API para el desarrollo de algoritmos interactivos en ingeniería del software basada en búsqueda

Aurora Ramírez, José Raúl Romero y Sebastián Ventura

Dpto. de Informática y Análisis Numérico, Universidad de Córdoba
{aramirez,jrromero,sventura}@uco.es

Resumen La experiencia y la intuición son factores clave a la hora de dar solución a los complejos problemas que plantea la ingeniería del software. Sin embargo, este tipo de criterios no suelen ser considerados cuando su resolución se aborda por medio de técnicas de búsqueda automática. La ingeniería del software basada en búsqueda (SBSE) no puede ni debe obviar la opinión del ingeniero, razón por la que cada vez es más frecuente encontrar propuestas que le invitan a participar en el proceso. Diseñar e implementar un mecanismo de interacción efectivo, a la par que atractivo para el ingeniero, puede resultar complejo. Por ello, este trabajo presenta una API para dar soporte al desarrollo de algoritmos interactivos en SBSE. En base a los enfoques interactivos actuales en SBSE, esta API expone cuáles son los requisitos propios de la interactividad que deben programarse como, por ejemplo, la forma de evaluar las soluciones y las acciones que el ingeniero puede realizar sobre ellas.

Keywords: Ingeniería del software basada en búsqueda, interactividad, *Application Programming Interface*

1. Introducción

En el proceso de desarrollo software, los ingenieros software se enfrentan a tareas que requieren amplios conocimientos sobre metodologías, tecnologías y dominios de aplicación. Además, también necesitan creatividad, experiencia e intuición a la hora de plantear las soluciones software más adecuadas en base a multitud de criterios. Obsérvese que no solo se trata de habilidades difíciles de adquirir, sino también de expresar formalmente y, por ende, transmitir a otros. Con el objetivo de aliviar la labor de los ingenieros software, la ingeniería del software basada en búsqueda (SBSE, *Search Based Software Engineering*) [1] propone aplicar técnicas de búsqueda a problemas que son realizados tradicionalmente por ingenieros software, como pueden ser la selección de requisitos, el diseño software o la generación de casos de prueba.

En sus inicios, SBSE se centró principalmente en abordar estas tareas de forma completamente automática, mediante algoritmos como las metaheurísticas [2]. Estas técnicas se basan en explorar, de forma iterativa y no exhaustiva, un espacio de soluciones candidatas, cuya calidad (denominada *fitness*) debe ser evaluada por una o más funciones numéricas. En la práctica, dichas funciones

se han venido correspondiendo con medidas software (a veces adaptadas o agregadas), ampliamente utilizadas en las distintas fases del proceso software para controlar aspectos de calidad. Por ejemplo, la selección de requisitos se suele basar en optimizar el coste frente al beneficio esperado, mientras que la cobertura del código es el principal objetivo en la generación de casos de prueba. Sin embargo, existen otras muchas tareas, especialmente las más creativas, para las que las medidas software se antojan insuficientes [3]. En estos casos, la búsqueda debe basarse en criterios más subjetivos y de naturaleza cualitativa [4]. Es por ello que cada vez más investigadores en SBSE están explorando el uso de técnicas interactivas [5]. Esta nueva vertiente, recientemente acuñada como SBSE interactiva (iSBSE) [6], tiene como principal objetivo la generación de soluciones más acordes con las expectativas del ingeniero.

La inclusión de interactividad en SBSE implica el estudio de aspectos como el rol del humano, las acciones que se le va a permitir realizar, cada cuánto se va a interactuar o cómo su opinión va a ser integrada en el proceso de búsqueda [6]. Es más, el diseño de un modelo de interacción puede estar condicionado por la propia naturaleza del problema, así como por factores inherentes a la búsqueda interactiva como son la incertidumbre o la fatiga [7]. Desde el punto de vista de su implementación, cabe señalar que actualmente no existen librerías o frameworks genéricos para el desarrollo de algoritmos interactivos, menos aún en el contexto concreto de iSBSE. Esto dificulta el desarrollo de nuevas propuestas debido principalmente a la falta de una capa de abstracción que permita aislar el modelo de interacción del algoritmo de búsqueda concreto. Dar soporte a la programación de este tipo de algoritmos implica conocer las necesidades específicas de la interactividad en SBSE a fin de ofrecer una contribución significativa al área. Disponer de un marco de desarrollo común también facilitaría la reutilización de componentes y su integración en herramientas especialmente dirigidas a ingenieros software.

En este sentido, este trabajo trata de dar solución a la problemática que rodea al diseño e implementación de propuestas interactivas en SBSE. En primer lugar, se explican los cinco aspectos fundamentales que deben tenerse en cuenta en el diseño de la interactividad [6]: el tipo de algoritmo, la naturaleza de la opinión del usuario, el mecanismo de evaluación de soluciones, el control de la intervención del usuario y la influencia de su opinión en el proceso. En base a ellos, se ha diseñado una API Java para dar soporte programático a la investigación en iSBSE. Esta API proporciona las clases e interfaces necesarias para codificar diferentes mecanismos de interacción con independencia del algoritmo de búsqueda utilizado. Actualmente, la API soporta un conjunto de acciones genéricas a realizar por el humano y diferentes mecanismos para la evaluación de la calidad de las soluciones. Con este trabajo se ponen de manifiesto las necesidades y requisitos que deben plantearse en iSBSE, proporcionando una solución programática que pueda ser de utilidad para futuros investigadores.

