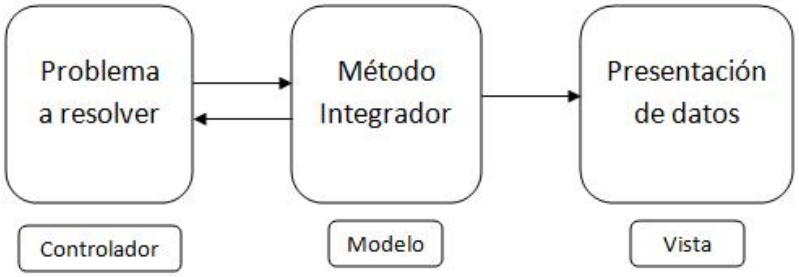


Figura 5.2: Arquitectura Multitasa con el patrón MVC.

en más subsistemas. En la etapa de diseño orientado a objetos se especificarán los elementos que conforman a cada subsistema.

Estrategias Multitasa

Como se mencionó en el capítulo 3, existen tres estrategias de integración Multitasa, a saber, *interpolación*, *información adelantada* e *información atrasada*. El subsistema *método de integración* necesita contemplar este hecho por lo que se genera un subsistema encargado de administrar y aplicar estas técnicas.

Un algoritmo multitasa se conforma de al menos dos métodos numéricos estándares por lo que será necesario un subsistema encargado del manejo de estos métodos. En la figura 5.3 se presentan los subsistemas que conforman al subsistema *método integrador*.

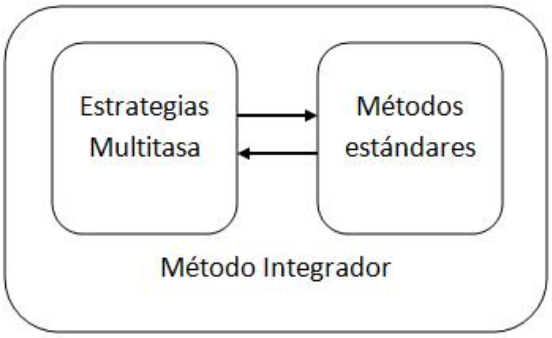


Figura 5.3: Subsistemas que conforman al subsistema *método integrador*.

Organización de los subsistemas

Con base en lo presentado por Sommerville en [Som95], el tipo de estructura utiliza por esta arquitectura es el *modelo cliente-servidor*. Esta estructura se caracteriza porque los datos son enviados de un subsistema a otro, en este caso los datos enviados son los valores calculados para la aproximación de la solución, esta información es necesaria para que los métodos numéricos puedan realizar su tarea. La estrategia multitasa elegida se encarga de manipular y transmitir esta información.

Entre las ventajas obtenidas al utilizar la estructura *modelo cliente-servidor*, se encuentra la de poder distribuir los subsistemas en diferentes servidores. Cada subsistema podría ejecutarse en un servidor a la espera de solicitudes o datos de entrada para realizar sus cálculos. Este enfoque permitiría realizar operaciones en forma paralela por lo que se obtendrían resultados más rápido.

Componentes de la arquitectura

El nivel de detalle alcanzado hasta el momento permite identificar los componentes funcionales de la arquitectura. Sommerville distingue entre *componentes funcionales* y *subsistemas* de la siguiente manera. Los *componentes funcionales* son aquellos que proveen una sola función, es decir, se encargan de realizar una sola tarea. Los *subsistemas* son multifuncionales, es decir, realizan varias tareas. Los componentes funcionales son parte de los subsistemas y se encargan de definir la clasificación de estos últimos.

- *Componentes tipo sensor*, coleccionan información del ambiente del sistema.
- *Componentes tipo actuador*, causan algún cambio en el ambiente del sistema.
- *Componentes de cómputo*, dada una entrada realizan un cálculo con ella y producen una salida.
- *Componentes de comunicación*, permiten a un sistema comunicarse con otro.
- *Componentes de coordinación*, se encargan de coordinar la comunicación entre componentes.
- *Componentes de interfaz*, transforman la representación utilizada por un componente en la utilizada por otro componente.

Nombre del subsistema	Tipo de componentes
Problema a resolver	Cómputo
Estrategias multitasa	Coordinación
Métodos estándares	Cómputo
Presentación de datos	Interfaz

Tabla 5.1: Tipo de los subsistemas de acuerdo a los componentes que los conforman.

Considerando los diferentes tipos de componentes descritos y las responsabilidades de los subsistemas hasta ahora generados, se obtiene la clasificación mostrada en la tabla 5.1. Los subsistemas que conforman a la arquitectura se presentan en la figura 5.4.

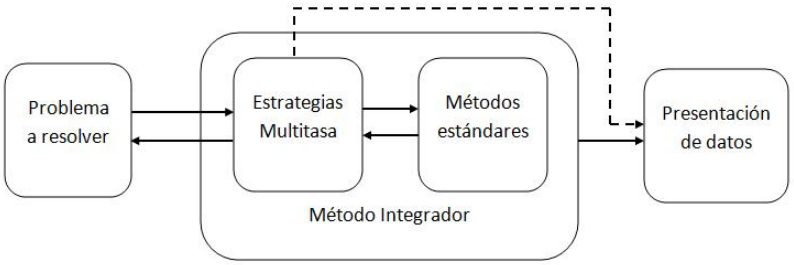


Figura 5.4: Vista de los subsistemas que conforman a la arquitectura.

5.3.2. Modelado del control

Con base en lo presentado por Sommerville en [Som95], el *modelo de control* que más se adecua a la arquitectura es el *centralizado*. En este modelo existe un subsistema encargado de controlar el inicio y fin de las operaciones de los demás subsistemas, este subsistema es *Método Integrador* ya que se encarga tanto de la sincronización de los métodos numéricos estándares como del envío de datos a los demás subsistemas.

El *modelo de control* es a su vez un modelo tipo *call-return* ya que la ejecución de los subsistemas se realiza de manera secuencial. Cuando se solicita la ejecución de un subsistema se le pasa el control para que ejecute sus operaciones y utilice los recursos

necesarios. Al término de sus operaciones debe regresar el control al subsistema que lo llamó.

Entre las ventajas del modelo *call-return* se encuentra la de facilitar el análisis del flujo de control, además de poder predecir el comportamiento del sistema a ciertas entradas.

5.3.3. Descomposición modular

La descomposición de los subsistemas para la generación de módulos se realiza a través de la generación de objetos. En un diseño orientado a objetos y bajo la perspectiva presentada por Shalloway en [ST04], un módulo es un conjunto de objetos que son responsables de su estado.

Como ya se mencionó en el capítulo anterior, al contar con un diseño orientado a objetos se logra, en principio, bajo acoplamiento entre los módulos debido al ocultamiento de información. Esta característica es básica en el diseño de software flexible ya que permite modificar e intercambiar objetos sin afectar a otros, ésto se debe a que no hay dependencias de datos entre los objetos modificados.

Entre las características de un modelo orientado a objetos se encuentran las siguientes:

- Los objetos representan abstracciones del mundo real o entidades de sistemas que son responsables de administrar su estado. Además, ofrecen servicios a otros objetos.
- Los objetos son entidades independientes, estos pueden ser cambiados por otros con la misma interfaz sin problemas ya que la implementación queda oculta a los usuarios.
- La funcionalidad del sistema queda expresada en términos de las operaciones y/o servicios que ofrecen los objetos, es decir, sus responsabilidades.
- No es necesario el uso de una base de datos compartida ya que cada objeto es responsable de sus datos. Ésto reduce el acoplamiento del sistema.
- Al contar con un diseño orientado a objetos el sistema puede ser desarrollado de manera secuencial o paralela.

En la siguiente sección se presenta de manera detallada la descomposición en objetos de los subsistemas que conforman a la arquitectura.

5.4. Diseño Orientado a Objetos

El modelado de sistemas mediante objetos se utiliza para representar a los datos del sistema junto con los procesos que los manejan, esta representación es útil para mostrar que las entidades del sistema se componen de otras. Un modelo de objetos describe las entidades lógicas del sistema, su clasificación y agregación. El modelo de objetos combina un modelo de datos con un modelo de procesamiento.

La técnica utilizada para generar el diseño orientado a objetos es aplicar los patrones de diseño de acuerdo a los requerimientos de la arquitectura.

En el trabajo de Sorana *et al.* [CC08], se menciona que la actividad de reutilizar un patrón, también llamada *Ingeniería de Software basada en Patrones* involucra tres actividades:

1. Selección de los patrones relevantes.
2. Adaptación al contexto del usuario.
3. Generación de código (opcional).

En las secciones siguientes se presenta la generación de objetos a partir de los subsistemas que conforman a la arquitectura. La metodología utilizada para la selección de patrones es la propuesta por Gamma *et al.* en [GHJV95], la cual se presentó en el capítulo de introducción de este trabajo.

5.4.1. Subsistema: Problema a resolver

Este subsistema se encarga de modelar el problema a resolver. Todo problema con valores de inicio se conforma de al menos las siguientes partes:

- Un sistema de ecuaciones.

- Condiciones de inicio.

A partir de esta información podemos generar una clase para modelar un problema con valores de inicio. Suponiendo que se trata de un problema con un sistema de EDO's con dos ecuaciones, la clase generada se presenta en la figura 5.5.

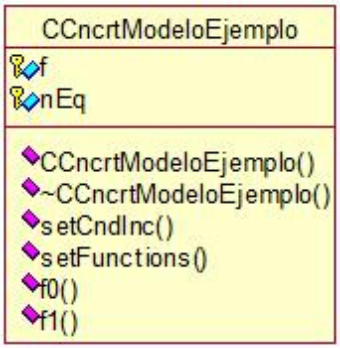


Figura 5.5: Clase para la representación de un problema con valores de inicio.

Esta clase representa al sistema de ecuaciones, dos para este caso concreto con las funciones **f0** y **f1**. La función **setCndInc** se encarga de establecer las condiciones de inicio del problema. La función **setFunctions** establece el orden de las ecuaciones en el sistema de ecuaciones. La variable **nEq** almacena el número de ecuaciones que conforman al sistema.

A partir de esta clase se obtiene una generalización para representar problemas con valores de inicio por lo que se crea una clase abstracta. A las clases abstractas se les identificará por el prefijo **CAbs**. No se pueden crear instancias de este tipo de clases.

La clase abstracta generada se presenta en la figura 5.6. Es importante notar que se agregaron dos elementos, el objeto **jacobiano** que define un algoritmo para calcular el Jacobiano del sistema de ecuaciones y la función **calculaJacobiano** la cual realiza la operación que su nombre describe.

La clase para la representación del Jacobiano se presenta en la figura 5.7.

La función **algorithm** se encarga de ejecutar el algoritmo definido para calcular el Jacobiano del sistema de ecuaciones.

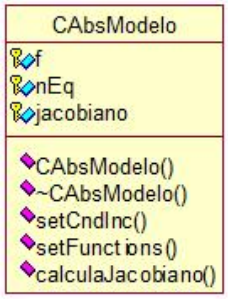


Figura 5.6: Clase abstracta que representa a un problema con valores de inicio.

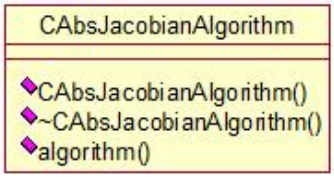


Figura 5.7: Clase abstracta para la representación del Jacobiano.

En el ejemplo al inicio de este capítulo, la clase que representaba al problema, CCncrtEjemplo *hereda* de la clase **CAbsModelo**. Las clases concretas se identifican por el prefijo **CCncrt**. De este tipo de clases es de las que se pueden crear instancias.

Debido a que cada problema con valores de inicio debe establecer sus condiciones de inicio y el orden de evaluación para las ecuaciones que conforman al sistema, es posible identificar un patrón que se encarga de definir una serie de pasos para la definición de un algoritmo. Además, este patrón deja libre la implementación de cada paso del algoritmo para cumplir con las necesidades del problema específico a definir.

El patrón de diseño adecuado para la situación es el *Template method*. De acuerdo a la definición presentada por Gamma *et al.* en [GHJV95], el patrón *Template method* define el esqueleto de un algoritmo, es decir, las operaciones que debe ejecutar, pero deja las implementaciones de éstas a las subclases por lo que es posible contar con diferentes implementaciones para las mismas operaciones. El patrón *Template method* factoriza el comportamiento común de un conjunto de objetos.

De acuerdo a Gamma *et al.* [GHJV95], el uso de este patrón es fundamental para la reutilización del código. La figura 5.8 presenta el resultado de aplicar este patrón.

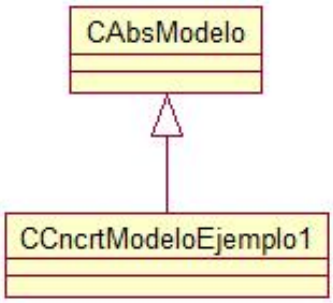


Figura 5.8: Patrón *Template method* para los problemas con valores de inicio.

5.4.2. Subsistema: Presentación de datos

Este subsistema será el encargado de presentar los resultados obtenidos al usuario ya sea a manera de gráficas, listas de datos numéricos, etc. Debe de ser independiente de la interfaz utilizada.

Uno de los patrones más utilizados para la presentación de resultados es el patrón de diseño *Observer*, este patrón define dependencias entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente, Gamma *et al.* [GHJV95].

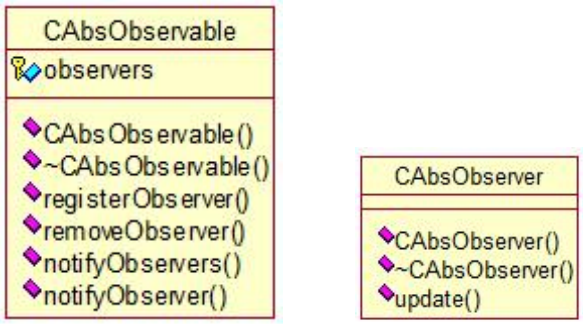
El punto clave en este patrón es definir una lista de *objetos observadores* y un *objeto observado*. Todos los *objetos observadores* serán notificados cuando el *objeto observado* cambie de estado.

Este efecto se logra de la siguiente manera:

1. Un *objeto observador* se suscribe a un *objeto observado*.
2. El *objeto observado* lleva un registro de los objetos que lo observan.
3. Cuando ocurre un cambio en el *objeto observado*, éste notifica a todos los *objetos observador* del cambio presentado.
4. (Opcional) Si los *objetos observador* no desean recibir más notificaciones de los cambios del *objeto observado*, pueden solicitar la eliminación de su suscripción a través de una petición al *objeto observado*.

El patrón *Observer* se utiliza en la arquitectura para obtener los resultados generados por el método multitasa.

De acuerdo a Gamma *et al.* [GHJV95], para utilizar el patrón *Observer* es necesario definir dos clases abstractas, una para los *objetos observador* y otra para los *objetos observados*. En la figura 5.9 se presentan las clases abstractas tanto para los *objetos observados* como para los *objetos observador*.



(a) Clase observable (b) Clase observer

Figura 5.9: Clases abstractas para los *objetos observado* y los *objetos observador*.

En la clase **CAbsObservable** la función **registerObserver** se encarga de registrar o suscribir a los objetos interesados en conocer el estado del objeto. La función **removeObserver** se encarga de remover del registro al *objeto observador* que haya solicitado su eliminación. Las funciones **notifyObservers** y **notifyObserver** se encargan de informar sobre el estado del *objeto observado* a los *objetos observador* interesados en él.

Al aplicar este patrón de diseño se logra un bajo acoplamiento entre los *objetos observado* y los *objetos observador*. Además, se logra un gran número de posibilidades para presentar los resultados ya que sólo es necesario suscribirse al *objeto observado* de interés.

Para el caso particular de la arquitectura diseñada, los *objetos observado* serán los algoritmos multitasa, estos notificarán sobre los resultados obtenidos a los *objetos observador*, los cuales pueden graficar los datos, imprimirlos o ser utilizados como entrada para otro dispositivo. El subsistema *presentación de datos* queda entonces definido como se muestra en la figura 5.10.

Figura 5.10: Comunicación entre las clases del patrón *Observer*.

5.4.3. Subsistema: Método Integrador

Este subsistema se encarga de utilizar una estrategia multitasa para calcular la aproximación a la solución de un problema con valores de inicio. La estrategia multitasa combina al menos dos algoritmos estándares para realizar su tarea.

En secciones anteriores se presentó la descomposición de este subsistema en los subsistemas: 1) *Métodos Multitasa* y 2) *Métodos estándares*. A continuación se expone la generación de objetos a partir de estos subsistemas.

Subsistema: Métodos estándares

La parte principal de todo sistema dedicado a aproximar soluciones a problemas con valores de inicio son los métodos numéricos. Entre los requerimientos del diseño se pide que los métodos numéricos sean independientes al problema que resuelven, es decir, que sea posible aplicar cualquier método numérico para resolver un problema dado.

Entre los patrones de diseño más conocidos se encuentra el patrón *Strategy*, el cual, de acuerdo a Gamma *et al.* [GHJV95], define una familia de algoritmos, los encapsula y permite que sean intercambiables. Esto permite variar algoritmos independientemente del cliente que los utilice.

El patrón de diseño *Strategy* es la opción seleccionada para representar al conjunto de métodos numéricos estándares. Las clases participantes en este patrón son: una *estrategia abstracta*, un conjunto de *estrategias concretas* y una clase *contexto* encargada de seleccionar un algoritmo concreto.

Para aplicar este patrón es necesario identificar las partes comunes en los métodos numéricos para crear la clase correspondiente a la *estrategia abstracta* del patrón

Strategy. Las características encontradas se presentan a continuación:

- Un modelo dado como un sistema de EDO's para solucionar.
- Un paso de integración.
- Tiempo actual.
- Valores previos de la función aproximada.

A partir de este conocimiento, se genera la clase abstracta **CAbsMethods** que actúa como la *estrategia abstracta*. Esta clase se presenta en la figura 5.11.

