UNIVERSITAT POLITÈCNICA DE VALÈNCIA | IN-RED 2023 IX Congreso de Innovación Educativa y Docencia en Red

Congreso In-Red 2023
UPV, 13 – 14 de julio de 2023
Doi:https://doi.org/10.4995/INRED2023.2023.16596

Pruebas unitarias para evaluar automáticamente la programación de clases en python

Laura Climent[1] **y Alejandro Arbelaez**[1]

[1]Departamento de Ingeniería Informática, Universidad Autónoma de Madrid (UAM), Spain.
laura.climent@uam.es alejandro.arbelaez@uam.es

How to cite: Laura Climent y Alejandro Arbelaez. 2023. Pruebas unitarias para evaluar au-tomáticamente la programación de clases en python. En libro de actas: *IX Congreso de Inno-vación Educativa y Docencia en Red*. Valencia, 13 – 14 de julio de 2023.
doi:https://doi.org/10.4995/INRED2023.2023.16596

Abstract

The Automatic Assessment (AA) of tasks presents several advantages, such as: its application to a large number of students (due to the very short time that it requires compared to manual evaluations), automatic and immediate feedback to the students and lack of subjectivity .

In this paper we explain how to design unit tests to evaluate how the Object Oriented Programming (OOP) classes of the assignment have been programmed. In this paper, we focus on the Python programming language (rather than using Java as other works of literature). In addition, we present a real case of practical laboratory assignment. We also present their solution and the corresponding unit tests.

Finally, we prove the effectiveness of the unit tests by verifying that the results obtained are the same as the evaluation carried out by visual inspection of the students' code. In addition, we also received positive feedback from students on the work presented in this article.

Keywords*: Automatic assessment, unit testing, object-oriented programming, python.*

Resumen

La Evaluación Automática (EA) de tareas presenta varias ventajas, tales como: su aplicación a un gran número de estudiantes (debido al tiempo tan reducido que requiere en comparación con las evaluaciones manuales), valoración automática e inmediata para los estudiantes y falta de subjetividad.

En este trabajo explicamos como diseñar pruebas unitarias para evaluar como han sido programadas las clases de tareas de Programación Orientada a Objetos (POO). En este trabajo, nos enfocamos en el lenguaje de programación Python (en vez de utilizar Java tal y como lo hacen los otros trabajos de la literatura). Además, presentamos un

caso real de un trabajo práctico de laboratorio. También presentamos su solución y las correspondientes pruebas unitarias.

Finalmente, corroboramos la efectividad de los tests unitarios comprobando que los resultados obtenidos son los mismos que la evaluación realizada por inspección visual del código de los estudiantes. Además, también recibimos valoraciones positivas de los alumnos del trabajo presentado en este artículo.

***Keywords**: Evaluación automática, pruebas unitarias, programación orientada a objetos.*

1 Introducción

En los últimos años, la enseñanza se ha orientado hacia la evaluación continua. En los grados de ingeniería, especialmente, tener varias tareas de programación es esencial para el aprendizaje, ya que, de esta forma, se pueden aplicar los conceptos adquiridos a escenarios reales y los estudiantes pueden tener valoraciones sobre su código.

La evaluación de trabajos puede ser manual o (semi)automática. Sin embargo, proporcionar varias evaluaciones de forma manual durante todo el curso es una tarea muy exigente y que requiere mucho tiempo, especialmente cuando el número de estudiantes es grande. Últimamente, el número de estudiantes de carreras de ingeniería ha tendido a crecer. Además, existe otra tendencia creciente por la alta demanda de cursos masivos abiertos en línea. En estos escenarios, calificar manualmente varias evaluaciones continuas es inviable o desaconsejable. Por estas razones, existe una tendencia creciente hacia las Evaluaciones Automáticas (EA).