El resto del artículo se estructura como sigue. La sección 2 presenta una introducción a la ingeniería del software basada en búsqueda interactiva. A continuación, la sección 3 cubre el proceso de diseño de un algoritmo interactivo

para SBSE. En base a ello, la sección 4 contiene la especificación de la API de desarrollo, mientras que la sección 5 muestra un ejemplo de su uso. Finalmente, las conclusiones y las líneas de trabajo futuro son abordadas en la sección 6.

2. Ingeniería del software basada en búsqueda interactiva

La ingeniería del software basada en búsqueda interactiva aglutina a cualquier propuesta SBSE en la que el humano interviene activamente en el proceso de búsqueda [6]. La interacción consiste en la interrupción del algoritmo, de forma puntual o periódica, con el objetivo de mostrarle al humano resultados intermedios. A continuación, el humano proporcionará al algoritmo cierto *feedback*, el cual debe producir un efecto en las siguientes iteraciones del algoritmo. La aparición de esta subárea dentro de SBSE se debe a los siguientes factores:

- Dificultad a la hora de formular una representación adecuada y completa del problema de ingeniería del software.
- Imposibilidad de definir una función de *fitness* cuantitativa para evaluar la calidad de las soluciones.
- Reticencia del humano a aceptar soluciones generadas automáticamente.

La reciente revisión sistemática del área [6] muestra un creciente interés en la aplicación de técnicas de búsqueda interactiva. Aún tratándose de un área emergente, es posible encontrar propuestas iSBSE en la mayoría de fases del ciclo de vida software. La fase de análisis y diseño sobresale como la más estudiada, posiblemente debido a su carácter creativo inherentemente humano, con trabajos como el análisis orientado a objetos [8]. Tareas relacionadas con el mantenimiento, como la refactorización de código [9], también han sido abordadas desde una perspectiva interactiva. En menor medida, las áreas de requisitos, centrada en el problema de la siguiente versión [10], y pruebas, con trabajos para generar sus entradas [11], han aplicado técnicas propias de iSBSE.

Entre las técnicas aplicadas destacan los algoritmos evolutivos, probablemente debido a su popularidad dentro de SBSE. Gracias a ello y a su eficacia, la interactividad ha sido aplicada sobre una gran variedad de problemas y con diferentes propósitos, desde la evaluación hasta la modificación de las soluciones candidatas. Más recientemente, se ha comenzado a explorar el uso de otras técnicas como la optimización de colonias de hormigas [8], cuyas versiones interactivas se centran principalmente en la fase de evaluación. También existe cierto interés en la combinación de iSBSE con técnicas de aprendizaje automático capaces de aprender un modelo de evaluación en base a la opinión del humano, con el objetivo de sustituirle y reducir así la fatiga [9,10].

Respecto a las características propias de los modelos interactivos, estos suelen estar encaminados a completar la formulación del problema o guiar al algoritmo hacia soluciones que cumplan ciertas preferencias. Esto contrasta con la concepción original de las metaheurísticas interactivas, donde el humano reemplazaba a la función de *fitness*. Así, todo el peso de la evaluación recae sobre el humano, siendo esta una tarea tediosa y que difícilmente se puede realizar de manera