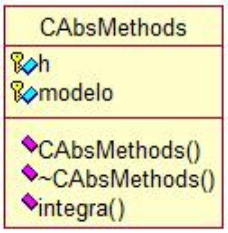


Figura 5.11: Clase abstracta para representar a los métodos estándares.

La función **integra** es la encargada de ejecutar el método numérico seleccionado. La variable **h** representa el tamaño del paso de integración. La variable **modelo**, de tipo *CAbsModelo*, es una instancia del problema con valores de inicio a resolver.

El patrón *Strategy* recomienda definir las clases que representan a las *estrategias concretas*. En nuestro caso se definen dos nuevas clases abstractas, éstas representarán a los dos tipos de métodos numéricos: *Métodos Explícitos* y *Métodos Implícitos*. En la figura 5.12 se presentan estas clases.

La clase **CAbsImplicitMethods** hace uso de la función **algorithm** que calcula la matriz Jacobiana del sistema de ecuaciones. Este cálculo es transparente al usuario

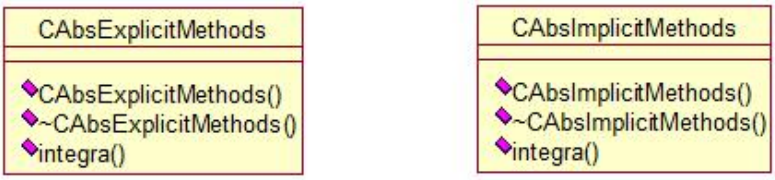


Figura 5.12: Clases abstractas para los métodos numéricos explícitos e implícitos.

ya que la implementación de los *métodos implícitos* se encuentra encapsulada. Por cuestiones de simplicidad no se presentan todos los métodos y variables de la clase **CAbsImplicitMethods**.

Es importante notar que la clase abstracta **CAbsExplicitMethods** puede ser eliminada ya que no define nuevos datos ni operaciones, sin embargo, se deja en el diseño para distinguir entre los tipos de métodos utilizados.

Con la distinción entre los diferentes tipos de métodos podemos definir las *estrategias concretas* o lo que en este caso son métodos numéricos estándares. Para ejemplificar, la figura 5.13 muestra el árbol de herencia para definir los métodos Euler y Backward Euler.

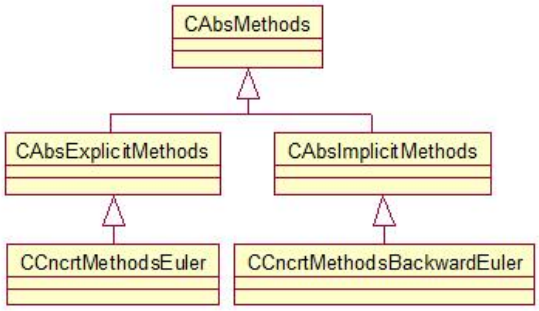


Figura 5.13: Métodos numéricos derivados de la clase abstracta del patrón *Strategy*.

Ya contamos con los elementos *estrategia abstracta* y *estrategias concretas* presentes en la definición del patrón *Strategy*. Falta definir la clase contexto, en este caso se trata de una clase que encapsula al método numérico utilizado, esta clase es **CCncrtIntegratorSimple** la cual forma parte del subsistema *Métodos Multitasa*. La definición de esta clase se presenta más adelante.

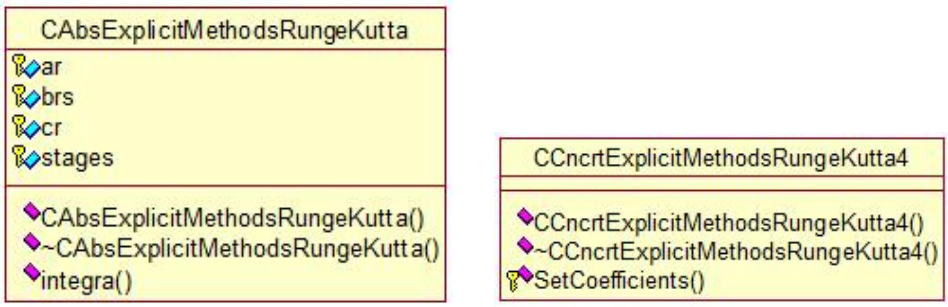
Agregando métodos numéricos

Para proporcionar un conjunto de métodos numéricos significativos en el diseño de la arquitectura, se optó por integrar métodos numéricos de la familia Runge Kutta. En el capítulo 3 se presentó esta familia de métodos y se definió el algoritmo general de los mismos. La generación de un método Runge Kutta particular se obtiene a

partir de la definición del número de etapas y coeficientes a utilizar.

Existe un patrón de diseño encargado de manejar un gran número de objetos que comparten características en común. Este patrón abstrae la parte común de los objetos y la encapsula en una clase abstracta. Cuando se desea obtener un objeto específico, se utiliza un método que configura al objeto abstracto para que se comporte como un objeto específico. El patrón al que nos referimos es el *Flyweight* definido por Gamma *et al.* en [GHJV95].

Para utilizar este patrón necesitamos objetos tipo *Flyweight abstractos* y *Flyweight concretos*, éstos últimos se encargaran de configurar al objeto *Flyweight abstracto*. Los objetos *Flyweight abstractos* serán identificados con una clase abstracta que representará a los métodos Runge Kutta. Los objetos *Flyweight concretos* serán responsables de establecer los coeficientes para generar un Runge Kutta particular. En la figura 5.14 se presentan la clase abstracta, que representa a los objetos *Flyweight abstractos*, y una clase concreta que ejemplifica a los objetos *Flyweight concretos*. La función **integra** de la clase abstracta se encarga de calcular la aproximación



(a) Clase *flyweight* abstracta (b) Clase *flyweight* concreta

Figura 5.14: Clases utilizadas para la representación de los métodos Runge Kutta.

a la solución del problema. Las variables **ar**, **brs** y **cr** se utilizan para almacenar los valores de los coeficientes del Runge Kutta generado. La variable **stages** almacena el número de estados del método Runge Kutta específico.

En la clase concreta, la función **SetCoefficients** se encarga de establecer los coeficientes, y por lo tanto, configurar al objeto *Flyweight abstracto* para obtener un método Runge Kutta concreto.

Al aplicar el patrón *Flyweight* se elimina código repetido y se optimiza espacio. Entre las desventajas de utilizar este patrón se encuentra el tiempo utilizado para configurar al objeto *Flyweight abstracto*.

En la figura 5.15 se presenta la estructura completa para los métodos numéricos estándares. Además, se muestra la participación del patrón *Strategy* y *Flyweight* los cuales ayudan a seleccionar y definir, respectivamente, un conjunto de algoritmos numéricos.

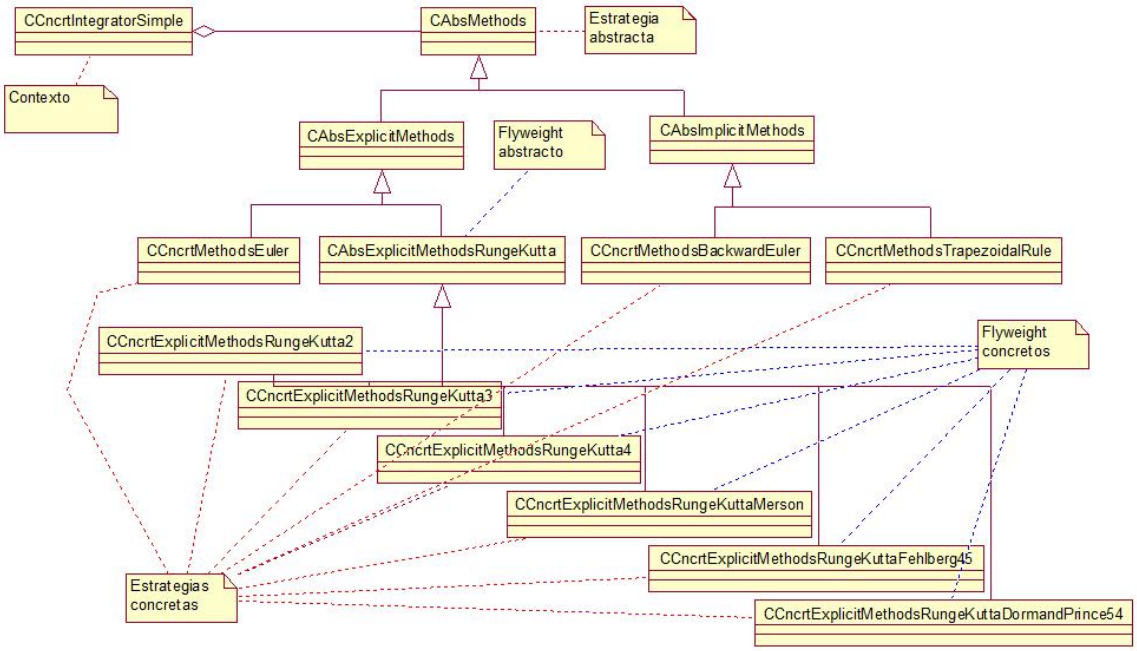


Figura 5.15: Arquitectura para los métodos numéricos estándares y participación de los patrones *Strategy* y *Flyweight*.

Hasta este punto, pareciera que el número de objetos ha crecido y por lo tanto la tarea de controlarlos será complicada. Sin embargo, lo único que se ha hecho es agregar métodos numéricos para ofrecer un conjunto interesante de métodos numéricos al usuario final.

No obstante, es necesario contar con una buena estrategia para administrar los métodos, sobre todo la selección e instanciación de éstos. La arquitectura debe permitir crear un método numérico de manera sencilla. De acuerdo a Gamma *et al.* [GHJV95], especificar el nombre de una clase cuando se crea un objeto genera una

implementación particular en lugar de una interfaz particular. Es necesario evitar la creación de implementaciones particulares si se desea crear un diseño reutilizable.

El patrón *Factory Method* define una interfaz para crear objetos, dejando a las subclases la decisión de que clase instanciar, además, este patrón resulta de gran utilidad para la arquitectura ya que de antemano no se sabe que método numérico se instanciará sino hasta el momento de su ejecución.

En el patrón *Factory Method* intervienen los siguientes elementos: un *producto abstracto*, un *producto concreto*, un *creador abstracto* y un *creador concreto*. La clase *creador abstracto* deja la implementación del método creador a la subclase *creador concreto* de modo que proporcione una instancia del *producto concreto*.

Al aplicar este patrón a la arquitectura nos damos cuenta que ya contamos con los objetos *productos abstractos* y *productos concretos*, en este caso nos referimos a las abstracciones de los métodos estándares y a sus implementaciones concretas. En la figura 5.16 se presentan las clases que representan tanto al *creador abstracto* como al *creador concreto*.

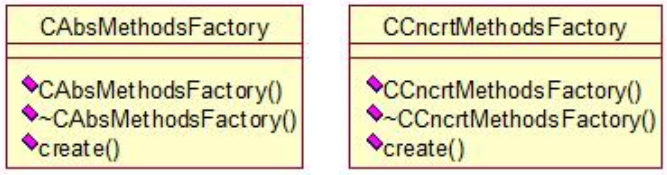


Figura 5.16: Clases para los objetos *creador abstracto* y *creador concreto*.

La función **create** será la encargada de instanciar los tipos de objetos solicitados. En la figura 5.17 se presenta la intervención del patrón *Factory Method* para la generación de las instancias de los métodos numéricos, en este caso se ejemplifica con la creación del método de *Euler*.

Subsistema: Métodos Multitasa

Este subsistema se encarga de la coordinación y sincronización de los datos generados por los métodos numéricos estándares. Además, en este subsistema se imple-

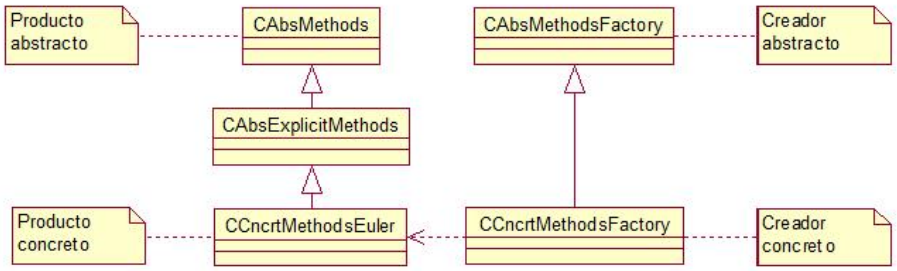


Figura 5.17: Uso del patrón *Factory Method* para crear métodos numéricos.

mentan las diferentes estrategias multitasa para resolver problemas con valores de inicio.

A través del análisis para identificar las partes comunes en las tres estrategias multitasa presentadas en el capítulo 3, se logra definir una clase abstracta, la cual se presenta en la figura 5.18.

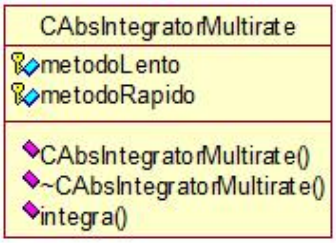
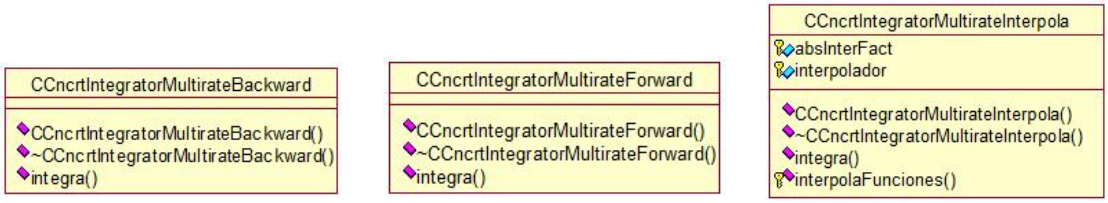


Figura 5.18: Clase abstracta que encapsula las partes comunes de los tres algoritmos multitasa.

La función **integra** es la encargada de ejecutar la técnica multitasa para aproximar a la función de un problema con valores de inicio. Las variables **metodoLento** y **metodoRapido** son instancias de métodos numéricos estándares **CAbsMethods** utilizados por las estrategias multitasa.

A partir de la clase abstracta de las estrategias multitasa se generan las clases que representan a las tres estrategias de integración multitasa. En la figura 5.19 se presentan estas clases.

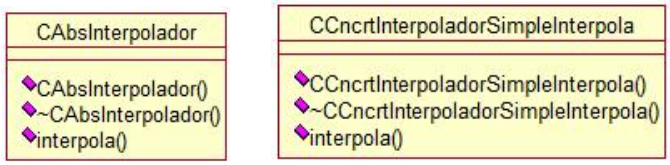
Para la clase **CCncrtIntegratorMultirateInterpola**, la variable **interpolador** es una instancia de la clase **CAbsInterpolador**. Este objeto es el encargado de



(a) Información atrasada (b) Información adelantada (c) Información interpolada

Figura 5.19: Clases que representan los tres algoritmos para Métodos Multitasa.

obtener los valores interpolados de las ecuaciones involucradas. En la figura 5.20 se presenta la clase abstracta para los interpoladores y una clase concreta que implementa un interpolador lineal.



(a) Clase abstracta (b) Clase concreta

Figura 5.20: Clases para representar a los algoritmos interpoladores.

La técnica para selección de algoritmos interpoladores es la misma que se utiliza para la selección de métodos numéricos. Es decir, se hace uso del patrón *Strategy* para seleccionar el algoritmo interpolador y con ayuda del patrón *Factory Method* se crea una instancia del algoritmo. En la figura 5.21 se presenta la participación de estos patrones para la selección y creación de los algoritmos interpoladores.

La arquitectura creada hasta el momento permite seleccionar un método multitasa para resolver un problema con valores de inicio con dos escalas de tiempo. Esto se logra al seleccionar una estrategia multitasa y definir un par de métodos numéricos estándares para generar un Método de Integración Multitasa particular. Sin embargo, es deseable contar con métodos multitasa que permitan definir combinaciones de más de dos métodos numéricos estándares en caso de que el problema presente más de dos escalas de tiempo.

Una manera de dar solución a este problema es tener un arreglo estático con instancias de métodos numéricos estándares dentro de la clase **CAbsIntegrator-**

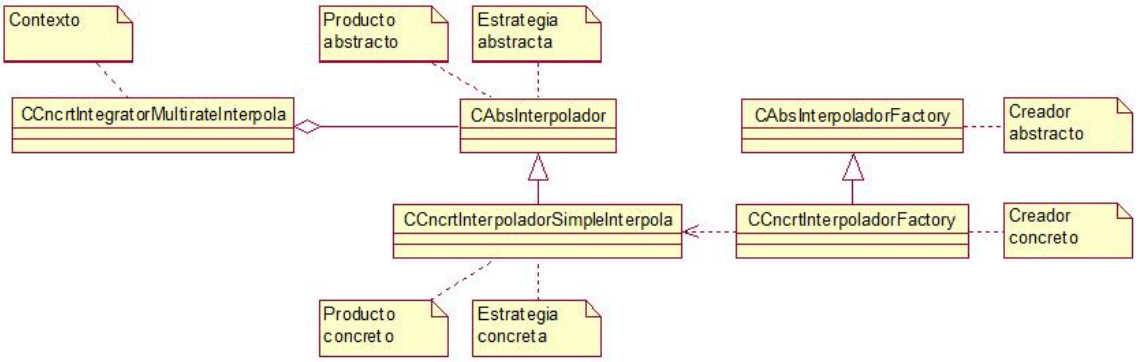


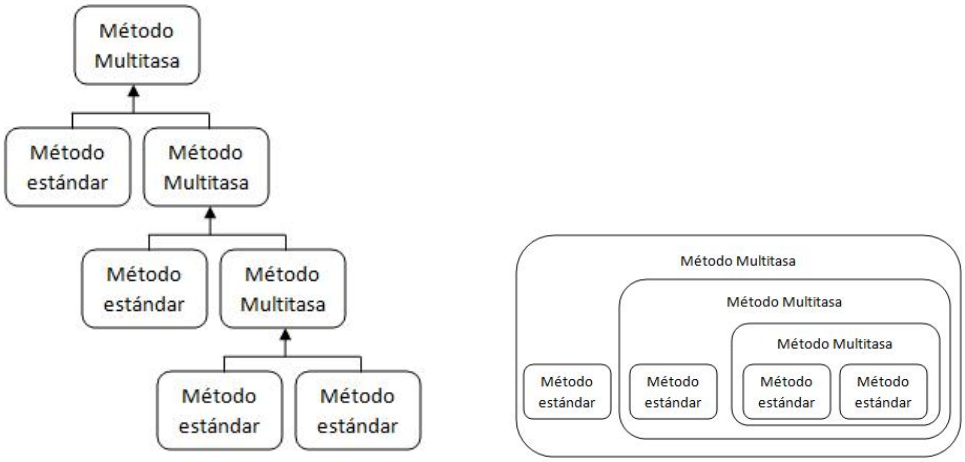
Figura 5.21: Participación de los patrones *Strategy* y *Factory Methods* para la selección y creación del algoritmo interpolador.