La evaluación es una tarea delicada en la enseñanza porque implica un cierto grado de subjetividad. Además, este nivel de subjetividad puede dar lugar a posibles quejas informales/formales por parte de los estudiantes. La EA soluciona este problema. Además, tiene otras ventajas, como la valoración instantánea que se puede proporcionar a los estudiantes, de manera similar a la tutoría individual. La EA puede orientarse hacia los resultados de aprendizaje del curso, permitiendo que el profesor y a los observadores externos puedan comprobar que los alumnos han alcanzado los objetivos de aprendizaje de las guías docentes.

En la literatura, hay muchos trabajos previos de EA para cursos de introducción a la programación e industria, pero no para Programación Orientada a Objetos (POO) en enseñanza. Por ello, descartamos las siguientes subareas del estado del arte: (i) EA de tareas de programación (sin pruebas unitarias y no para POO), (ii) pruebas unitarias para la docencia de cursos introductorios de programación (no para POO) y (iii) pruebas unitarias para POO en la industria. Respecto a estas tres subáreas, nos gustaría mencionar que Ihantola et al., 2010 presenta una revisión detallada de los trabajos de la subárea (i). Acerca de (ii), hay muchos trabajos, entre ellos: Barriocanal et al., 2002, Whalley y Philpott, 2011, Combéfis y Paques, 2015, etc. Con respecto a (iii), entre otros, algunos autores aplican aprendizaje automático (machine learning) a POO (por ejemplo, Touré y Badri, 2018), otros trabajos se enfocan en generar automáticamente casos de prueba para pruebas unitarias (por ejemplo, Wappler y Wegener, 2006, Ribeiro et al., 2009, Hsiao et al., 2009, etc.).

Sólo hay dos trabajos previos sobre EA para enseñanza con pruebas unitarias para POO: Torchiano y Morisio, 2009 y Torchiano y Bruno, 2018. En ambos trabajos se proporciona a los alumnos clases wrapper (envoltorio) en lenguaje Java con las definiciones de todas las funciones que requiere la tarea, (por ejemplo, *obtener_Persona()*) y excepciones (por ejemplo, *ErrorPersonaNoExistente*).

Además, los autores diseñan pruebas unitarias en formato JUnit para verificar que cada función de la clase wrapper ha sido correctamente implementado por los estudiantes. Sin embargo, en este artículo usamos el lenguaje de programación python tanto para el enfoque de EA (con la biblioteca *unittest*) como para el caso de estudio. Como se menciona en Srinath, 2017, Python se considera el lenguaje de programación de más rápido crecimiento en los últimos tiempos. Python está cada año más presente en las ingenierías y también específicamente en los cursos de POO. Esto ha motivado el trabajo desarrollado en este artículo. Ya que hay una necesidad de tener una método de EA en python para los cursos de POO.

2 Objetivos

En general, los objetivos de este artículo y principales contribuciones del artículo son la presentación de un caso de estudio real de tarea de laboratorio y la EA de la programación de sus clases en Python mediante sus correspondientes test unitarios. Esto incluye la EA de:

- *El correcto uso de los atributos públicos, protegidos y privados.*
- *La correcta creación de instancias, con sus correspondientes tipos de datos y rango de valores.*
- *Manipulación adecuada de atributos*
- *Manejo de errores en Python.*

3 Desarrollo de la innovación

En esta sección primero presentamos el caso de estudio de la tarea y posteriormente los test unitarios correspondientes.

3.1 Tarea propuesta

En este artículo presentamos un caso de estudio real de una tarea de programación que está orientada al tema de los videojuegos. Otros autores, como en Chen y Cheng, 2007 presentaron una tarea combinada con un videojuego y justifican que hay una razón pedagógica detrás de esta elección. En primer lugar, casi todos los alumnos disfrutan de los juegos de ordenador y, por tanto, les resulta más divertido trabajar con ellos que en proyectos más convencionales. En segundo lugar, los videojuegos se componen de una cierta cantidad de objetos complejos que interactúan y que se pueden modelar con el diseño de POO.

La tarea consiste en desarrollar una aplicación de librería que permita la creación de personajes de un videojuego. Los personajes tienen cierto tipo de características y funcionalidades. Por ejemplo, tienen nombre y ciertas armas, grados de fuerza, podrían pelear entre ellos, etc.