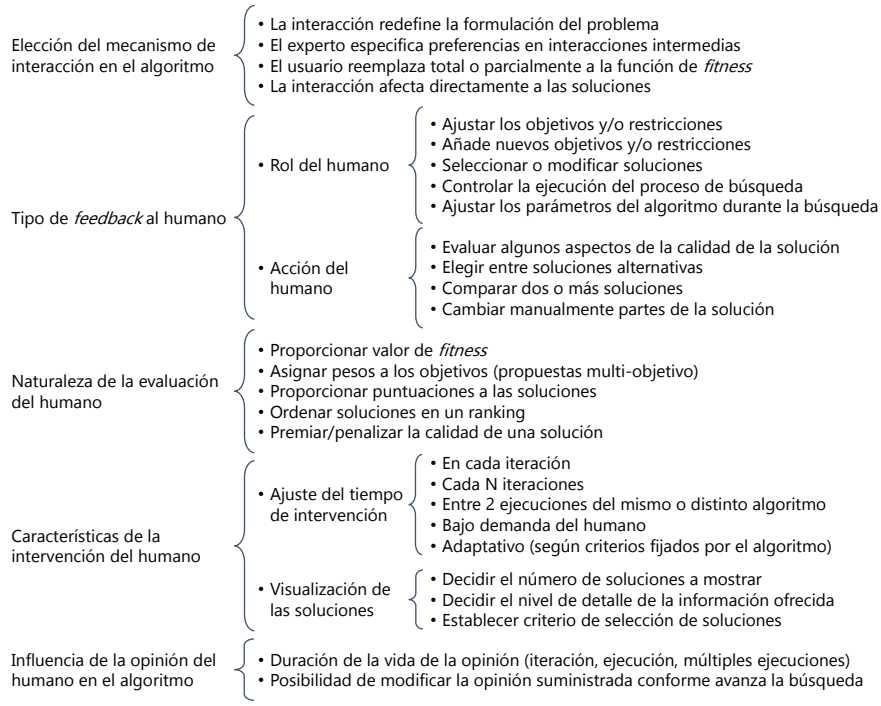

Figura 1. Factores a considerar en el diseño de propuestas en iSBSE

consistente. No obstante, la evaluación de soluciones, directa o indirecta, es una acción frecuente en iSBSE. Otras acciones habituales son la selección, comparación o modificación de soluciones. Este tipo de acciones son más cercanas al modo en que un experto toma decisiones y pueden tener mayor impacto en otras fases del algoritmo, como la generación de nuevas soluciones.

3. Proceso de diseño de un algoritmo interactivo

La Figura 1 detalla los cinco factores que deben considerarse en el diseño de un modelo interactivo [6]. A continuación se describen brevemente:

1. Mecanismo de interacción. Hace referencia al objetivo que se persigue con la interacción.
2. Tipo de *feedback*. Incluye aspectos como el rol que tomará el humano y su ámbito de actuación.
3. Naturaleza de la evaluación. En el caso de que la interacción contemple la evaluación de soluciones, es necesario establecer cómo se va a realizar.
4. Intervención del humano. Considera la frecuencia con la que se interactúa con el humano y el criterio de selección de las soluciones a mostrar.

5. Influencia de la opinión. Determina la validez de la información, esto es, durante cuánto tiempo va a ser considerada y si puede modificarse.

Estos factores son independientes de cómo el algoritmo genera y transforma las soluciones durante el proceso de búsqueda. No obstante, es posible que la elección de las características concretas del modelo de interacción pueda verse influenciada por el problema de ingeniería del software a resolver. Por ejemplo, la posibilidad de redefinir o incluir restricciones y/o objetivos (funciones de *fitness*) claramente va a depender de la formulación del problema. El número de soluciones a mostrar y su nivel de detalle pueden verse restringidas por la naturaleza de las soluciones (código, diagramas, etc.) ya que la carga cognitiva inherente a su análisis puede ser muy diferente.

Dos de los aspectos más importantes hacen referencia a las acciones y al tipo de evaluación que se le permite hacer al experto, pues están muy relacionados con la formulación que se haga del problema de ingeniería del software y con el objetivo que se persigue al incorporar la interactividad. Respecto al tipo de acción, el humano podrá: (1) *evaluar* algún aspecto de la calidad de las soluciones, (2) *seleccionar* las soluciones que presentan ciertas características de su interés, (3) *comparar* dos o más soluciones y (4) *modificar* partes de la solución para corregirlas. Por otro lado, si la interactividad implica, de manera directa o indirecta, la evaluación de las soluciones, el humano podrá realizarla:

- Proporcionando un valor para la función de *fitness*;
- Asignando *pesos* relativos a los objetivos en un problema multi-objetivo;
- Puntuando mediante un valor discreto entre un rango dado;
- Ordenando un conjunto de soluciones en base a su calidad (*ranking*);
- Premiando o penalizando la presencia de alguna característica en la solución.

4. API de desarrollo en Java

Tras identificar los principales elementos a considerar en el diseño de una propuesta iSBSE, esta sección aborda los aspectos requeridos para su implementación. Con el objetivo de facilitar la integración con otros desarrollos, se establece un marco de desarrollo común para iSBSE, a la vez que se da libertad a los investigadores a la hora de adaptar la interacción a problemas software concretos. Además, se han seguido buenas prácticas como el uso de patrones de diseño [12]. Para la implementación se ha escogido el lenguaje Java por ser el más empleado en la práctica[1], siendo también el más frecuentemente utilizado para el desarrollo de frameworks basados en metaheurísticas [13].

La Figura 2 muestra las clases e interfaces principales de la API, donde la letra cursiva representa clases abstractas[2]. Dos conceptos clave en la API son la *sesión interactiva* y el *evento interactivo*. Ambos conceptos son representados

[1] https://www.tiobe.com/tiobe-index/

[2] El código y especificación de la API, así como diagramas adicionales, se encuentran disponibles en http://www.uco.es/grupos/kdis/sbse/isbse/api