Multirate. El inconveniente de esta técnica es que no sabemos con anticipación el número de métodos numéricos que se necesitaran. Al utilizar un arreglo dinámico para cubrir este inconveniente se complicaría el manejo del mismo lo que afectaría el manejo de las estrategias multitasa.

Una alternativa para la implementación de esta característica es la de definir a un método multitasa con tan sólo dos métodos, el lento y el rápido, donde el método lento será una instancia de un método estándar y el método rápido que puede ser una instancia de un método estándar o una instancia de un método multitasa si el problema presenta mas de dos escalas de tiempo. Con este enfoque se consigue el efecto de varios métodos estándares cuando se presentan varias escalas de tiempo. En la figura 5.22 se muestra esta idea a manera de árbol y de anidamiento suponiendo que se presentan cuatro escalas de tiempo por lo que se necesitan cuatro métodos estándares.

El problema de utilizar esta idea es ¿cómo identificar un elemento compuesto de uno simple?, es decir, ¿cómo distinguir entre un método estándar y una estrategia multitasa?

El patrón de diseño *Composite* resuelve este problema. De acuerdo a Gamma *et al.* [GHJV95], este patrón compone objetos en una estructura tipo árbol para representar parte de una jerarquía grande. El patrón *Composite* permite tratar objetos individuales u objetos compuestos *de la misma manera*. En nuestro caso se trata de



(a) En forma de árbol (b) Estructura interna

Figura 5.22: Representaciones de la estructura de un método multitasa con cuatro métodos estándares.

los objetos tipo método estándar y estrategia multitasa.

Para utilizar este patrón necesitamos objetos tipo *componente*, *hoja* y *compuesto*. Comenzamos por definir el objeto tipo *componente* que define la interfaz de los objetos en la composición. En la figura 5.23 se presenta la clase para este tipo de objetos.

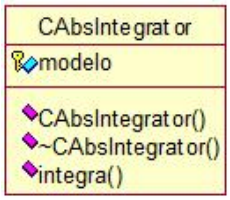


Figura 5.23: Clase para los objetos tipo *componente* del patrón *Composite*.

La función **integra** será la encargada de aproximar a la solución del problema con valores de inicio. La variable **modelo** es una instancia de la clase **CAbsModelo** que representa al problema con valores de inicio.

La clase para los objetos tipo *compuesto* es **CAbsIntegratorMultirate**, la cual se definió en la figura 5.18. Es necesario definir una nueva clase para representar a los objetos tipo *hoja* del patrón *Composite*. La clase para representar a estos objetos

se presenta en la figura 5.24. La variable **absMtdsFact** es una instancia de la clase

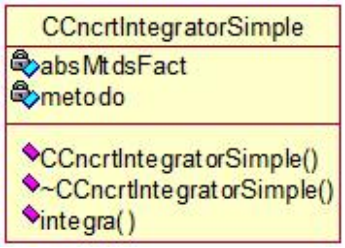


Figura 5.24: Clase para los objetos tipo *hoja* del patrón *Composite*.

CAbsMethodsFactory encargada de la creación de los métodos estándares. La variable **metodo** es una instancia de la clase **CAbsMethods** que representa los métodos numéricos estándares.

En la figura 5.25 se presenta la participación del patrón *Composite*.

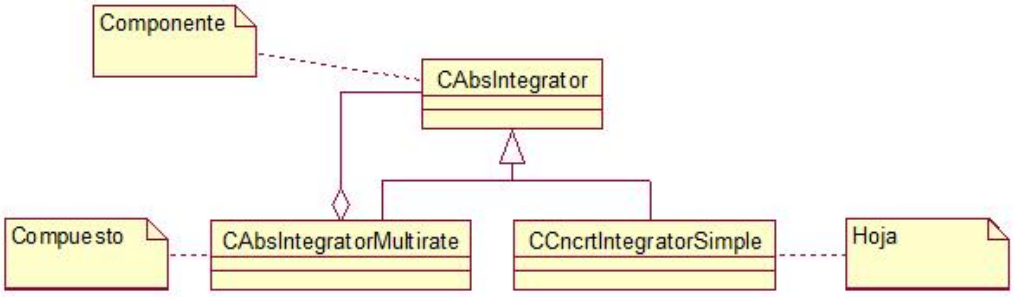


Figura 5.25: Participación del patrón *Composite*.

Recordemos que la clase encargada de calcular las aproximaciones al problema con valores de inicio es una clase de tipo *Observable* ya que los datos que genera serán utilizados por los *Observadores* para ser presentados al usuario. Por lo tanto, la clase **CAbsIntegrator**, definida en la figura 5.23 *hereda* de la clase **CAbsObservable** de la figura 5.9(a).

Para crear a los objetos tipo **CAbsIntegrator** se utiliza nuevamente el patrón **Factory Method**. La participación de este patrón se presenta en la figura 5.26.

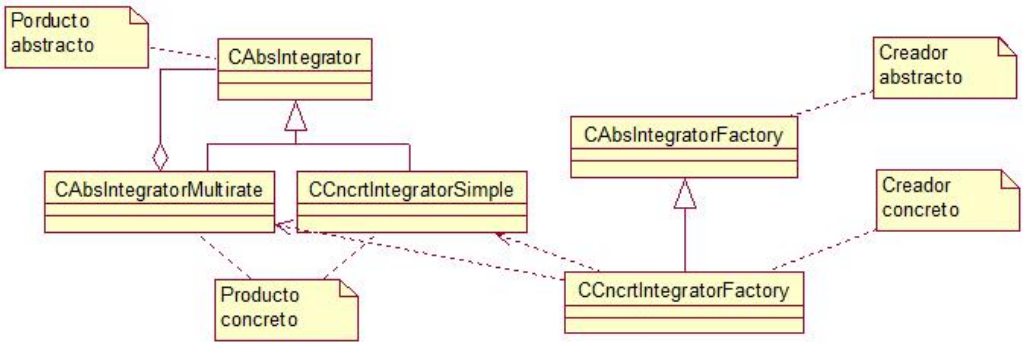


Figura 5.26: Participación del patrón *Factory Method* para la creación de las instancias de objetos tipo **CAbsIntegrator**.

5.5. Arquitectura final

Los componentes de los subsistemas se obtuvieron al utilizar patrones de diseño para resolver los problemas de diseño encontrados. En la figura 5.27 se presenta la interacción entre estos componentes y se identifican los patrones de diseño utilizados.

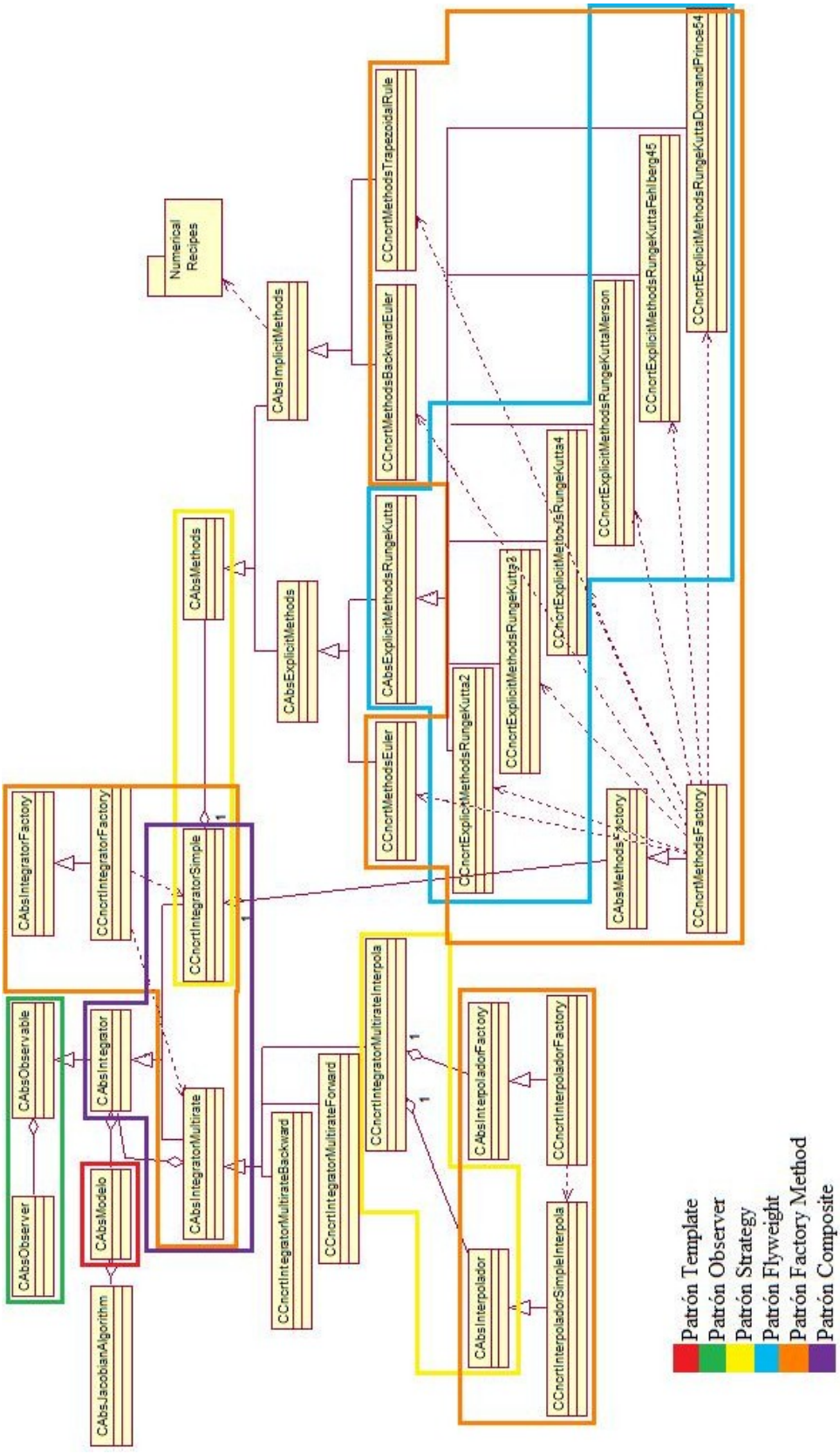


Figura 5.27: Patrones de diseño en la Arquitectura Multitasa propuesta.

Capítulo 6

Validación y verificación

De acuerdo a Sommerville [Som95], las tareas de validación y verificación se refieren a:

- Validación, comprobar que el sistema cumple con los requerimientos.
- Verificación, comprobar que el sistema se implementa de acuerdo al diseño propuesto.

En esta sección se presentan las pruebas realizadas a la arquitectura de software presentada en el capítulo anterior. Para la parte de validación, se presenta una métrica de software la cual se utiliza para comprobar que la arquitectura cumple con los requerimientos de diseño. Para la parte de verificación, fue necesario implementar el diseño de la arquitectura por lo que en esta parte se presentan las pruebas realizadas al sistema de software desarrollado.

6.1. Validación

Para validar la arquitectura de software propuesta se debe revisar que cumpla con los requerimientos de diseño propuestos. Primero mencionaremos las características más importantes que debe cumplir un diseño de software flexible y como se utilizan estas características para medir la flexibilidad del diseño. Después, se presenta una

métrica de software que ayuda a identificar y medir dependencias entre las entidades que conforman el diseño de la arquitectura propuesta.

6.1.1. Medidas de la calidad del diseño

Un buen diseño debería permitir producir código eficiente además de ser un diseño donde la implementación sea de fácil mantenimiento o lo más compacta posible. Entre la literatura de Ingeniería de Software no existe un acuerdo general sobre la noción de un buen diseño aparte del criterio obvio de que el diseño debe cumplir con la definición de los requerimientos.

Un diseño de fácil mantenimiento puede ser adaptado para modificar y/o agregar nuevas funcionalidades. Para lograr un diseño de este tipo es necesario que sea fácil de entender y que los cambios o modificaciones puedan realizarse de manera local.

Los componentes del diseño deben mantener cierto nivel de *cohesión*, ésto significa que todas las partes en un componente deben mantener una relación cercana. Por otro lado, los componentes deben evitar en lo posible las dependencias con otros componentes. El *acoplamiento* es la medida que estima la cantidad de dependencias entre componentes. Entre menos acoplados estén los componentes de un diseño de software, más fácil será realizar adaptaciones o cambios en él.

Cohesión

La *cohesión* es una medida para componentes. Se encarga de calcular la cercanía de las relaciones entre los elementos de los componentes. Un componente está encargado de realizar una operación o representar una entidad lógica, es decir, todas las partes del componente deben contribuir a la ejecución de una sola tarea.

La *cohesión* es una características deseable ya que significa que una unidad representa una sola parte de la solución al problema y, por lo tanto, no es necesario cambiar otros componentes si se requiere algún cambio en el diseño.

Acoplamiento

El *acoplamiento* esta relacionado con la *cohesión*, sin embargo, a diferencia de este último, el *acoplamiento* mide la dependencia entre los componentes del diseño. Un sistema altamente acoplado se caracteriza porque las dependencias entre los componentes son altas. Por otro lado, un sistema con bajo acoplamiento indica que los componentes operan de manera casi independiente. Para lograr un sistema con esta característica, es necesario evitar, en lo posible, el uso de información de tipo global.

Fácil entendimiento

El fácil entendimiento de un diseño es importante debido a que la persona que desee cambiarlo debe entenderlo primero. Algunos elementos que afectan el fácil entendimiento del diseño son, *cohesión*, *acoplamiento*, *nombres significativos* en los componentes, *documentación* de los componentes y la *complejidad de los algoritmos* implementados.

Capacidad de adaptación

En general, la capacidad de adaptación de un diseño se estima con la facilidad para cambiar a éste, es decir, que la dependencia entre sus componentes sea baja. Esta característica es similar a la *extensibilidad* de un sistema de software.

6.1.2. Métrica para evaluación

En esta sección se presenta la métrica propuesta por Martin en [Mar03], esta métrica determina las dependencias y abstracciones en la estructura de un diseño de software. Para aplicar esta métrica a la arquitectura multitasa propuesta es necesario construir *paquetes* a partir de las clases que conforman la arquitectura. Las reglas para la construcción de *paquetes* están dadas por las medidas de calidad del diseño, principalmente por la *cohesión* y el *acoplamiento*.

Un *paquete* es un contenedor para un grupo de clases. Las clases contenidas en

éstos guardan ciertas relaciones y, por lo tanto, las dependencias entre clases pueden crear dependencias entre paquetes. Las relaciones que existen entre los paquetes ofrecen un *nivel de organización superior* al obtenido mediante el uso de clases.

A continuación se presentan las consideraciones propuestas por Martin en [Mar03] para generar paquetes a partir de un diagrama de clases.

1. *Principio de equivalencia entre la reutilización y liberación de software* o REP por sus siglas en inglés. Este principio dice que la reutilización se basa en paquetes por lo que *los paquetes reutilizables deben contener clases reutilizables.*

2. *Principio de reutilización común* o CRP por sus siglas en inglés. Este principio dice que las clases que tienden a ser utilizadas en conjunto deben de pertenecer al mismo paquete, dicho de otra manera *las clases dentro de un paquete deben ser inseparables.*

3. *Principio de cerradura común* o CCP por sus siglas en inglés. Este principio dice que las clases que conforman un paquete deben ser cerradas a un mismo tipo de cambio, es decir, *un cambio al paquete afecta a todas las clases de ese paquete y no a otros.* Los siguientes principios complementan a éste.

    a) *Principio de responsabilidad común* o SRP por sus siglas en inglés. Aunque este principio aplica originalmente a clases, el cual dice que *las clases no deben contener múltiples razones para cambiar*, se puede aplicar de manera análoga a paquetes.

    b) *Principio de apertura-cerradura común* u OCP por sus siglas en inglés. Al igual que el principio anterior, aplica originalmente a clases. Este principio dice que las clases deben ser cerradas a modificaciones pero abiertas a extensiones. Al aplicarlo en paquetes indica que *las clases en el paquete deben ser abiertas al mismo tipo de cambios.*

6.1.3. Generación de paquetes

Con base en los principios mencionados, se procede a la generación de paquetes y se crea el diagrama de dependencias entre éstos. Hay que tener en cuenta que un diagrama de este tipo *no* muestra el comportamiento del sistema a desarrollar, sin embargo, es de gran utilidad para la etapa de implementación del software ya que con éste se puede repartir el trabajo implicado en la tarea de desarrollo de software. Los paquetes generados y la manera en que se llegó a ellos se presenta en esta sección.