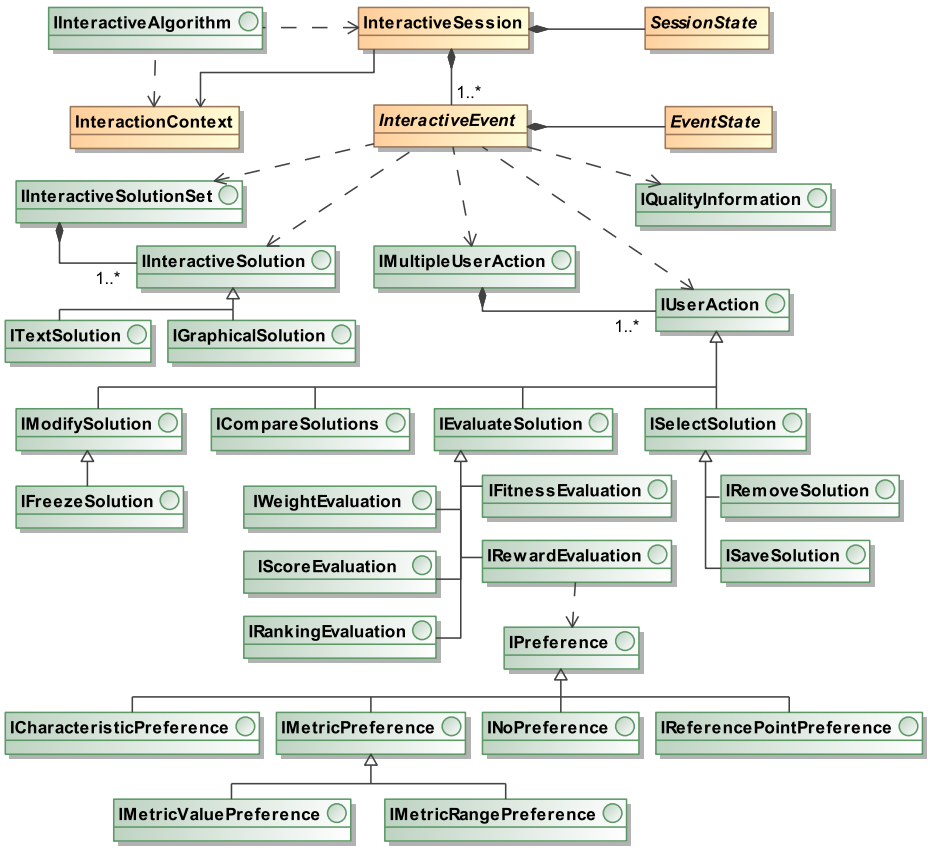

Figura 2. Clases e interfaces principales que componen la API para iSBSE

por las clases `InteractiveSession` e `InteractiveEvent`, respectivamente, y tienen asociado un estado. En primer lugar, una sesión interactiva simboliza cada una de las interrupciones del algoritmo destinadas a interactuar con el humano. Es por tanto el elemento que cualquier algoritmo que implemente la interfaz `IInteractiveAlgorithm` deberá crear y manejar en el contexto de un modelo interactivo (`InteractiveContext`). Dentro de cada sesión, un evento interactivo se define como cada uno de los momentos en los que el humano realiza alguna acción para un conjunto diferente de resultados intermedios. Cada evento deberá implementar las interfaces correspondientes a las acciones definidas en la sección 3. Dichas acciones se realizarán para un tipo de solución determinado (`IInteractiveSolution`), sobre la que se puede mostrar información adicional (`IQualityInformation`). En el caso de la evaluación, se contemplan todos los tipos descritos en la sección 3. Las siguientes subsecciones explican cada una de estas clases e interfaces en detalle.

4.1. Algoritmo interactivo y contexto de interacción

Para no condicionar la codificación del modelo interactivo a la estructura con la que se codifique el algoritmo, toda la funcionalidad relacionada con la interactividad es encapsulada en la sesión. Para ello, todo algoritmo interactivo debe implementar la interfaz `IInteractiveAlgorithm`. Esta interfaz consta de dos métodos, `getInteractiveSession()` y `setInteractiveSession()`, que obligan al algoritmo a dar acceso a la sesión interactiva cada vez que se vaya a realizar una interacción.

Con el objetivo de que el algoritmo pueda disponer de información adicional sobre el ámbito de la aplicación en la que se está ejecutando, la interfaz `IInteractiveAlgorithm` también requiere manejar un objeto de contextualización. Para ello la API define la clase `InteractionContext`, la cual permite almacenar un identificador de tipo cadena, para identificar la ejecución o el experimento actual; el idioma en el que se deben devolver mensajes de información o ayuda sobre la interacción, según los códigos definidos en la enumeración `Language`; y el número de sesiones planificadas y el número de eventos que conforman cada una, en caso de que sean fijados a priori.

4.2. Sesiones y eventos interactivos

La sesión es representada con la clase `InteractiveSession` y se compone de un contexto de interacción, una lista de eventos y un estado, tal y como se muestra en el código 1.

```
1  public class InteractiveSession {
2   protected InteractionContext context;        // El contexto de interacción
3   protected List<InteractiveEvent> events;     // La secuencia de eventos
4   protected SessionState state;                // El estado de la sesión
5    ...
6   public InteractiveSession(InteractionContext context) {
7    this.context = context;
8    this.events = new ArrayList<InteractiveEvent>();
9    this.state = SessionStateUndefined.getInstance();
10  }
11  public InteractiveSession(InteractionContext context, List<InteractiveEvent>
        events) {
12   this.context = context;
13   this.events = events;
14   this.events.forEach(e -> e.setInteractionContext(this.context));
15   this.state = SessionStateIdle.getInstance();
16  }
17   ...
18 }
```

Código 1. Propiedades y constructores de la clase *InteractiveSession*

En primer lugar, cabe señalar que, en cualquier caso, la sesión siempre debe tener asociado un contexto de interacción. Este objeto deberá ser creado y

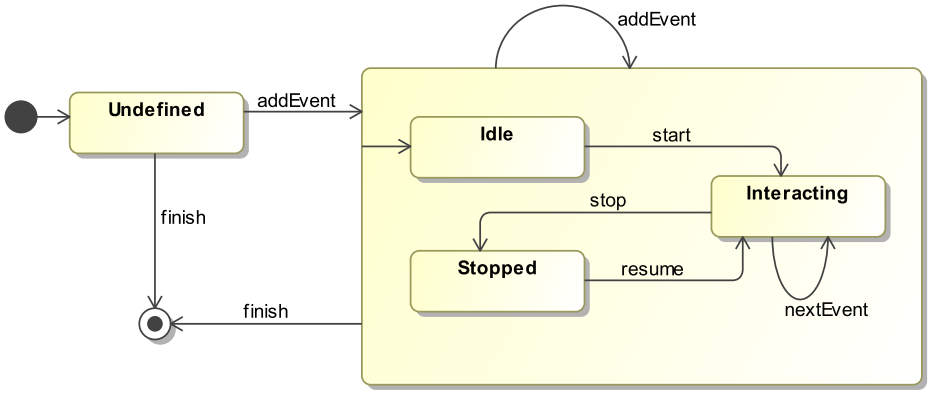

Figura 3. Máquina de estados para una sesión interactiva

configurado por el algoritmo antes de crear la sesión. La sesión puede ser inicializada con una lista de eventos o no (a criterio del programador), en cuyo caso deberán añadirse posteriormente con el método `addEvent`. Cada evento contendrá la información necesaria para realizar una interacción, y serán procesados secuencialmente hasta completar la sesión o hasta que el humano decida terminarla. En función de si la sesión se crea con eventos o sin ellos, el estado de la sesión va a ser diferente. Como puede observarse, la sesión estará en estado "*Idle*" en el primer caso y en estado "*Undefined*" en el segundo.

La Figura 3 muestra los posibles estados de una sesión. Una sesión en estado "*Undefined*" pasará a estado "*Idle*" cuando tenga al menos un evento interactivo asociado. Una vez iniciada, la sesión irá procesando los eventos interactivos a medida que el humano interactúe. Las distintas transacciones (*start*, *stop*, etc.) hacen referencia a las acciones de control que se pueden recibir externamente, por ejemplo, por medio del controlador de una interfaz gráfica. De esta forma, también se contemplan posibles interrupciones debidas a problemas de comunicación entre el algoritmo y la interfaz. Cuando se procesa la transición *finish*, la sesión se da por concluida, tanto si el humano ha decidido terminar la interacción antes de realizar todos los eventos programados como si estos han sido completados satisfactoriamente. Cada estado de la sesión se ha implementado como una clase independiente siguiendo las directrices de los patrones de diseño *State* y *Singleton* [12].

Cada uno de los eventos interactivos que conforman la sesión se define mediante la clase abstracta `InteractiveEvent`. Esta clase encapsula la funcionalidad genérica del evento, principalmente su cambio de estado y el control de los tiempos de ejecución e interrupción. El manejo del estado sigue la misma filosofía que para la sesión. Como puede verse en la Figura 4, un evento interactivo puede pasar por tres estados: "*Idle*", "*Interacting*" e "*Interrupted*". Dado que la clase es abstracta, el desarrollador debe proporcionar su propia implementación del evento interactivo de acuerdo al modelo de interacción diseñado. Para ello deberá extender la clase `InteractiveEvent` e implementar aquellas interfaces

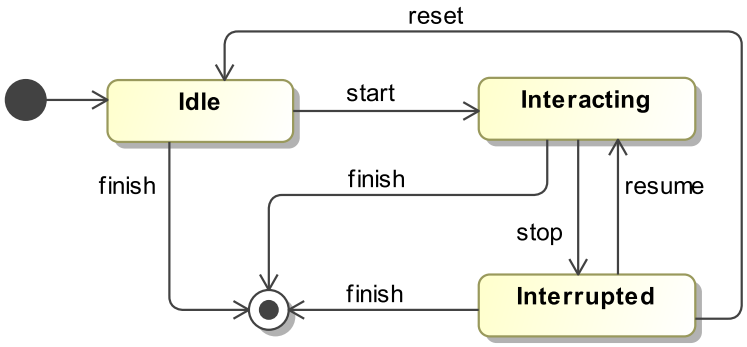

Figura 4. Máquina de estados para un evento interactivo

relacionadas con las acciones de usuario y la visualización de las soluciones. En la sección 5 se ilustra este proceso con un ejemplo.