El primer paquete obtenido es **PModelo**, se obtiene a partir del principio REP y por ser el único que cambiaría al momento de definir un nuevo problema con valores de inicio, éste se presenta en la figura 6.1. La clase **CAbsJacobianAlgorithm** queda fuera del paquete por el principio CCP ya que no siempre es necesario definir una función Jacobiana para los nuevos problemas con valores de inicio definidos.

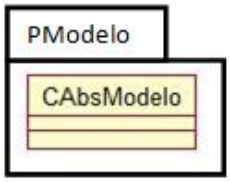


Figura 6.1: Paquete para los modelos de problemas con valores de inicio.

La clase **CAbsJacobianAlgorithm** queda entonces dentro de otro paquete llamado **PJacobiano** que será modificado únicamente cuando se desee establecer una función específica para el cálculo del Jacobiano. En la figura 6.2 se presenta este paquete.

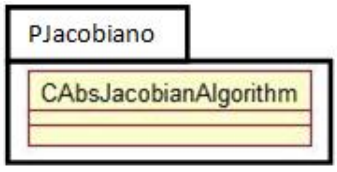
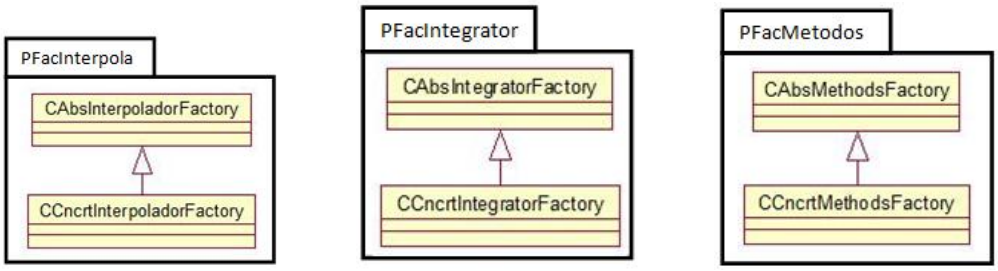


Figura 6.2: Paquete para el algoritmo Jacobiano.

Los paquetes **PFacInterpola**, **PFacIntegrator** y **PFacMetodos** se generan a partir de los principios CCP y SRP ya que al agregar un nuevo interpolador, método

integrador o método estándar, respectivamente, es necesario indicar a las fábricas para que se encarguen de instanciarlos. En la figura 6.3 se presentan estos paquetes.



(a) Fábrica interpoladores (b) Fábrica integradores (c) Fábrica métodos estándar

Figura 6.3: Fabricas de objetos.

El paquete **PMultirate** se obtiene al aplicar los principios CRP, SRP y CCP ya que las clases concretas, que definen los algoritmos multitasa, dependen de la clase abstracta **CAbsIntergratorMultirate**. La única razón por la que se tendrían que realizar cambios a este paquete es porque se desee modificar o agregar una nueva estrategia multitasa. En el caso de agregar una nueva estrategia multitasa hay que dar aviso de su existencia al paquete **PFacIntegrator**. En la figura 6.4 se presenta el contenido de este paquete.

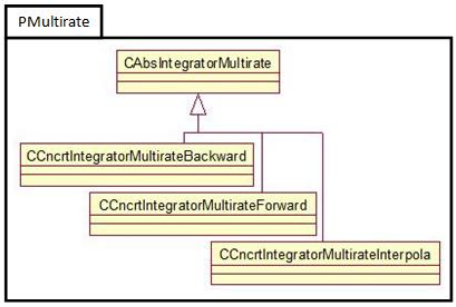
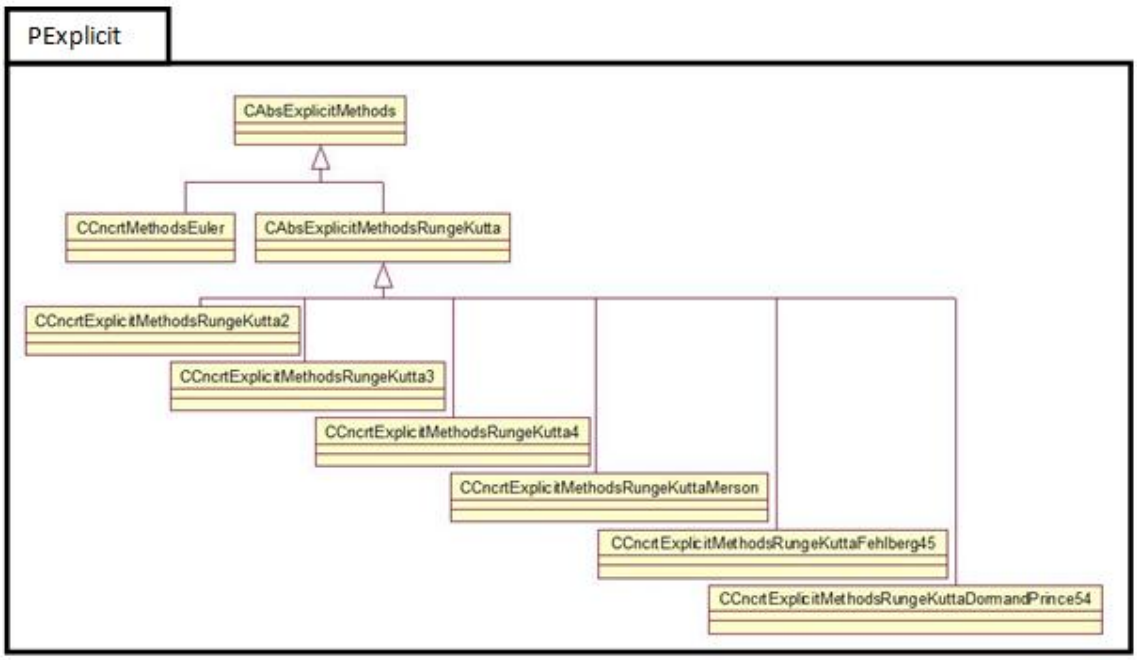


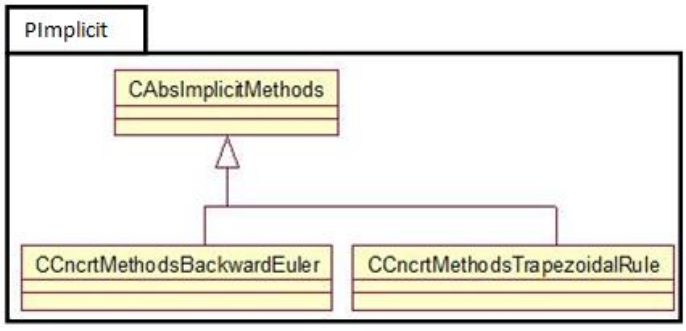
Figura 6.4: Paquete representante de los métodos multitasa.

Los paquetes **PExplicit** y **PImplicit** se generan con ayuda de los principios CCP, CRP y SRP. Al dividir los tipos de métodos numéricos estándares, se logra que al agregar un método de un tipo, por ejemplo explícito, solamente sea necesario realizar

modificaciones en el primer paquete. Lo mismo sucede para el paquete **PImplicit**, solamente en el caso de agregar un nuevo método hay que dar aviso al paquete **PFacMetodos** para que pueda ser generado. Estos paquetes se presentan en la figura 6.5.



(a) Métodos Explícitos



(b) Métodos Implícitos

Figura 6.5: Paquetes para los dos tipos de métodos numéricos.

El paquete **PInterpolador** se obtiene a partir del principio CCP y SRP ya que al modificar o agregar un nuevo algoritmo interpolador hay que cambiar o extender las clases de este paquete y sólo en caso de agregar un nuevo algoritmo interpolador se da aviso al paquete **PFacInterpola**. En la figura 6.6 se presenta este paquete.

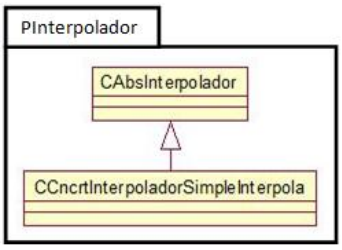


Figura 6.6: Paquete para los métodos interpoladores.

El paquete **PObservable** se crea al aplicar el principio REP ya que se trata de clases altamente reutilizables. Este paquete contiene la interfaz de los algoritmos multitasa para comunicarse con otros sistemas, la figura 6.7 muestra este paquete.

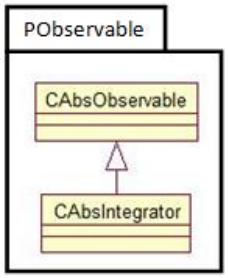


Figura 6.7: Paquete representando las clases observables reutilizables.

El paquete **PObserver** se obtiene al aplicar los principios CCP y SRP. Al agregar un nuevo elemento observador sólo es necesario extender este paquete. En la figura 6.8 se presenta este paquete.

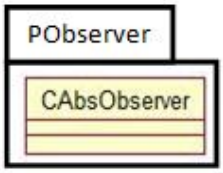
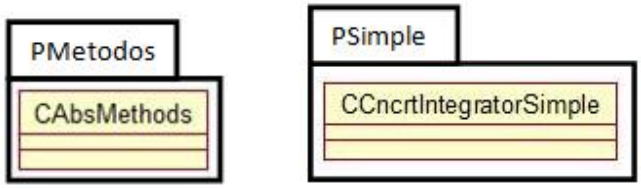


Figura 6.8: Paquete reutilizable por los observadores.

Para las clases restantes se genera un paquete a cada una, el primero, **PMetodos**, generado con base en el principio REP, contiene la abstracción de los métodos numéricos y la interfaz para comunicarse con otros sistemas. El segundo, **PSimple** encargado de la comunicación entre los métodos numéricos multitasa y los métodos

numéricos estándares. La generación de un solo paquete a partir de estos no es posible debido a los principios CCP y SRP. En la figura 6.9 se presentan estos paquetes y en la figura 6.10 se muestran las dependencias entre los paquetes generados.



(a) Abstracción de métodos

(b) Comunicación estándar - multitasa

Figura 6.9: Paquetes para las clases restantes.

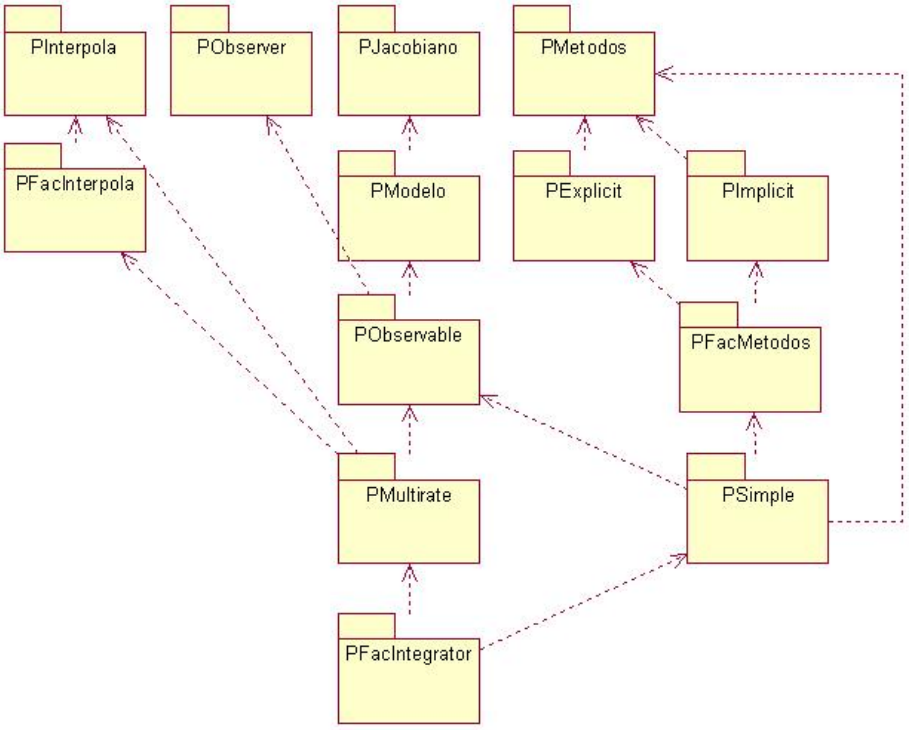


Figura 6.10: Dependencia entre los paquetes generados.

Estabilidad

Para lograr flexibilidad en un diseño es necesario que no todas las partes que lo componen sean estables, es decir, que existan elementos inestables o diseñados para

el cambio. Cuando aplicamos el principio CCP creamos paquetes sensibles a cierto tipo de cambio, es decir, paquetes *volátiles*.

Cualquier paquete que esperemos vaya a cambiar no debe depender de paquetes difíciles de cambiar, es decir, de paquetes estables, ya que debido a esta dependencia la tarea de cambio se dificultaría. Al aplicar el *principio de dependencias estables* o SDP por sus siglas en inglés, el cual menciona que ningún paquete debe depender de paquetes menos estables que él, podemos asegurar que los paquetes orientados al cambio no dependan de paquetes difíciles de cambiar.

En este contexto, *estabilidad* se relaciona con la cantidad de trabajo necesario para realizar un cambio. Por ejemplo, un paquete de software del cual dependen varios paquetes es muy estable ya que al removerlo se requiere trabajar con las clases dependientes para adaptarlas al cambio realizado. Se dice entonces que el paquete es *responsable* de sus dependientes y si no depende de otros paquetes entonces es *independiente*.

De acuerdo a Martin [Mar03], una manera de medir la estabilidad es contar tanto el número de dependencias que entran al paquete como las que salen. Esto nos permite calcular la estabilidad posicional del paquete.

- C_a es el número de clases fuera del paquete que dependen de clases dentro del paquete.
- C_e es el número de clases dentro del paquete que dependen de clases fuera del paquete.
- I de inestabilidad

$$I = \frac{C_e}{C_a + C_e} \tag{6.1.1}$$

La métrica tiene un rango entre cero y uno, donde $I = 0$ indica un paquete completamente estable, es decir, otros paquetes dependen de él pero éste no depende de otros. Se dice entonces que es un paquete *responsable* e *independiente*. Por otro lado, cuando $I = 1$ indica que el paquete es completamente inestable, es decir, ningún paquete depende de él pero éste si depende de otros. En este caso se dice que el paquete es *irresponsable* y *dependiente*.

Al aplicar esta métrica a la arquitectura multitasa se obtienen los resultados de la columna *(I) Inestabilidad* de la tabla 6.1.

Recordemos que hay partes en la arquitectura que no deberían cambiar muy a menudo, es decir, las abstracciones de alto nivel realizadas durante el proceso de diseño. El software que encapsule el diseño de alto nivel debería ser puesto en paquetes con medida $I = 0$. Y los paquetes inestables, es decir, con $I = 1$ deberían contener elementos que muy probablemente cambiarán.

Sin embargo, si ponemos el diseño de alto nivel en paquetes altamente estables se pierde *flexibilidad*, la pregunta es ¿cómo maximizar el valor I de los paquetes pero que además sean *flexibles* al cambio?

Abstracción

Las clases abstractas permiten extender un diseño para volverlo flexible sin necesidad de hacer modificaciones a los elementos base. Un paquete estable debería también ser abstracto de modo que pueda ser extendido, mientras que un paquete inestable debería ser concreto lo que permitiría ser sustituido fácilmente.

La métrica A definida por Martin [Mar03] es una medida de abstracción para paquetes, es decir, mide la razón de clases abstractas en un paquete.

- N_c es el número de clases en el paquete.
- N_a es el número de clases abstractas en un paquete.
- A de abstracción

$$A = \frac{N_a}{N_c} \tag{6.1.2}$$

En la columna *(A) Abstracción* de la tabla 6.1 se presentan los resultados de aplicar esta métrica.

Relación (I) Inestabilidad - (A) Abstracción

Al poner en una gráfica a I en el eje horizontal y a A en el vertical, obtenemos una manera de relacionar la cantidad de *inestabilidad* con la *abstracción* de un paquete.

De acuerdo a Martin, los paquetes que son altamente estables y abstractos se encuentran en la esquina superior izquierda $(0, 1)$, mientras que los paquetes inestables y concretos en la esquina inferior derecha $(1, 0)$.

Ya que no todos los paquetes de un diseño de software pueden estar en los puntos $(0, 1)$ y $(1, 0)$ se asume que existen zonas razonables para la posición de los paquetes, éstas se encuentran al analizar las zonas en donde los paquetes *no* deberían estar y se describen a continuación.

En la zona cercana a $(0, 0)$ se encuentran los paquetes altamente estables y concretos. Este tipo de paquetes no son deseables ya que no pueden ser extendidos al no ser abstractos además de ser muy difíciles de cambiar debido a su estabilidad. La zona cercana a $(1, 1)$ es la de los paquetes completamente abstractos y que no presentan dependencias, es decir, tales paquetes no se utilizan.

La zona que se encuentra más lejos de las zonas de exclusión es la línea que conecta a los puntos $(0, 1)$ y $(1, 0)$, esta zona es conocida como *main sequence*. La distancia normalizada de un paquete a la zona definida por *main sequence* se calcula de la siguiente manera:

$$D = |A + I - 1| \tag{6.1.3}$$

Donde $D = 0$ indica que el paquete se encuentra en la zona definida por *main sequence* y $D = 1$ sugiere que el paquete está lo más lejos posible de esta zona. Los resultados obtenidos al evaluar esta métrica en los paquetes generados se muestran en la tabla 6.1. En la figura 6.11 se presenta la posición de los paquetes respecto a la zona *main sequence*. En esta gráfica se observa que la mayoría de los paquetes se encuentran cerca de la zona *main sequence* por lo que son altamente reutilizables y adaptables al cambio. Los paquetes a mayor distancia, $D = 0.5$ en este caso, se encuentran a mitad del camino por lo que necesitarán futuras modificaciones para mejorar su flexibilidad. Es importante notar que ningún paquete se encuentra en las zonas que definen software *no* reutilizable e inservible.