4.3. Visualización de soluciones

Es habitual que un evento interactivo necesite mostrar al humano varias soluciones para que las analice y emita su opinión, o bien realice alguna acción sobre ellas. Actualmente, la API proporciona las interfaces `ITextSolution`, para aquellas soluciones que puedan representarse en modo texto (código, especificación de requisitos, etc.), e `IGraphicalSolution`, destinada a la visualización de diagramas. En este último caso, se espera que la solución sea almacenada en un fichero. Otra posibilidad es representarla en algún lenguaje estructurado como XMI. En este sentido, se ha incluido la interfaz `IXMISolution`, subtipo de `IGraphicalSolution`. La responsabilidad de codificar la solución en un formato legible por la interfaz gráfica recae en el programador.

Aparte de visualizar la solución, es posible que se quiera incluir información adicional sobre su calidad, con el fin de ayudar al humano en la toma de decisiones. Para ello, la interfaz `IQualityInformation` define una serie de métodos para recuperar valores asociados a las medidas que elija el propio desarrollador. Por ejemplo, en un modelo para refactorización de código, se podrían incluir medidas como LOC (*lines of code*) o CBO (*coupling between objects*). Además, también se da soporte a la definición de medidas con múltiples valores que permitirían, por ejemplo, indicar la complejidad ciclomática de cada método.

4.4. Acciones de usuario

Cada evento interactivo puede solicitar la realización de una o varias acciones por parte del humano. Por este motivo, se ha definido la interfaz `IUserAction`, para acciones simples, así como `IMultipleUserAction`, para acciones compuestas (véase la Figura 2). Esta interfaz simplemente impone el acceso a una lista de

acciones simples. Para cada acción simple se ha definido una interfaz que extiende de de `IUserAction` y tiene un tipo asociado (representado por la enumeración `InteractiveActionType`). Las interfaces actualmente definidas son:

- `ICompareSolution`: define una serie de métodos para indicar la mejor solución, la peor, o un orden relativo entre todas las mostradas.
- `IEvaluateSolution`: define métodos para llevar a cabo la evaluación. Su especialización según el tipo de evaluación se aborda en la siguiente sección.
- `ISelectSolution`: incluye métodos para indicar qué solución, de entre las mostradas, ha sido elegida por el humano. Dos casos particulares, definidos mediante interfaces, consisten en eliminar una solución (`IRemoveSolution`) y marcarla para guardarla en un archivo (`ISaveSolution`).
- `IModifySolution`: declara un único método para actualizar la solución. Un caso particular es la posibilidad de "congelar" una parte de la solución que es de especial interés, para lo cual se ha definido la interfaz `IFreezeSolution`.

Todas las interfaces relacionadas con las acciones de usuario se basan en la definición de pares de métodos `get/set` para manejar el acceso a las propiedades que representarán la elección del experto. De esta forma, el *feedback* del humano puede almacenarse en el propio evento para su posterior procesamiento en el algoritmo. Además, la interfaz `IUserAction` define métodos para obtener el nombre y la descripción de la acción, de forma que dicha información pueda ser mostrada a modo de instrucciones de uso.

4.5. Mecanismos de evaluación

Se ha definido una interfaz para cada uno de los mecanismos presentados en la sección 3: `IFitnessEvaluation`, `IWeightEvaluation`, `IScoreEvaluation`, `IRankingEvaluation`, `IRewardEvaluation`. Todas estas interfaces extienden de `IEvaluateSolution`, añadiendo métodos `get/set` específicos en función del tipo de evaluación. Así, `IWeightEvaluation` contiene métodos para asignar tantos pesos como objetivos tenga el problema, mientras que `IScoreEvaluation` permite seleccionar uno o varios valores de entre un conjunto de alternativas.

Un caso particular de evaluación de especial utilidad en iSBSE es el denominado "*premiar/penalizar*", ya que permite al humano indicar preferencias cualitativas [14]. En la API, esta idea se refleja en la interfaz `IRewardEvaluation`, donde se especifica que el algoritmo interactivo deberá proporionar una lista de posibles preferencias, que deben extender la interfaz `IPreference`. Esta interfaz impone la implementación de métodos para describir la preferencia, conocer su tipo (entre los definidos en la enumeración `PreferenceType`) y asignar un nivel de confianza a la hora de establecerla. Aunque la implementación concreta va a depender de qué preferencias se puedan aplicar al problema en cuestión, se han definido cuatro tipos de preferencias y sus corrrespondientes subinterfaces:

- `ICharacteristicPreference` permite premiar o penalizar la presencia de una característica estructural de la solución.

- `IMetricPreference` permite indicar una preferencia con respecto al valor deseado para una medida (`IMetricValuePreference`) o a su rango esperado (`IMetricRangePreference`).
- `INoPreference` permite representar la ausencia de preferencia, en caso de que se desee tener constancia de ello.
- `IReferencePointPreference` permite indicar el valor deseado para cada uno de los objetivos en un espacio multi-objetivo, asociándoles un peso.