Nombre del paquete	(I) Inestabilidad	(A) Abstracción	(D) Distancia
PModelo	0.5	1.0	0.5
PJacobiano	0.0	1.0	0.0
PFacInterpola	0.5	0.5	0.0
PFacIntegrator	1.0	0.5	0.5
PFacMetodos	0.5	0.5	0.0
PMultirate	$0.66\bar{6}$	0.250	0.083
PExplicit	0.5	$0.22\bar{2}$	$0.27\bar{7}$
PImplicit	0.5	$0.33\bar{3}$	$0.16\bar{6}$
PInterpolador	0.0	0.5	0.5
PObservable	0.5	1.0	0.5
PObserver	0.0	1.0	0.0
PMetodos	0.0	1.0	0.0
PSimple	0.5	0.0	0.5

Tabla 6.1: Resultados obtenidos al aplicar la métrica propuesta.

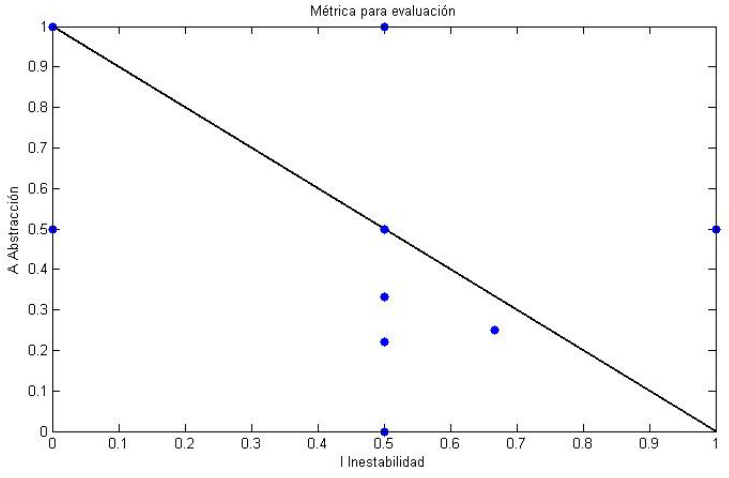


Figura 6.11: Interpretación gráfica de los resultados obtenidos.

6.1.4. Análisis y discusión de resultados

Como se ha mencionado a lo largo del capítulo, una arquitectura que presuma de ser flexible debe cumplir con las medidas de calidad de software presentadas, alta cohesión entre los elementos que conforman a los paquetes para que las modificaciones se realicen en el menor número de entidades posible. Por otro lado bajo acoplamiento entre los paquetes para garantizar flexibilidad al cambio.

En los requerimientos se plantean las características deseables de la arquitectura, el primero de ellos habla sobre una interfaz consistente e independiente tanto del problema a resolver como del método numérico a utilizar. En el diagrama de dependencias entre paquetes, presentado en la figura 6.10, se puede observar que los métodos numéricos presentan una interfaz consistente, ya que cualquiera que sea el tipo, explícito o implícito, debe utilizar la interfaz dictada por el paquete **PMetodos**. Además, este paquete no presenta dependencias con el paquete **PModelo**, encargado de establecer la interfaz para los problemas a resolver, lo cual significa que los métodos numéricos son independientes de los problemas a resolver.

La tarea de agregar un nuevo método numérico debe de ser una tarea sin dificultades de acuerdo a los requerimientos presentados. En el diagrama de dependencias de paquetes de la figura 6.10, se observan las consecuencias de agregar un nuevo método, éstas implican modificar, ya sea el paquete **PExplicit** o **PImplicit** dependiendo del tipo de método a agregar e informar al paquete **PFacMetodos** de la existencia de un nuevo método numérico. Basado en los valores obtenidos al aplicar la métrica, los paquetes involucrados en la tarea de agregar un nuevo método numérico presentan muy buenas características para ser extendidos y reutilizados.

De acuerdo al mismo diagrama, al agregar un nuevo método numérico sería necesario modificar también el paquete **PSimple**, sin embargo, esto no es del todo cierto ya que éste solo se encarga de encapsular el método estándar dejándolo listo para ser utilizado por un método multitasa, por lo que los cambios en este paquete no son necesarios.

La selección del método a utilizar se lleva a cabo con la ayuda de los patrones de

diseño *Strategy* y *Factory Method*, el primero encargado de la selección del método y el segundo responsable de la creación del objeto que encapsula al método. Con la intervención de estos patrones esta tarea no presenta mayores dificultades, además de que permiten intercambiar el método numérico en tiempo de ejecución.

En caso de que fuera necesario alterar algún método numérico de los ya establecidos, sólo se necesita modificar el paquete en donde éste se encuentre, esto es gracias al encapsulamiento de datos ofrecido por el paradigma orientado a objetos. En todo caso, siempre es posible agregar un nuevo método con las características particulares necesarias.

Los métodos implícitos, los cuales se encuentran dentro del paquete **PImplicit**, hacen uso de las funciones **ludcmp** y **lubksb** de la librería *Numerical Recipes* para ayudar a resolver un sistema de ecuaciones no lineales, sin embargo, gracias a la característica de encapsulamiento, los métodos numéricos *no saben* de la existencia de estas funciones y utilizan una función en común para esta tarea, esta función se encarga de resolver el sistema de ecuaciones no lineales. El usuario que agregue un nuevo método implícito solamente tiene que llamar a la función mencionada cuando necesite resolver el sistema de ecuaciones generado. De acuerdo a Gamma *et al.* [GHJV95], esta es una característica propia del patrón *Facade*, el cual se encarga de definir una interfaz sencilla para la comunicación con elementos complejos.

Agregar un nuevo método interpolador es similar a agregar un nuevo método numérico, sólo hay que agregar el método interpolador en el paquete **PInterpola** e informar de su existencia al paquete **PFacInterpola**. Con base en los valores obtenidos por la métrica, no deben existir problemas en esta tarea.

La selección de un algoritmo multitasa no debe presentar dificultades según los requerimientos de diseño. Los paquetes encargados de esta tarea son **PMultirate**, encargado de la sincronización y administración de la información producida por los métodos multitasa, y **PFacIntegrator** que tiene la responsabilidad de crear el método multitasa con la configuración de métodos indicada. Esta tarea se lleva a cabo por medio de la intervención de los patrones *Composite* y *Factory Method*, el

primero se encarga de crear una estructura compleja de objetos dando lugar a los métodos multitasa, y el segundo se ocupa de ocultar la complejidad de la tarea de creación de métodos multitasa. Entonces, para seleccionar un método multitasa es necesario indicar solo tres parámetros a la fábrica creadora:

1. Estrategia multitasa.
2. Método numérico para integrar la parte lenta.
3. Método numérico para integrar la parte rápida.

La tarea de agregar un nuevo problema es una de las que con mayor frecuencia se realizará, esta tarea involucra modificar los paquetes **PModelo** y **PObserver**, este último no siempre es modificado. El primero, guarda la parte abstracta para todos los modelos de problemas y debe extenderse para crear una subclase que represente al problema con valores de inicio que se desea resolver. El segundo paquete, contiene la clase abstracta de los observadores, es decir, de los objetos que se encargaran de manipular los datos de salida. Las modificaciones a este paquete pueden ser tan sencillas como solicitar la impresión de datos en pantalla o tan sofisticada como conectar algún dispositivo que reciba los datos generados por los integradores. En el diagrama de dependencias de clases de la figura 6.10 se observan dependencias con el paquete **PObservable**, sin embargo, no es necesario modificar este paquete ya que solo define la interfaz *observable* para las estrategias multitasa. La dependencia que presentan se debe a que los métodos multitasa necesitan un modelo que resolver.

6.1.5. Una métrica modificada: Composición vs Herencia

En el análisis anterior encontramos que muchos paquetes que deberían ser modificados de acuerdo al diagrama de dependencias de la figura 6.10, no lo son, ésto se debe a que la mayoría de las dependencias están dadas por la *composición de objetos*, es decir, las responsabilidades de un objeto las lleva a cabo con la colaboración de las responsabilidades de varios objetos. Los patrones de diseño hacen uso exhaustivo de esta característica y por tanto se presenta en gran medida en la arquitectura diseñada.

Uno de los principios en el desarrollo de software orientado a objetos, dice que hay que *favorecer la composición de objetos sobre la herencia de clases*. Los expertos en el diseño de software basan sus diseños en este principio, y por lo tanto, se encuentra incluido en los patrones de diseño.

La métrica presentada, propuesta por Martin en [Mar03], no distingue entre las dependencias de paquetes dadas por: herencia, composición o asociación. Para lograr una evaluación que exhiba en su totalidad los beneficios al diseñar software con ayuda de patrones se propone, como una de las aportaciones de este trabajo, realizar algunas modificaciones a la métrica de Martin.

Con base en lo expuesto por Gamma *et al.* en [GHJV95], sobre la ventaja de la composición de objetos ante la herencia de clases, se propone distinguir entre los tipos de dependencia, específicamente las dependencias por herencia y las dependencias por composición o asociación, representadas por DH y DC respectivamente.

Para lograr esta distinción se modifican las reglas para calcular el valor de *(I) Inestabilidad*, quedando de la siguiente manera.

- $C_a = DH + \alpha DC$, número de clases fuera del paquete que dependen de clases dentro del paquete por herencia *(DH)* y por composición *(DC)*.
- $C_e = DH + \alpha DC$, número de clases dentro del paquete que dependen de clases fuera del paquete por herencia *(DH)* y por composición *(DC)*.
- I de inestabilidad

$$I = \frac{C_e}{C_a + C_e} \tag{6.1.4}$$

En caso de que alguna clase dependa o sea dependiente tanto por herencia como por composición, se le da prioridad a la herencia por ser una dependencia más fuerte.

La variable α se utiliza para indicar el peso de las dependencias dadas por composición. Debido a que los patrones de diseño buscan crear diseños con dependencias de este tipo, y éstos, son considerados buenos, se propone no dar importancia a este tipo de dependencias por lo que $\alpha = 0$.

Los valores obtenidos para *(I) Inestabilidad* con esta modificación, junto con los valores de las otras medidas se presentan en la tabla 6.2.

Nombre del paquete	(I) Inestabilidad	(A) Abstracción	(D) Distancia
PModelo	0.0	1.0	0.0
PJacobiano	0.0	1.0	0.0
PFacInterpola	0.0	0.5	0.5
PFacIntegrator	0.0	0.5	0.5
PFacMetodos	0.0	0.5	0.5
PMultirate	1.0	0.250	0.250
PExplicit	1.0	$0.22\overline{2}$	$0.22\overline{2}$
PImplicit	1.0	$0.33\overline{3}$	$0.33\overline{3}$
PInterpolador	0.0	0.5	0.5
PObservable	0.0	1.0	0.0
PObserver	0.0	1.0	0.0
PMetodos	0.0	1.0	0.0
PSimple	1.0	0.0	0.0

Tabla 6.2: Resultados obtenidos al aplicar la métrica modificada.

La gráfica que presenta la distancia de los paquetes respecto a la zona definida por *main sequence* se presenta en la figura 6.12. En esta gráfica se observa, al igual que en la figura 6.11, que la mayoría de los paquetes se encuentran cerca de la zona *main sequence* además de que el número de paquetes a mayor distancia se ha reducido. Estos paquetes, con $D = 0.5$, son en mayoría paquetes de fábricas, es decir, se encargan de la generación de métodos numéricos estándares, métodos multitasa o algoritmos de interpolación. Para que puedan acercarse a la zona *main sequence* será necesario incrementar su *inestabilidad (I)* y/o *abstracción (A)*. Para lograr ésto, en el caso de la medida *(I)* sería necesario que otros paquetes dependieran de estos a través de la herencia, y, para el caso de la medida *(A)* hacer que las fábricas generen *familias* de métodos, por ejemplo, para el caso de métodos numéricos, generar las familias Runge Kuttas, Adams, BDF, entre otros.

Al aplicar esta nueva métrica se elimina el efecto de las dependencias por agregación o asociación. Por ejemplo, la dada por los métodos multitasa con el problema a resolver, la cual existe porque el método multitasa necesita un modelo o problema a resolver. En la figura 6.13 se presenta el diagrama de dependencias entre paquetes, las dependencias por herencia se indican con color azul, las demás son por composición.

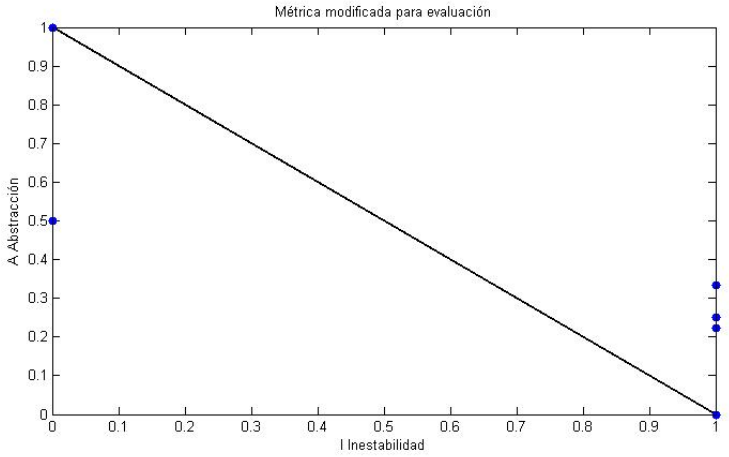


Figura 6.12: Resultados obtenidos con la métrica modificada.

6.1.6. Ventajas de la arquitectura

Entre los requerimientos de diseño, se encuentra el de una interfaz intuitiva para los usuarios no experimentados, ésta se logra gracias a que los patrones de diseño se encargan de establecer tales interfaces.

La arquitectura permite resolver problemas con valores de inicio que presentan diferentes escalas de tiempo, sin embargo, si fuera necesario realizar pruebas con problemas que no presenten tales características o simplemente se quieren utilizar métodos numéricos estándares, es posible hacerlo y sin necesidad de modificaciones. Estas características hacen de la arquitectura ser considerada como de *propósito general*. No obstante, no puede ser considerada *robusta*, no por el diseño, sino por la falta de una teoría de estabilidad completa en lo que respecta a los Métodos de Integración Multitasa.

6.1.7. Patrones de diseño utilizados en la arquitectura

- *Template*, este patrón permite definir el esqueleto del algoritmo de inicialización para los problemas con valores de inicio, es decir, asignación de condiciones de inicio, establecer el orden de evaluación de las ecuaciones y definición de las ecuaciones del problema.

- *Observer*, con este patrón se logra establecer una interfaz para definir compo-

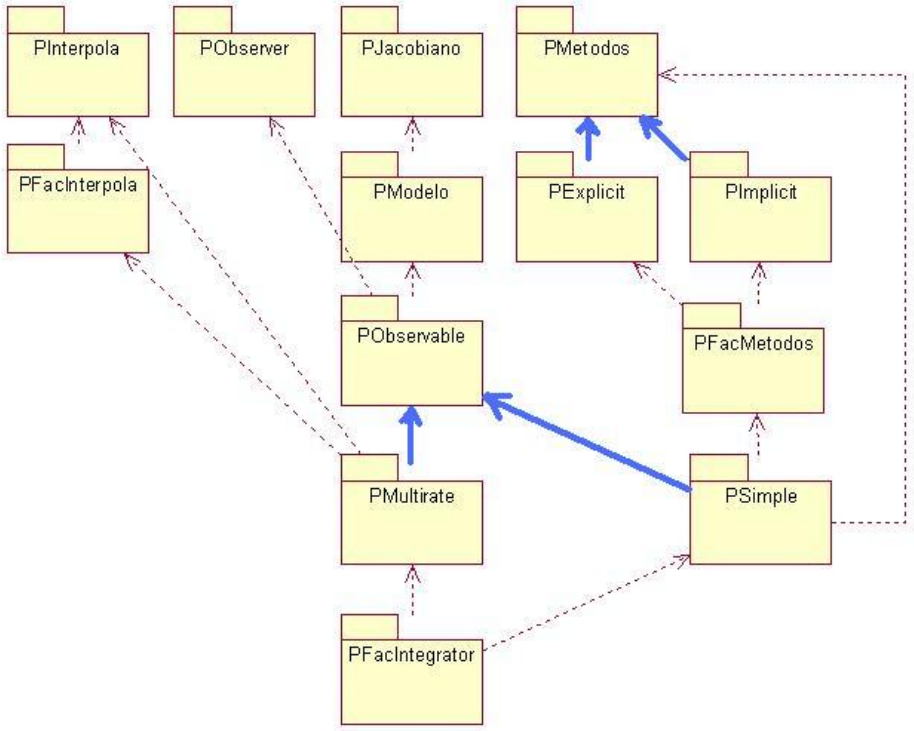


Figura 6.13: Dependencia de paquetes diferenciando entre el tipo de dependencias.

nentes que se encarguen de la manipulación y administración de los datos de salida.

- *Strategy*, con la ayuda de este patrón se logra seleccionar de manera dinámica el método numérico estándar, la estrategia de integración multitasa y el algoritmo de interpolación, utilizado por la estrategia de interpolación multitasa.

- *Factory method*, este patrón se encarga de la creación de los objetos tipo método numérico estándar, método multitasa y algoritmo de interpolación. Define también una interfaz común para la creación de estos objetos.

- *Flyweight*, con este patrón se logra implementar la familia de métodos explícitos Runge Kutta al encapsular el algoritmo general para que los métodos específicos se encarguen de configurarlo y funcione como tal.

- *Composite*, gracias a este patrón se pueden crear diferentes configuraciones de estrategias multitasa. Al combinarse con el patrón *Factory Method* se logra la creación de objetos complejos.

6.2. Verificación

Como parte de las tareas de verificación de la arquitectura se encuentra la búsqueda de errores en la implementación. Para llevar a cabo esta tarea se utilizaron datos simulados para comprobar que los subsistemas estuvieran bien implementados, tanto lógica como estructuralmente. Al encontrar un error, se buscaba su origen y se reparaba, a esta técnica de verificación se le conoce como depuración (*debugging*).