5. Ejemplo práctico

Esta sección ilustra cómo algunas de las interfaces de la API son implementadas para el problema del descubrimiento de arquitecturas software [14,15]. Para este problema, cada interacción consiste en la visualización y evaluación de una arquitectura basada en componentes. Para cada una, el experto puede indicar una serie de preferencias, tales como cuál es el mejor o el peor componente, o los rangos de cohesión y acoplamiento apropiados. Además, se le permite guardar la solución en un archivo, marcarla para su eliminación y "congelar" uno de sus componentes, impidiendo así mover las clases que lo conforman.

Como se detalló en la sección 4.2, en primer lugar es necesario crear una implementación concreta de un evento interactivo. Como puede verse en el código 2, la clase `InteractiveDiscoveryEvent` extiende de `InteractiveEvent` e implementa las interfaces necesarias para manejar una solución en formato XMI (`IXMISolution`) y su información de calidad (`IQualityInformation`), y dar soporte a múltiples acciones de usuario (`IMultipleUserAction`). Entre las propiedades declaradas se encuentran la solución a mostrar, `xmiSolutionDiagram`, que contiene el diagrama de componentes en formato XMI, un conjunto de medidas de diseño (`designMetrics`) para cada uno de los componentes de la solución candidata, un conjunto de medidas simples sobre el proceso de búsqueda (`searchMetrics`) y la lista de posibles acciones de usuario (`actions`). El único método abstracto de la clase `InteractiveEvent` obliga a devolver una descripción del evento. Aquí, dicha descripción es leída de un fichero de propiedades en función del idioma almacenado en el contexto.

```
1  public class InteractiveDiscoveryEvent extends InteractiveEvent
2      implements IXMISolution, IQualityInformation, IMultipleUserAction {
3    protected String xmiSolutionDiagram; // Solución a mostrar
4    protected double [][] designMetrics;    // Medidas de diseño por componente
5    protected double [] searchMetrics;      // Medidas sobre la búsqueda
6    protected List<IUserAction> actions; // Lista de acciones
7    public String getDescription() {
8      String description = LanguagePropertyReader.getInstance().getProperty("
           event.description", this.context.getLanguage());
9      return description;
10   } ...
11 }
```

Código 2. Definición del evento interactivo para el caso de estudio

El siguiente paso consiste en implementar los métodos definidos en cada una de las interfaces. Por motivos de espacio, el código 3 muestra solo algunos de ellos. Por ejemplo, la interfaz `IXMISolution` define un método para indicar si la solución mostrada puede ser modificada por el humano, y el tipo concreto de solución, de entre los definidos en `InteractiveSolutionType`. Respecto al manejo de las acciones, se debe permitir el acceso a las mismas según su tipo (`getUserActionByType()`) y descartar cualquier información marcada previamente por el usuario (`clearAllActions()`).

```
1  public class InteractiveDiscoveryEvent extends InteractiveEvent
2      implements IXMISolution, IQualityInformation, IMultipleUserAction {
3     ...
4     // Métodos de la interfaz IXMISolution
5     public boolean isSolutionModifiable() { return false; }
6     public InteractiveSolutionType getSolutionType() {
7       return InteractiveSolutionType.COMP_DIAGRAM;
8     }...
9     // Métodos de la interfaz IMultipleUserAction
10    public IUserAction getUserActionByType(InteractiveActionType type) {
11      for(IUserAction action: this.actions) {
12        if(action.getActionType() == type) { ... }
13      }
14      return null;
15    }
16    public void clearAllActions() {
17      this.actions.forEach(a -> a.clearAction());
18    } ...
19 }
```

Código 3. Implementación de operaciones sobre soluciones y acciones

El código 4 muestra parte de la implementación para la acción *"congelar componente"*, como ejemplo de una acción de usuario. La interfaz `IFreezeSolution` declara métodos para indicar el número de elementos que pueden ser congelados (1 en este caso) mediante el método `getNumberFreezableElements()`, así como métodos para asignar y acceder al índice del elemento congelado.

```
1  public class FreezeComponentAction implements IFreezeSolution {
2    protected int indexFrozenComponent; // Índice del componente seleccionado
3    public void clearAction() { this.indexFrozenComponent = -1; }
4    public boolean isFrozenSolution() { return (this.indexFrozenComponent
         !=-1); }
5    public int getNumberFreezableElements() { return 1; }
6    public void setFrozenElement(int index) { this.indexFrozenComponent =
         index; }
7    public int getFrozenElement() { return this.indexFrozenComponent; }
8     ...
9  }
```

Código 4. Implementación de la acción *congelar componente*

Finalmente, el mecanismo de evaluación para este ejemplo se basa en premiar o penalizar características de la arquitectura mostrada. Por ello, es necesario definir los distintos tipos de preferencias (veáse la sección 4.5). Una de las preferencias posibles consiste en seleccionar el mejor componente de la solución. Para ello se ha definido la clase `BestComponent`, que implementa la interfaz `ICharacteristicPreference`. Como puede verse en el código 5, una preferencia de este tipo permite seleccionar una o varias opciones, según se indique en el método `isMultipleOptionAllowed()`. En este caso, habrá tantas opciones como componentes tenga la solución mostrada y se permitirá elegir únicamente uno de ellos como el mejor.

```
1  public class BestComponent implements ICharacteristicPreference {
2    protected int numberOfComponents;        // Número de componentes
3    protected int index;                     // Índice del componente seleccionado
4    public int getNumberOfOptions() { return this.numberOfComponents; }
5    public boolean isMultipleOptionAllowed() { return false; }
6    public void selectOption(int index) { this.index = index; }
7    public boolean isOptionSelected(int index) { return (index == this.index); }
8    public int getSelectedOption() { return this.index; }
9     ...
10 }
```