Con la ayuda de esta técnica se verificó la correcta implementación del diseño siguiendo el flujo de datos a través de la arquitectura y comprobando que los objetos participantes tomaran parte en la tarea del procesamiento de datos al verificar la traza de ejecución.

La metodología utilizada para la corrección de errores en la implementación del diseño se conoce como *regression testing*, ésta se describe a continuación:

1. Al encontrar defectos en la implementación del diseño se busca el origen de éstos.
2. Se identifica el tipo de error y se corrige.
3. Se prueba nuevamente la implementación para comprobar que los cambios realizados no generaron nuevos errores.

6.2.1. Implementación

La implementación de la arquitectura multitasa se hizo en lenguaje C++. La interfaz de desarrollo elegida fue *Borland Builder C++ 6.0* por las facilidades ofrecidas para el diseño de interfaces gráficas.

Verificación estática

El proceso de verificación involucra la etapa de *verificación estática*, ésta se relaciona con la búsqueda de defectos en la implementación que puedan causar corrupción de datos y por lo tanto errores en los resultados. La verificación estática debe ser utilizada como un proceso de verificación inicial para encontrar la mayoría de los defectos en un programa.

La verificación estática, en general, no requiere de la ejecución del programa, más bien involucra la revisión del código fuente en busca de anomalías, por una parte lógicas y por otra de codificación. Sin embargo, debido a la dificultad para encontrar anomalías lógicas sin la ejecución del programa, se utilizan herramientas de software que buscan por este tipo de problemas. Las herramientas de verificación exploran por anomalías lógicas buscando variables que no sean inicializadas o utilizadas, fragmentos de código inalcanzables, asignación de tipos inconsistentes, entre otros.

La interfaz de desarrollo seleccionada, *Builder*, cuenta con una herramienta de verificación integrada llamada $CodeGuard^{TM}$ [HSCG03]. Esta herramienta fue utilizada durante todo el proceso de implementación del diseño de software.

6.2.2. Pruebas

En esta etapa, implementación, se verificó la correcta implementación de los algoritmos numéricos. Para esto se resolvieron diferentes problemas planteados como problemas con valores de inicio. Primero se evaluaron los métodos numéricos estándares para luego evaluar a los métodos multitasa.

A continuación se presenta un sistema de EDO's para probar a los método estándares.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

Las funciones elípticas de Jacobi dan solución a este problema, ver [SG75].

$$\begin{aligned} y_1(x) &= sn(x|0.51) \\ y_2(x) &= cn(x|0.51) \\ y_3(x) &= dn(x|0.51) \end{aligned}$$

El periodo de estas funciones está dado por $p = 4\kappa$ con $\kappa = 1.862640802332739$. Las soluciones para $x = j\kappa$ se obtienen con la ayuda de la tabla 6.3

j	$y_1(j\kappa)$	$y_2(j\kappa)$	$y_3(j\kappa)$
0	0.0	1.0	1.0
1	1.0	0.0	0.7
2	0.0	-1.0	1.0
3	-1.0	0.0	0.7

Tabla 6.3: Valores solución para 4 puntos en el periodo $p = 4\kappa$.

En la tabla 6.4 se presentan las diferencias entre los resultados obtenidos por la arquitectura multitasa y los valores solución de la tabla 6.3. Para esta prueba se utilizó un método *Runge Kutta Fehlberg 45* con $h = 0.05$, seleccionado como el mínimo de los pasos utilizados por la rutina adaptativa *ode45* de *MatLab*.

j	$y_1(j\kappa)$	$y_2(j\kappa)$	$y_3(j\kappa)$
0	0.0	0.0	0.0
1	3.42×10^{-4}	0.026	2.49×10^{-4}
2	0.024	3.05×10^{-4}	1.56×10^{-4}
3	3.6×10^{-5}	8.454×10^{-3}	2.60×10^{-5}

Tabla 6.4: Diferencias en la solución para 4 puntos en el periodo $p = 4\kappa$.

Los resultados obtenidos por la arquitectura muestran que el algoritmo numérico *Runge Kutta Fehlberg 45* ha sido implementado de manera correcta. Durante el proceso de implementación se realizaron pruebas similares para todos los métodos numéricos implementados.

Pruebas Multitasa

En esta sección se presenta la evaluación de la arquitectura multitasa con problemas que muestran diferentes escalas de tiempo. Las pruebas fueron realizadas en una computadora con procesador *AMD Turion* 64×2 con $1Gb$ de memoria RAM.

A continuación se presenta un problema con valores de inicio con dos escalas de tiempo.

$$y_1' = 3y_2 - 5y_1 + \sin(0.05 * t) \quad y_1(0) = 1$$
$$y_2' = -25y_2 + \sin(20 * t) \qquad y_2(0) = 1$$

La ecuación y_2' presenta cambios rápidos respecto a y_1', es decir, y_2' es la *parte rápida* y y_1' la *parte lenta*. En la tabla 6.5 se muestra el tiempo de cómputo al utilizar un método estándar y un método multitasa. Además, se presenta la diferencia de los valores obtenidos por la arquitectura multitasa con los obtenidos por las rutinas *ode45* y *NDSolve*, de *MatLab* y *Mathematica* respectivamente. Esta diferencia se calcula como la norma infinita de la diferencia de los valores obtenidos por la arquitectura, representados por $\mathbf{y}$, contra los de *ode45* representados por **ML** y los de *NDSolve* representados por **MT**. Es decir, $E_{ML} = ||\mathbf{y}(T) - \mathbf{ML}(T)||_\infty$ y $E_{MT} = ||\mathbf{y}(T) - \mathbf{MT}(T)||_\infty$ con $T = 1$ para este ejemplo.

Método numérico	Tiempo (ms)	Tamaño del paso	$E_{ML}(T=1)$	$E_{MT}(T=1)$
RungeKuttaF45	123.40	$h = 8.0 \times 10^{-3}$	8.0×10^{-6}	4.0×10^{-7}
Multitasa (R)RungeKutta2 (L)RungeKuttaF45	113.57	$H = 0.016$ $h = 8.0 \times 10^{-3}$	4.245×10^{-3}	4.240×10^{-3}

Tabla 6.5: Resultados obtenidos al aplicar métodos multitasa.

El tamaño del paso h se selecciona como el mínimo de los pasos utilizados por la rutina adaptativa *ode45* de *MatLab*.

En el ejemplo anterior no se presentan los tiempos de cómputo de las rutinas *ode45* y *NDSolve* debido a que éstas son de tipo adaptativo y necesitan, además de la rutina integrador, tiempo de cómputo extra para la ejecución de tareas para la selección del paso de integración. En el caso específico de la rutina *NDSolve*, el tiempo de cómputo de la manipulación simbólica realizada por *Mathematica* es incluido en el tiempo total de ejecución. Por estas razones, sólo se presentan los tiempos de ejecución de los métodos numéricos implementados en la arquitectura multitasa propuesta.

Los resultados obtenidos no muestran gran diferencia en los tiempos de ejecución de un método numérico estándar a uno multitasa, esto se debe al número reducido de ecuaciones del sistemas de EDOs. En el siguiente ejemplo se presenta un sistema

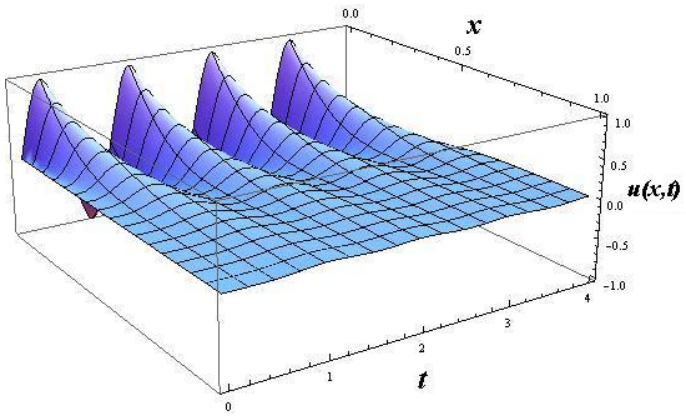


Figura 6.14: Superficie solución obtenida por *Mathematica*, $0 \leq x \leq 1$ y $0 \leq t \leq 4$.

de EDO's con un mayor número de ecuaciones el cual se obtuvo al aplicar el método de líneas a la siguiente EDP con condiciones de *Dirichlet*.

$$\mu_t = \frac{1}{8}\mu_{xx} \tag{6.2.1}$$

$$\mu(0,t) = \sin(2\pi t) \quad mu_x(1,t) = 0 \quad \mu(x,0) = 0 \tag{6.2.2}$$

La solución calculada por el software *Mathematica* se presenta de manera gráfica en la figura 6.14, donde es posible identificar al menos dos escalas de tiempo.

Al aplicar el método de líneas al problema definido por 6.2.1 y 6.2.2 se obtiene el sistema de EDOs presentado a continuación.

$$\begin{aligned}
\mu_0' &= 2\pi\cos(2\pi t) && \mu_0(0) = 0 \\
\mu_i' &= \frac{1}{8}(100\mu_{i-1} - 200\mu_i + 100\mu_{i+1}) && \mu_i(0) = 0 \quad i = 1,\ldots,9 \\
\mu_{10}' &= \frac{1}{8}(200\mu_9 - 200\mu_{10}) && \mu_{10}(0) = 0
\end{aligned}$$

Podemos dividir el sistema de ecuaciones en dos, las ecuaciones $\mu_i, i = 2,\ldots,10$ representarán a las *componentes lentas* y las ecuaciones $\mu_i, i = 0, 1$ a las *componentes rápidas*. La figura 6.15 presenta los resultados al aplicar un método multitasa.

En la tabla 6.6 se presenta el tiempo de ejecución de un método numérico estándar y uno multitasa. En la misma tabla se muestran las diferencias con los resultados obtenidos por las rutinas *ode45* y *NDSolve* de *MatLab* y *Mathematica* respectivamente al termino del intervalo de simulación, el cual fue de $0 \leq t \leq 4$.

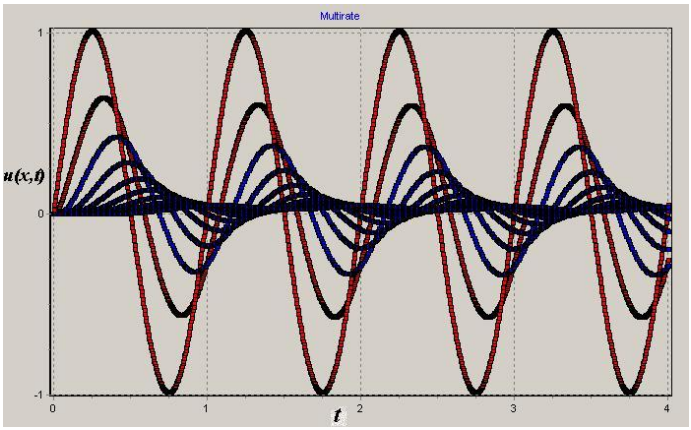


Figura 6.15: Solución obtenida con un método multitasa, las componentes rápidas se muestran de color rojo y las lentas de color azul.

Método numérico	Tiempo (ms)	Tamaño del paso	$E_{ML}(T=4)$	$E_{MT}(T=4)$
Euler	3244.85	$h = 3.90625 \times 10^{-3}$	0.024544	0.024543
Multitasa (R)Euler (L)Euler	1716.05	$H = 7.8125 \times 10^{-3}$ $h = 3.90625 \times 10^{-3}$	0.049080	0.049079

Tabla 6.6: Resultados al aplicar un método multitasa y el método de líneas.

En este caso se observa una reducción de un 47.12 % en el tiempo de ejecución a pesar de que únicamente dos ecuaciones representaron a la *parte rápida*. El error cometido tanto por el método estándar *Euler* como por el método multitasa *Euler-Euler* respecto a los valores obtenidos por *ode45* y *NDSolve* se encuentran dentro de los márgenes de error para aplicaciones de simulación en tiempo real, donde la velocidad en cálculos y respuesta del sistema son prioritarios; ver *4.1 Simulator Capabilities Criteria* de la norma ANSI / ANS 3.5 - 1985 en [ESW+85]. El objetivo de estas pruebas fue el de mostrar la diferencia en tiempos de ejecución de los métodos.

Si fuera necesario, una manera de disminuir el error sería reducir el paso de integración, ésto implicaría un incremento en el tiempo de ejecución. En la tabla 6.7 se presentan los resultados al reducir el tamaño del paso de integración con $h = 0.001$ y $H = 0.002$. En esta caso, la reducción en el tiempo de ejecución es del 25.31 %.

Método numérico	Tiempo (ms)	Tamaño del paso	$E_{ML}(T=4)$	$E_{MT}(T=4)$
Euler	7064.0	$h = 1.0 \times 10^{-3}$	6.283×10^{-3}	6.282×10^{-3}
Multitasa (R)Euler (L)Euler	5276.0	$H = 2.0 \times 10^{-3}$ $h = 1.0 \times 10^{-3}$	8.879×10^{-3}	6.282×10^{-3}

Tabla 6.7: Reduciendo el paso de integración para disminuir el error absoluto.

6.2.3. Análisis y discusión de resultados

Entre las características de la arquitectura se encuentra la descomposición en subsistemas. Esta característica permite desarrollar cada uno de los subsistemas por separado, y hasta que se cuente con versiones estables, entonces probar su funcionamiento con los demás subsistemas.

Durante la etapa de verificación se encontraron diferentes problemas en la implementación tanto lógicos como de codificación. Todos ellos fueron corregidos utilizando la metodología presentada.

Una de las ventajas obtenidas al realizar un diseño orientado a objetos es que al utilizar características como *herencia* y *polimorfismo* se reduce el esfuerzo requerido en la etapa de depuración y pruebas ya que de existir un problema sólo es necesario revisar las nuevas clases agregadas, ver [Mil98].

Cambiar el uso de funciones virtuales con herencia permite crear componentes con una interfaz común, esto facilita su uso además de que permite crear nuevos elementos rápidamente al heredar atributos y operaciones de los componentes originales. Por ejemplo, al agregar un nuevo método numérico sólo es necesario redefinir la función **integra**, la cual encapsula al algoritmo numérico. Los elementos como el tamaño del paso h, número de etapas y vectores para coeficientes $\mathbf{a}, \mathbf{b}$ y $\mathbf{c}$, en el caso de los métodos *Runge Kutta*, no tienen que ser redefinidos gracias a la herencia de clases.

Con la ayuda del método de líneas es posible resolver algunas ecuaciones diferenciales parciales. En general, los sistemas de ecuaciones obtenidos al aplicar este método presentan las características para aplicar métodos multitasa, es decir, difer-

entes escalas de tiempo. En el ejemplo mostrado, la matriz $\mathbf{A}$, que representa a un sistema de ecuaciones en la ecuación

$$\mu' = \mathbf{A}\mu + \mathbf{B} \tag{6.2.3}$$

es tridiagonal lo que convierte al sistema de EDO's definido por (6.2.3) en un buen candidato para utilizar los métodos multitasa.

Basado en los resultados obtenidos al utilizar métodos multitasa podemos garantizar que el tiempo de ejecución se reduce de manera considerable respecto al tiempo de un método estándar. En el caso del segundo ejemplo multitasa se obtuvieron reducciones del 47.12 % y 25.31 %. Estos valores dependen de los pasos de integración seleccionados.

Conclusiones

En esta sección se presentan las conclusiones de este trabajo de tesis así como el posible trabajo futuro a desarrollar.

Arquitectura Multitasa

A lo largo de este trabajo se ha presentado el proceso de diseño de una arquitectura flexible para Métodos de Integración Multitasa. Esta arquitectura es capaz de administrar la información producida por métodos numéricos estándares usados bajo una estrategia multitasa. Entre las tareas de administración se encuentra la sincronización de la información producida por dichos métodos.

La arquitectura multitasa propuesta es capaz de resolver sistemas de EDO's que presenten o no diferentes escalas de tiempo. Los sistemas de EDO's deben ser planteados como problemas con valores de inicio. La arquitectura multitasa permite combinar diferentes métodos numéricos estándares y combinarlos para generar métodos multitasa.

La arquitectura multitasa presentada va enfocada a usuarios que deseen realizar pruebas numéricas con métodos multitasa. El usuario puede *crear* sus propios métodos multitasa al combinar los métodos numéricos estándares existentes o los nuevos definidos por él.

Entre las ventajas principales ofrecidas por el paradigma orientado a objetos se encuentra la *herencia*. Esta característica es utilizada en la arquitectura multitasa tanto para definir la interfaz de los métodos numéricos estándares como la de

los modelos de problemas con valores de inicio. Gracias a ella, la tarea de agregar nuevos elementos a la arquitectura se simplifica a *particularizar* las responsabilidades definidas por la interfaz heredada.

Otra de las ventajas ofrecidas por la *herencia*, es que el trabajo y tiempo implicados en la búsqueda y corrección de errores se ve disminuido, debido a que únicamente es necesario buscar y reparar errores en los nuevos elementos agregados.

La arquitectura multitasa puede ser implementada en diferentes plataformas ya que no se utilizan componentes propios de alguna plataforma, esta caracteriza favorece la *portabilidad* de la arquitectura.

Los patrones de diseño favorecen la composición de objetos sobre la herencia de clases, por lo tanto, las ventajas ofrecidas por la composición se presentan a través de las brindadas por los patrones de diseño.

Patrones de diseño y la Arquitectura Multitasa

La arquitectura multitasa ha sido creada con ayuda de patrones de diseño de software orientado a objetos con el fin de obtener un diseño con alta *cohesión* y bajo *acoplamiento* entre sus componentes, es decir, generar un diseño *flexible*. Los patrones de diseño de software incorporan de manera natural las recomendaciones más importantes para el desarrollo de software orientado a objetos, entre las que destacan las siguientes:

- Favorecer a la composición de objetos sobre la herencia de clases.
- Programar una interfaz, no una implementación.
- Identificar lo que cambia y encapsularlo.

La combinación de diferentes patrones de diseño facilita la creación de objetos complejos, por ejemplo, los objetos tipo método multitasa, encargados de la administración y sincronización de la información, son generados a partir del trabajo conjunto de los patrones *Composite* y *Factory Method*. La generación de un método multitasa involucra tres pasos:

1. Crear el *método numérico estándar* encargado de aproximar la solución a la *parte lenta* del sistema.

2. Crear el *método numérico estándar* encargado de aproximar la solución a la *parte rápida* del sistema.

3. Crear el *método numérico multitasa* como una combinación de los métodos *lento* y *rápido*.

La unión de los patrones *Strategy* y *Factory Method* simplifican las tareas de agregación y selección de un método numérico, un nuevo modelo a resolver, un algoritmo de interpolación o una estrategia multitasa.

En lo que respecta a los datos de salida, es decir, la aproximación a la solución, es posible direccionarlos a diferentes dispositivos de salida, por ejemplo, desde una pantalla para su visualización o hasta algún dispositivo externo para su futuro procesamiento. Ésto se logra gracias a la intervención del patrón *Observer*.

Una de las grandes ventajas del uso de patrones es que los usuarios con conocimientos sobre patrones, podrán hacer uso casi inmediato de la arquitectura gracias a la terminología ofrecida de manera natural por los patrones de diseño.

Para mostrar las ventajas ofrecidas por la arquitectura multitasa fue necesario modificar la métrica propuesta por Martin en [Mar03]. Esto se debe a que dicha métrica no diferencia entre las dependencias de paquetes dadas por herencia o por composición, siendo que ésta última es recomendada durante el diseño de software orientado a objetos y los patrones de diseño hacen amplio uso de ella.

Patrones de diseño y desarrollo de software científico

A pesar de que los patrones de diseño han sido creados para el diseño de software de propósito administrativo, el uso de ellos en la tarea de desarrollo de software científico es viable y benéfico ya que permiten, entre otras cosas, crear estructuras complejas de manera sencilla, encapsular algoritmos para ser intercambiados fácil-

mente y establecer interfaces para el manejo de los elementos involucrados en la solución de un problema.

Con ayuda de los patrones de diseño, es posible identificar las partes comunes en los algoritmos utilizados, ésto da lugar a la generación de interfaces comunes para el manejo de los algoritmos empleados.

Entre las aportaciones de este trabajo, se encuentra la identificación de un conjunto de patrones para el diseño de software científico, los detalles en el uso de éstos se han presentado a lo largo de este trabajo por lo que en esta sección nos limitamos únicamente a mencionarlos.

- *Strategy*, útil en la selección de algoritmos numéricos.
- *Factory method*, utilizado para la creación de algoritmos o métodos numéricos.
- *Observer*, ayuda en el manejo de los datos de salida.
- *Template method*, define una plantilla para la ejecución de un algoritmo.
- *Flyweight*, ayuda a eliminar código redundante al encapsular las partes comunes de un algoritmo.
- *Composite*, eficiente para la creación de objetos que presenten una estrategia recursiva.

Desarrollo de software para resolver EDO's

Con base en lo planteado por Gear en [Gea81], sobre el proceso de desarrollo de software para la solución numérica de EDOs con métodos de integración multitasa, este trabajo se sitúa en las etapas de *primeros códigos para producción* y *análisis del software para una clase de problemas*. La etapa siguiente, y última, se refiere a la *generación de software robusto*, sin embargo, y aunque la arquitectura multitasa sea considerada de propósito general, no es posible alcanzar esta última etapa debido a

la carencia de una teoría completa de estabilidad para los Métodos de Integración Multitasa.

La arquitectura multitasa propuesta puede ser utilizada como un *Framework* para desarrollar aplicaciones de simulación de procesos físicos, en particular, aquellos que sean modelados como problemas con valores de inicio. La simulación de este tipo de procesos puede realizarse con métodos numéricos estándares o combinados para crear un método multitasa.

Al considerar a la arquitectura multitasa como un *Framework*, entonces ésta dictará la estructura y organización general de la aplicación específica a desarrollar.

Métodos multitasa para simulación en tiempo real

Las aplicaciones en tiempo real donde la velocidad en cálculos y respuesta de la aplicación son prioritarias pueden hacer uso de esta arquitectura y por tanto de los métodos multitasa. Las comparaciones entre los cálculos obtenidos por la arquitectura multitasa y los métodos adaptativos como *ode45* de *MatLab* y *NDSolve* de *Mathematica* muestran que los errores en exactitud, producidos por la implementación de la arquitectura multitasa en C++, se encuentran dentro de los márgenes de error para aplicaciones de simulación en tiempo real de acuerdo a la norma ANSI / ANS 3.5 - 1985, apartado *4.1 Simulator Capabilities Criteria*; ver [ESW+85].

Trabajo futuro

Como parte del trabajo futuro se encuentra la extensión de la arquitectura multitasa para soportar métodos numéricos adaptativos. Este tipo de métodos se caracteriza por aproximar la solución de un problema con valores de inicio con mayor exactitud ya que el paso de integración es calculado de manera dinámica. El uso de estas técnicas permite adaptar el paso de integración de acuerdo a la dinámica del problema a resolver. Al contar con este tipo de métodos numéricos es posible crear

métodos multitasa adaptativos y por lo tanto obtener mejores aproximaciones a las soluciones.

Para agregar este tipo de métodos a la arquitectura multitasa, se propone el uso del patrón de diseño *Decorator* que, de acuerdo a Gamma *et al.*, permite agregar nuevas responsabilidades a un objeto de manera dinámica, estas responsabilidades serán las relacionadas con el cálculo dinámico del paso de integración.

La arquitectura multitasa presenta la incorporación de la familia de métodos numéricos *Runge Kutta*, además, presenta la opción para agregar nuevas familias, por ejemplo, la familia de métodos numéricos tipo *Adams*.

Al incrementar el número de métodos numéricos se complicará su manejo por lo que se sugiere utilizar el patrón *Abstract Factory* que, de acuerdo a Gamma *et al.*, provee de una interfaz para la creación de familias de objetos relacionados, en este caso, los métodos numéricos que conforman a las familias de métodos.

Utilizando como base esta arquitectura, es posible diseñar una interfaz gráfica amigable para interaccionar con ella, de modo que la creación y uso de métodos multitasa se simplifique aún más. Por ejemplo, para la tarea de creación de un método multitasa se propone *arrastrar y soltar* métodos numéricos dentro de un contenedor, el cual representará a un método multitasa. Este contenedor puede ser agregado a otro contenedor de modo que se genere un método multitasa más complejo.

El objetivo principal de este trabajo de tesis fue el de diseñar una arquitectura multitasa con patrones de diseño orientados a objetos. La arquitectura multitasa presentada cumple con esta característica y puede ser utilizada para aproximar soluciones de problemas con valores de inicio que presenten o no diferentes escalas de tiempo donde no se requiera demasiada exactitud. Por ejemplo, aplicaciones en tiempo real o simuladores con un gran número de ecuaciones donde el objetivo de la simulación sea observar el comportamiento cualitativo de la solución.

Con el diseño de esta arquitectura se da evidencia de la eficiencia del uso de los patrones de diseño de software orientados a objetos para el diseño y desarrollo de software de tipo científico.

Apéndice A

Aproximación mediante diferencias finitas

En este apartado se presentan de manera breve los temas relacionados con la aproximación de soluciones a EDO's mediante diferencias finitas.

A.1. Métodos de diferencias finitas y error

Un método de diferencias finitas sustituye las derivadas en una ecuación diferencial por aproximaciones mediante diferencias finitas, este procedimiento da lugar a resolver un sistema de ecuaciones algebraicas en lugar de las ecuaciones diferenciales originales.

Sea $\mu(x)$ una función suave, es decir, existe $\mu^{(i)}(x), i = 0, 1, \ldots$ y cada derivada es una función acotada sobre un intervalo que contiene a un punto de interés $\bar{x}$.

¿Cómo podemos aproximar $\mu'(x)$ usando una fórmula de diferencias finitas? Recordemos la definición de la derivada:

$$f'(\bar{x}) = \lim_{h \to 0} \frac{f(\bar{x} + h) - f(\bar{x})}{h} \tag{A.1.1}$$

Utilizando la ecuación (A.1.1) podemos definir la aproximación mediante diferencias

finitas $D_+\mu(\bar{x})$, para algún h pequeño, como:

$$D_+\mu(\bar{x}) = \frac{\mu(\bar{x}+h) - \mu(\bar{x})}{h} \tag{A.1.2}$$

La ecuación (A.1.2) aproxima a la derivada por el lado derecho, es decir, para $x \geq \bar{x}$. Se define la aproximación con diferencias finitas $D_-\mu(\bar{x})$, para algún h pequeño, como:

$$D_-\mu(\bar{x}) = \frac{\mu(\bar{x}) - \mu(\bar{x}-h)}{h} \tag{A.1.3}$$

La ecuación (A.1.3) aproxima a la derivada por el lado izquierdo, es decir, para $x \leq \bar{x}$. Las aproximaciones $D_+\mu(\bar{x})$ y $D_-\mu(\bar{x})$ son conocidas como aproximaciones de *un solo lado* y la exactitud en la aproximación a $\mu'(\bar{x})$ es de *primer orden*, es decir, el tamaño del error respecto a $\mu'(\bar{x})$ es proporcional a h.

Es posible obtener otra fórmula utilizando las ecuaciones (A.1.2) y (A.1.3). La aproximación $D_0\mu(\bar{x})$ es conocida como *aproximación centrada* y se define como:

$$D_0\mu(\bar{x}) = \frac{1}{2}(D_+\mu(\bar{x}) + D_-\mu(\bar{x})) = \frac{\mu(\bar{x}+h) - \mu(\bar{x}-h)}{2h} \tag{A.1.4}$$

La ecuación (A.1.4) ofrece una mejor aproximación para $\mu'(\bar{x})$, presenta una exactitud de *segundo orden* y por lo tanto el error es proporcional a h^2. Un aproximación de tercer orden, donde el error es proporcional a h^3 es:

$$D_3\mu(\bar{x}) = \frac{1}{6h}(2\mu(\bar{x}+h) + 3\mu(\bar{x}) - 6\mu(\bar{x}-h) + \mu(\bar{x}-2h)) \tag{A.1.5}$$

Generalmente, es posible obtener mejores aproximaciones haciendo combinaciones de los métodos presentados, sin embargo, este tema queda fuera del alcance de este trabajo. Para el lector interesado se sugiere revisar los trabajos de Lambert [Lam73] y Leveque [LeV07].

Definimos el *error* como la diferencia entre la aproximación a la derivada $D\mu(\bar{x})$ y el valor real $\mu'(\bar{x})$, es decir:

$$Error = D\mu(\bar{x}) - \mu'(\bar{x}) \tag{A.1.6}$$

En lo mencionado respecto a los ordenes de las ecuaciones podemos notar que el error se comporta como una potencia de h, es decir:

$$E(h) = Ch^p \tag{A.1.7}$$

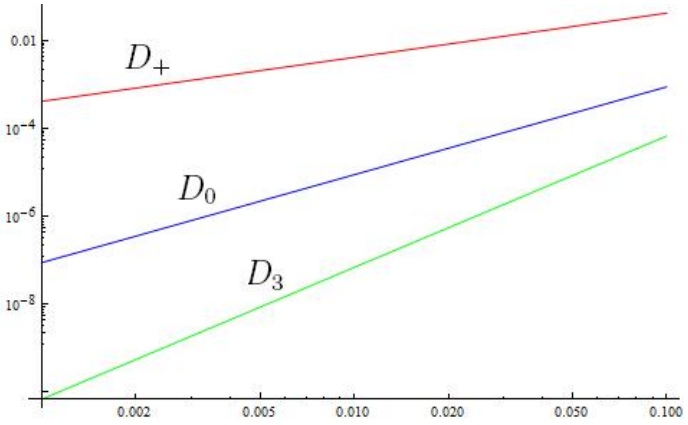


Figura A.1: Error para las $D\mu(\bar{x})$ contra h en escala $\log - \log$.

A.2. Ejemplo

Sea $\mu(x) = \sin(x)$ y $\bar{x} = 1$. Queremos aproximar $\mu'(1) = \cos(1) = 0.5403023$ En la figura A.1 se presenta el comportamiento del error para las aproximaciones $D_+\mu(\bar{x}), D_0\mu(\bar{x})$ y $D_3\mu(\bar{x})$ respecto a h. Por medio de manipulación algebraica podemos llevar a la ecuación (A.1.7) a la forma:

$$\log E(h) \approx \log C + p \log h \tag{A.2.1}$$

El error, de acuerdo a la figura A.1, se comporta de acuerdo a la pendiente dada por p en la ecuación (A.2.1).

A.3. Error de truncamiento

Recordemos la expansión en *series de Taylor* para la función $f(x)$ alrededor de un punto x_0.

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + ... \tag{A.3.1}$$

Si hacemos un cambio de variable $\Delta x = x - x_0$, podemos redefinir la ecuación (A.3.1) de la siguiente manera:

$$f(x + x_0) = f(x_0) + f'(x_0)\Delta x + \frac{f''(x_0)}{2!}\Delta x^2 + \frac{f'''(x_0)}{3!}\Delta x^3 + ... \tag{A.3.2}$$

El análisis del error en la aproximación mediante diferencias finitas consiste en expandir los valores de las funciones en series de Taylor alrededor del punto $\bar{x}$. Estas expansiones son validas ya que la función μ es suave. La expansión para los puntos $\mu(\bar{x}+h)$ y $\mu(\bar{x}-h)$ es la siguiente:

$$\mu(\bar{x}+h) = \mu(\bar{x}) + \mu'(\bar{x})h + \frac{1}{2}h^2\mu''(\bar{x}) + \frac{1}{6}h^3\mu'''(\bar{x}) + O(h^4) \tag{A.3.3}$$

$$\mu(\bar{x}-h) = \mu(\bar{x}) - \mu'(\bar{x})h + \frac{1}{2}h^2\mu''(\bar{x}) - \frac{1}{6}h^3\mu'''(\bar{x}) + O(h^4) \tag{A.3.4}$$

Podemos calcular el error para la aproximación $D_+\mu(\bar{x})$ utilizando la igualdad (A.3.3), éste se presenta a continuación.

$$\begin{aligned} D_+\mu(\bar{x}) &= \frac{\mu(\bar{x}+h) - \mu(\bar{x})}{h} \\ &= \frac{\mu(\bar{x}) + \mu'(\bar{x})h + \frac{1}{2}h^2\mu''(\bar{x}) + \frac{1}{6}h^3\mu'''(\bar{x}) + O(h^4) - \mu(\bar{x})}{h} \\ &= \mu'(\bar{x}) + \frac{1}{2}h\mu''(\bar{x}) + \frac{1}{6}h^2\mu'''(\bar{x}) + O(h^3) \end{aligned} \tag{A.3.5}$$

Los elementos $\mu'(\bar{x})$ y $\mu''(\bar{x})$ son constantes al ser dependientes únicamente de $\bar{x}$ e independientes de h. Para un valor de h lo suficientemente pequeño, el error será dominado por el término $\frac{1}{2}h\mu''(\bar{x})$. Los otros elementos son insignificantes respecto a éste último. Es posible realizar un análisis similar para la aproximación del error de $D_-\mu(\bar{x})$.

Para el caso de $D_0\mu(\bar{x})$ es necesario calcular la expansión para $\mu(\bar{x}+h)-\mu(\bar{x}-h)$, combinando los resultados de (A.3.3) y (A.3.4) tenemos:

$$\mu(\bar{x}+h) - \mu(\bar{x}-h) = 2h\mu'(\bar{x}) + \frac{1}{3}h^3\mu'''(\bar{x}) + O(h^5) \tag{A.3.6}$$

Aplicando este resultado a $D_0\mu(\bar{x})$ obtenemos:

$$\mu'(\bar{x}) + \frac{1}{6}h^2\mu'''(\bar{x}) + O(h^4) \tag{A.3.7}$$

El error se encuentra dominado por $\frac{1}{6}h^2$ por lo que se confirma la exactitud de segundo orden para esta aproximación.

A.4. Error local y global

Aproximaremos la solución a una ecuación diferencial ordinaria de segundo orden a partir de un método de diferencias finitas.

$$\mu''(x) = f(x) \quad 0 < x < 1 \tag{A.4.1}$$

Con las siguientes condiciones de frontera

$$\mu(0) = \alpha \quad \mu(1) = \beta \tag{A.4.2}$$

A través de este problema se explicarán conceptos esenciales tales como *error local*, *error global* y la importancia de la *estabilidad* para relacionar ambos errores.

El objetivo del método de diferencias finitas es calcular una función *malla* con valores en los puntos $U_0, U_1, \ldots, U_m, U_{m+1}$ de modo que U_j sea la aproximación a la solución $\mu(x_j)$. Denotamos a $x_j = jh$ donde $h = 1/(m+1)$ es el ancho de la malla, es decir, la distancia entre los puntos de la función encontrada. Sabemos que $U_0 = \alpha$ y $U_{m+1} = \beta$ por las condiciones de frontera dadas, por lo tanto tenemos m puntos que calcular.