Código 5. Implementación de la preferencia *mejor componente*

6. Conclusiones

La consideración de enfoques interactivos puede promover una mayor adopción de las técnicas de búsqueda a la hora de resolver problemas en ingeniería del software, ya que permiten acercar el proceso al ingeniero. Sin embargo, diseñar un algoritmo interactivo requiere de un estudio profundo para decidir el modo de interacción más apropiado. Basándose en los trabajos más relevantes del área, este artículo recopila los factores fundamentales a considerar en el desarrollo de propuestas iSBSE y ofrece un marco de desarrollo común a través de una API.

La API, implementada en Java, da soporte programático a las acciones y mecanismos de evaluación más frecuentes detectados en el ámbito de iSBSE. Esta API puede resultar de gran utilidad a la comunidad SBSE, pues marca las pautas necesarias para el desarrollo de propuestas interactivas y establece una capa de abstracción que permite independizar la implementación del algoritmo de búsqueda del modelo de interacción. Además, las clases e interfaces que componen la API pueden ser adaptadas y combinadas a criterio del investigador para adecuar la interactividad a cada problema software concreto.

La API debe aún ser extendida para cubrir el resto de factores mencionados en la sección 3, por lo que el trabajo futuro más inmediato estará encaminado a incluir nuevas interfaces para el ajuste del tiempo de interacción y el modelado del *feedback* del humano. También se contempla su integración con frameworks de desarrollo habituales en SBSE.

Agradecimientos

Trabajo financiado por el Ministerio de Economía y Competitivad (TIN2017-83445-P), el Ministerio de Educación (FPU13/01466), y fondos FEDER.

Referencias

1. M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–64, 2012.
2. I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Inf. Sci.*, vol. 237, pp. 82–117, 2013.
3. C. Simons, J. Singer, and D. R. White, "Search-based refactoring: Metrics are not enough," in *Proc. 7th Int. Symposium on Search-Based Software Engineering*, pp. 47–61, 2015.
4. G. R. Santhanam, "Qualitative optimization in software engineering: A short survey," *J. Syst. Softw.*, vol. 111, pp. 149–156, 2016.
5. D. Meignan, S. Knust, J.-M. Frayret, G. Pesant, and N. Gaud, "A Review and Taxonomy of Interactive Optimization Methods in Operations Research," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 3, pp. 17:1–43, 2015.
6. A. Ramírez, J. R. Romero, and C. Simons, "A Systematic Review of Interaction in Search-Based Software Engineering," *IEEE Trans. Softw. Eng.*, En prensa, 2018.
7. I. C. Parmee, "Poor-Definition, Uncertainty, and Human Factors - Satisfying Multiple Objectives in Real-World Decision-Making Environments," in *Proc. Int. Conf. Evolutionary Multi-Criterion Optimization*, pp. 52–66, 2001.
8. C. L. Simons, J. Smith, and P. White, "Interactive ant colony optimization (iACO) for early lifecycle software design," *Swarm Intell.*, vol. 8, no. 2, pp. 139–157, 2014.
9. B. Amal, M. Kessentini, S. Bechikh, and J. Dea, "On the Use of Machine Learning and Search-Based Software Engineering for Ill-Defined Fitness Function: A Case Study on Software Refactoring," in *Proc. 6th Int. Symposium on Search-Based Software Engineering*, pp. 31–45, 2014.
10. A. A. Araújo, M. Paixao, I. Yeltsin, A. Dantas, and J. Souza, "An architecture based on interactive optimization and machine learning applied to the next release problem," *Automat. Softw. Eng.*, vol. 24, no. 3, pp. 623–671, 2017.
11. B. Marculescu, R. Feldt, R. Torkar, and S. Poulding, "An initial industrial evaluation of interactive search-based testing for embedded software," *Appl. Soft Comput.*, vol. 29, pp. 26–39, 2015.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Pearson, 1995.
13. J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernández, "Metaheuristic optimization frameworks: A survey and benchmarking," *Soft Comput.*, vol. 16, no. 3, pp. 527–561, 2012.
14. A. Ramírez, J. R. Romero, and S. Ventura, "Interactividad en el descubrimiento evolutivo de arquitecturas," in *XX Jornadas en Ingeniería del Software y Bases de Datos*, 2015.
15. A. Ramírez, J. R. Romero, and S. Ventura, "An approach for the evolutionary discovery of software architectures," *Inf. Sci.*, vol. 305, pp. 234–255, 2015.