Al reemplazar $\mu''(x)$ de la ecuación (A.4.1) por una fórmula de diferencias centradas obtenemos:

$$D^2 U_j = \frac{1}{h^2}(U_{j-1} - 2U_j + U_{j+1}) \tag{A.4.3}$$

Formamos un sistema de ecuaciones algebraicas para $j = 1, 2, \ldots, m$.

$$\frac{1}{h^2}(U_{j-1} - 2U_j + U_{j+1}) = f(x_j) \tag{A.4.4}$$

Obtenemos un sistema de m variables que puede ser reescrito como:

$$\mathbf{AU} = \mathbf{F} \tag{A.4.5}$$

donde U representa un vector con m elementos, $U = [U_1, U_2, \ldots, U_m]^T$ y:

$$\text{(A.4.6)} \qquad \mathbf{A} = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 \end{bmatrix} \qquad \mathbf{F} = \begin{bmatrix} f(x_1) - \alpha/h^2 \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{m-1}) \\ f(x_m - \beta/h^2) \end{bmatrix}$$

Ya que U_j aproxima a $\mu(x_j)$, el error en ese punto esta dado por $U_j - \mu(x_j)$. Si denotamos a $\hat{U}$ como el vector de las soluciones verdaderas, $\hat{U} = [\mu(x_1), \ldots, \mu(x_m)]^T$ entonces es posible definir el error como:

$$\text{(A.4.7)} \qquad \mathbf{E} = \mathbf{U} - \hat{\mathbf{U}}$$

Para obtener una cota en el error y pueda dar evidencia que se comporta como $O(h^2)$ conforme $h \to 0$ utilizaremos una *norma*, en este la norma infinita definida por $||\mathbf{E}||_\infty$.

$$\text{(A.4.8)} \qquad ||\mathbf{E}||_\infty = \max_{1 \le j \le m} |E_j| = \max_{1 \le j \le m} |U_j - \mu(x_j)|$$

Al acotar $||\mathbf{E}||_\infty \in O(h^2)$ entonces todos los valores del vector estarán también acotados.

¿Cómo estimar el error cometido por la aproximación (A.4.3)? Utilizaremos una técnica básica en el análisis de métodos de diferencias finitas.

1. Calcular el *error de truncamiento local* del método.
2. Utilizar el concepto de *estabilidad* para mostrar que el *error global* puede ser acotado en términos del error local.

A.4.1. Error de truncamiento local

El cálculo de este error se obtiene al reemplazar las U_j por la solución verdadera $\mu(x_j)$ en la fórmula de diferencias (A.4.3). Desde luego, la aproximación por diferencias no satisface exactamente a la solución por lo que la diferencia obtenida será el

error de truncamiento local, y lo denotaremos por τ_j.

$$\tau_j = \frac{1}{h^2}(\mu(x_{j-1}) - 2\mu(x_j) + \mu(x_{j+1})) - f(x_j) \tag{A.4.9}$$

Al expandir por series de Taylor los elementos de τ_j obtenemos:

$$\tau_j = \left[\mu''(x_j) + \frac{1}{12}h^2\mu''''(x_j) + O(h^4)\right] - f(x_j) \tag{A.4.10}$$

Al sustituir $\mu''(x_j)$ por (A.4.1) obtenemos:

$$\tau_j = \left[\frac{1}{12}h^2\mu''''(x_j) + O(h^4)\right] \tag{A.4.11}$$

Al igual que en diferencias finitas, $\mu''''(x_j)$ es independiente de h por lo que se toma constante de modo que $\tau_j \in O(h^2)$ conforme $h \to 0$. A manera de vectores, definimos a $\boldsymbol{\tau}$ como un vector con elementos τ_j, y a $\mathbf{F}$ como otro vector con elementos $f(x_j)$ entonces

$$\boldsymbol{\tau} = \mathbf{A}\hat{\mathbf{U}} - \mathbf{F} \tag{A.4.12}$$

La solución verdadera sera entonces

$$\mathbf{A}\hat{\mathbf{U}} = \boldsymbol{\tau} + \mathbf{F} \tag{A.4.13}$$

A.4.2. Error global

Para obtener una relación entre el error local $\boldsymbol{\tau}$ y el error global $\mathbf{E} = \mathbf{U} - \hat{\mathbf{U}}$, utilizaremos la aproximación (A.4.5) y la solución verdadera (A.4.13).

$$\begin{aligned}
\mathbf{A}\mathbf{U} - \mathbf{A}\hat{\mathbf{U}} &= \mathbf{F} - (\mathbf{F} + \boldsymbol{\tau}) \\
\mathbf{A}(\mathbf{U} - \hat{\mathbf{U}}) &= -\boldsymbol{\tau} \\
\mathbf{A}\mathbf{E} &= -\boldsymbol{\tau}
\end{aligned}$$

Formamos un sistema de ecuaciones a partir de la representación matricial anterior para obtener

$$\frac{1}{h^2}(E_{j-1} - 2E_j + E_{j+1}) = -\tau_j \quad j = 1, 2, \ldots, m \tag{A.4.14}$$

El cuál cumple con las condiciones de frontera

$$E_0 = E_{m+1} = 0 \tag{A.4.15}$$

debido a que se utilizan los valores conocidos para $U_0 = \alpha$ y $U_{m+1} = \beta$.

Ya que el error se comporta como un conjunto de ecuaciones similares a las definidas para U, pero con $-\boldsymbol{\tau}$ en lugar de $\mathbf{F}$, se espera que el error se comporte de manera similar a $\boldsymbol{\tau}$. De (A.4.14) obtenemos

$$e''(x) = -\tau(x) \quad 0 \leq x \leq 1 \tag{A.4.16}$$

con las condiciones de frontera

$$e(0) = 0 \quad e(1) = 0 \tag{A.4.17}$$

Sabemos por (A.4.11) que $\tau(x) \approx \frac{1}{12}h^2\mu''''(x)$. Al integrar $e''(x)$ dos veces se obtiene

$$e(x) \approx -\frac{1}{2}h^2\mu(x) + \frac{1}{2}h^2(\mu''(0) + x(\mu''(1) - \mu''(0))) \tag{A.4.18}$$

por lo que el error se comporta como $O(h^2)$.

A.5. Estabilidad, consistencia y convergencia

A.5.1. Estabilidad

Retomando el sistema $\mathbf{A}^h\mathbf{E}^h = \boldsymbol{\tau}^h$, donde h no es una potencia sino que indica el sistema obtenido para una h particular, sabemos que la matriz $\mathbf{A}^h$ es de tamaño $m \times m$ y éste está dado por el valor de $h = 1/(m+1)$, de modo que la dimensión de $\mathbf{A}^h$ incrementa conforme $h \to 0$.

Supongamos que existe $(\mathbf{A}^h)^{-1}$, la matriz inversa de $\mathbf{A}^h$, entonces al resolver el sistema mencionado se tiene:

$$\mathbf{E}^h = -((\mathbf{A}^h)^{-1}\boldsymbol{\tau}^h) \tag{A.5.1}$$

Al aplicar alguna norma se obtiene

$$||\mathbf{E}^h|| = ||-((\mathbf{A}^h)^{-1}\boldsymbol{\tau}^h)|| \leq ||((\mathbf{A}^h)^{-1}||||\boldsymbol{\tau}^h)|| \tag{A.5.2}$$

Sabemos que $||\boldsymbol{\tau}^h|| \in O(h^2)$ y se espera que $||\mathbf{E}^h||$ presente el mismo comportamiento. Es necesario que $||((\mathbf{A}^h)^{-1}||$ se encuentre acotada por alguna constantes independiente de h, es decir, $||((\mathbf{A}^h)^{-1}|| \leq C$ para h suficientemente pequeña.

Si la condición mencionada se cumple, entonces:

$$||\mathbf{E}^h|| \leq C||\boldsymbol{\tau}^h|| \tag{A.5.3}$$

Por lo tanto $||\mathbf{E}^h|| \to 0$ cuando $||\boldsymbol{\tau}^h \to 0||$.

Definición de *estabilidad* para problemas lineales con valores en la frontera.

Suponiendo que un método de diferencias finitas para un problema con valores en la frontera resulta en una secuencia de ecuaciones de la forma $\mathbf{A}^h\mathbf{U}^h = \mathbf{F}^h$. Se dice que el método es *estable* si $(\mathbf{A}^h)^{-1}$ existe para toda h suficientemente pequeña $(h < h_0)$ y si existe una constante C independiente de h tal que:

$$||((\mathbf{A}^h)^{-1}|| \leq C, \quad \text{para toda } h < h_0 \tag{A.5.4}$$

A.5.2. Consistencia

Un método es *consistente* con una ecuación diferencial y sus condiciones de frontera si

$$||\boldsymbol{\tau}^h|| \to 0 \quad \text{cuando } h \to 0 \tag{A.5.5}$$

Es decir, el error local tiende a cero conforme h se hace pequeña. Generalmente, se cumple que $||\boldsymbol{\tau}^h|| \in O(h^p)$ para alguna p entera y $p > 0$.

A.5.3. Convergencia

Un método *converge* si

$$||\mathbf{E}|| \to 0, \quad \text{cuando } h \to 0 \tag{A.5.6}$$

Esto nos lleva al *teorema fundamental de los métodos de diferencias finitas* el cual se presenta a continuación:

$$\textit{consistencia} + \textit{estabilidad} \Rightarrow \textit{convergencia} \tag{A.5.7}$$

Este teorema se obtiene de la interpretación de la ecuación $||\mathbf{E}|| \leq ||\mathbf{A}^{-1}||||\boldsymbol{\tau}|| \leq C||\boldsymbol{\tau}|| \to 0$ conforme $h \to 0$.

En resumen, si el *error de truncamiento local (ETL)* se comporta como $O(h^p)$ y al utilizar *estabilidad* podemos acotar el *error global (EG)* con ayuda del *error de truncamiento local (ETL)*, entonces el método *converge*.

(A.5.8) $$ETL \in O(h^p) + \text{estabilidad} \Rightarrow EG \in O(h^p)$$

Índice de tablas

Índice de figuras

Bibliografía

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, third edition edition, 1999.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Oxford University Press, New York, 1977.

[Ale79] Christopher Alexander. *The timeless way of building*. Oxford University Press, New York, 1979.

[AMB97] Erlend Arge and Are Magnus Bruaset. Object-oriented numerics. *Numerical Methods and Software Tools in Industrial Mathematics*, pages 7–26, 1997.

[And93] Jan Frederick Andrus. A runge-kutta method with stepsize control for separated systems of first-order odes. *Appl. Math. Comput.*, 59(2-3):193–214, 1993.

[Bir93] C. R. Birchenhall. Matclass : A matrix class for c++. In *Computational Techniques for Econometrics and Economic Analysis*, pages 151–172. Kluwer, 1993.

[BL97] Are Magnus Bruaset and Hans Petter Langtangen. Object-oriented design of preconditioned iterative methods in diffpack. *ACM Trans. Math. Softw.*, 23(1):50–80, 1997.

[Bli02] Charles Blilie. Patterns in scientific software: An introduction. *Computing in Science and Engineering*, 4(3):48–53, 2002.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[Bun06] Peter Bunus. A simulation and decision framework for selection of numerical solvers in. In *ANSS '06: Proceedings of the 39th annual Symposium on Simulation*, pages 178–187, Washington, DC, USA, 2006. IEEE Computer Society.

[CC08] Sorana Cimpan and Vincent Couturier. Can styles improve architectural pattern reuse? In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 263–266, Washington, DC, USA, 2008. IEEE Computer Society.

[Con95] Darrel J. Conway. A c++ integrator class. *Dr. Dobb's Journal*, pages 52–58, December 1995.

[Coo00] James W. Cooper. *Java design patterns: a tutorial*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CVRG04] E. Cuellar Vazquez and G. Rodríguez Gómez. Software patterns identification to implement dynamic simulation systems. In *Proceedings of the Summer Computer Simulation Conference*, pages 349–354, San Jose, California, USA, 2004.

[CWG84] D. R. Wells C. W. Gear. Multirate linear multistep methods. *BIT Numerical Mathemetics*, 24:484–502, 1984.

[EL97] Ch. Engstler and Ch. Lubich. Multirate extrapolation methods for differential equations with different time scales. *Computing*, 58(2):173–185, 1997.

[ESW+85] N. S. Elliot, P. J. Swanson, G. H. Wanner, J. M. Black, R. J. Bruno, T. Dennis, R. A. Felker, W. B. Geise, F. C. Grams, J. H. Harris, S. M. Halverson, N. K. Hunemuller, T. E. James, R. A. Johnson, T. J. Kerwin, J. W. Lehner, G. P. Mecchi, M. F. Reisinger, S. C. Roessner, J. P. Sursock, J. A. Wachtel, and K. P. Welchel. Nuclear power plants simulators for use in operator training and examination. Technical Report ANSI ANS 3.5 - 1985, American Nuclear Society, 555 North Kensington Avenue La Grange Park, Illinois 60526 U.S.A., 1985.

[Gea81] C. W. Gear. Numerical solution of ordinary differential equations: Is there anything left to do? *SIAM Review*, 23(1):10–24, 1981.

[Gea82] C. W. Gear. Stiff sofware: What do we have and what do we need? Technical Report UIUCDCS-R-82-1109, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue Urbana, November 1982.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[Hin83] A. C. Hindmarsh. Odepack, a systematized collection of ode solvers. *IMACS Transactions on Scientific Computation*, 1:55–64, 1983.

[Hol06] Steve Holzner. *Design patterns for dummies®*. John Wiley & Sons, Inc., New York, NY, USA, 2006.

[HSCG03] Jarrod Hollingworth, Bob Swart, Mark Cashman, and Paul Gustavson. *Borland C++ Builder 6 Developer's Guide*. Sams, Indianapolis, IN, USA, 2003.

[KEM93] M. S. Kamel, W. H. Enright, and K. S. Ma. Odexpert: an expert system to select numerical solvers for initial value ode systems. *ACM Trans. Math. Softw.*, 19(1):44–62, 1993.

[KM99] Christopher E. Kees and Cass T. Miller. C++ implementations of numerical methods for solving differential-algebraic equations: design and optimization considerations. *ACM Trans. Math. Softw.*, 25(4):377–403, 1999.

[KR99] A. Kværnø and P. Rentrop. Low order multirate runge-kutta methods in electric circuit simulation. Submitted for publication, 1999.

[Kvæ00] A. Kværnø. Stability of multirate runge-kutta schemes. *Int. J. Differ. Equ. Appl.*, 1(1):97–105, 2000.

[Lam73] J. D. Lambert. *Computational methods in ordinary differential equations.* Wiley, London, New York, 1973.

[Lar01] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition).* Prentice Hall PTR, 2 edition, July 2001.

[LeV07] Randall LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[LM01] Hans Petter Langtangen and Otto Munthe. Solving systems of partial differential equations using object-oriented programming techniques with coupled heat and fluid flow as example. *ACM Trans. Math. Softw.*, 27(1):1–26, 2001.

[Mar03] Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices.* Prentice Hall, 2003.

[MCH+04] Thierry Matthey, Trevor Cickovski, Scott Hampton, Alice Ko, Qun Ma, Matthew Nyerges, Troy Raeder, Thomas Slabach, and Jesús A. Izaguirre. Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Trans. Math. Softw.*, 30(3):237–265, 2004.

[Mil98] M. Milde. Ode++ a class library for ordinary differential equations. Technical Report 07740 Jena, Fakultat fur Mathematik und Informatik Friedrich-Schiller-Universitat Jena, Ernst-abbe-Platz 1-4, Germany, March 1998.

[Mod05] Modelica Association. *Modelica - A Unified Object-Oriented Languaje for Physical Systems Modeling - Languaje Specification Version 2.2*, February, 2005.

[MSU00] Gou Masuda, Norihiro Sakamoto, and Kazuo Ushijima. Redesigning of an existing software using design patterns. *International Symposium on Principles of Software Evolution*, 0:165–169, 2000.

[Ouy02] Y Ouyang. Explaining design patterns through one application. *Frontiers in Education, 2002. FIE 2002. 32nd Annual Publication*, 3, 2002.

[Pet05] Dana Petcu. Software issues in solving initial value problems for ordinary differential equations. *Creative Mathematics*, 13:97–110, 2005.

[PNB89] Alan C. Hindmarsh Peter N. Brown, Geroge D. Byrne. Vode, a variable-coefficient ode solver. *SIAM Journal on Scientific and Statistical Computing*, 10(5):1038–1051, 1989.

[RG98] Gustavo Rodríguez Gómez. Métodos de integración multitasa para simulación de procesos en tiempo real. Master's thesis, Universidad Nacional Autónoma de México, México, D.F., 1998.

[RG02] Gustavo Rodríguez Gómez. *Absolute stability analysis of semi-implicit multirate linear multistep methods*. PhD thesis, Instituto Nacional de Astrofísica, Óptica y Electrónica, Tonantzintla, Puebla, México, 2002.

[RGM04] Gustavo Rodríguez, Pedro González, and Jorge Martínez. Computing general companion matrices and stability regions of multirate methods. *International Journal for Numerical Methods in Engineering*, 61(2):255–273, 2004.

[RH93] Krishnan Radhakrishnan and Alan C. Hindmarsh. Description and use of lsode, the livermore solver for ordinary differential equations. Technical report, NASA, 1993.

[SC00] L. F. Shampine and Robert M. Corless. Initial value problems for odes in problem solving environments. *J. Comput. Appl. Math.*, 125(1-2):31–40, 2000.

[SG75] L. F. Shampine and M. K. Gordon. *Computer Solution of Ordinary Diferential Equations: Initial Value Problems*. Freeman and Company, California, USA, third edition edition, 1975.

[SH99] Harris A. Schilling and Sandra L. Harris. *Applied Numerical Methods for Engineers Using MATLAB*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1999.

[Ske84] Stig Skelboe. Multirate integration methods: The numerical solution of stiff systems of ordinary differential equations by multirate integration methods. Technical Report ECR-150, ElektronicCentralen, Venlighedsvej 4 DK-2970 Horsholm Denmark, March 1984.

[Sán79] David A. Sánchez. *Ordinary Differential Equations and Stability Theory: An Introduction*. General Publishing Company, 1979.

[Som95] Ian Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[SR97] Lawrence F. Shampine and Mark W. Reichelt. The matlab ode suite. *SIAM Journal Science Computing*, 18(1):1–22, 1997.

[ST04] Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design (2nd Edition) (Software Patterns Series)*. Addison-Wesley Professional, 2004.

[VJ97] Todd L. Veldhuizen and M. Ed Jernigan. Will c++ be faster than fortran? In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 49–56, London, UK, 1997. Springer-Verlag.

[Wel82] Daniel Raymond Wells. *Multirate linear multistep methods for the solution of systems of ordinary differential equations*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982.