En este artículo proponemos la creación del personaje *Orco*. (También se pueden proponer otros personajes similares, como arquero, caballero, elfo, etc.). Los estudiantes deben programar su clase, la cual se define con los siguientes atributos:

- El *nombre del orco* (por ejemplo, "Ogrorg")
- La puntuación de *fuerza del orco* en el dominio [0-5]
- ¿Tiene el orco un *arma?* dentro del dominio [Verdadero, Falso]

Los métodos asociados a la clase son:

- Un *constructor* para inicializar instancias (por ejemplo, orc1 = Orc("Ogrorg", 4.3, True)). En Python, el constructor es un método especial llamado *__init__*.
- Las *propiedades* para acceder y modificar los valores de todos los atributos (por ejemplo, orc1.name = "Grunghi").

Es importante remarcar, que pueden añadirse más métodos. Por ejemplo, el método __str__ (método especial de Python que se usa para mostrar la información de un objeto), métodos para luchar entre orcos, etc.

Además, se les pide a los estudiantes que implementen el *manejo de errores* de tipos y valores para los atributos de la clase. Deben verificar que la configuración de valores por parte de usuarios externos del software sea correcta y dentro del dominio de los atributos. De esta manera, el usuario introduce un valor numérico superior al permitido, se truncará al máximo valor (por ejemplo, el valor de fuerza máxima es 5). Si el valor es inferior al permitido, se truncará al mínimo valor (por ejemplo, el valor mínimo de fuerza es 0). Para errores relacionados con el tipo de atributos (por ejemplo, intentar asignar una fuerza igual a "Ogrorg"), la asignación no se completará y el mensaje de error "type ERROR" se imprimirá en la pantalla.

Los estudiantes también deben implementar un módulo de prueba para verificar todas las funcionalidades de la clase Orc. Deben importar el módulo Orc y probar cada una de las funcionalidades descritas anteriormente.

3.2 Solución

A continuación, presentamos las pruebas unitarias para la EA del código de los estudiantes para la tarea del caso de estudio presentada anteriormente. El principal objetivo de las pruebas unitarias es asegurar que una determinada unidad de código funcione correctamente. Como se menciona en Wappler y Wegener, 2006 el código para pruebas unitarias de POO es más complejo. Por esta razón, se deben desarrollar secuencias de llamadas a métodos que realicen escenarios de pruebas unitarias interesantes. Durante la ejecución de la prueba unitaria, todos los objetos que participan en la tarea deben crearse y ponerse en estados particulares llamando a varios métodos asociados a estos objetos. Por lo general, cada caso de prueba se centra en un método en particular. Para realizar una prueba unitaria completa, se deben verificar todos los métodos dentro de la clase.

Python ofrece una biblioteca para pruebas unitarias llamada *unittest* que importamos a nuestro archivo de prueba. Además, importamos el código de cada alumno y lo llamamos *mod*. También incluimos las bibliotecas básicas típicas, como *io* y *sys*. Posteriormente, después de importar dichas

bibliotecas, ya podemos crear una nueva clase (la llamamos *CharactersTest*) que hereda de unittest e incluye todas las pruebas. A continuación, presentamos el código asociado.

```
import io

import sys

import unittest

import code_student as mod


class CharactersTest(unittest.TestCase):

    ...
```

Cada test unitario consta de una secuencia de llamadas a métodos y una o más aserciones ("assertion") Wappler y Wegener, 2006. La aserción verifica que la condición que sigue sea válida. Si todas las aserciones de una prueba son válidas, el resultado de la prueba unitaria para la prueba será *ok*. De lo contrario, será *fail*. A continuación se muestran dos ejemplos del resultado de dos pruebas unitarias, donde la primera falla y la segunda es correcta.

```
test_values_range_constr_orc

   (__main__.CharactersTest) ... FAIL

test__str__Archer

    (__main__.CharactersTest) ... ok
```

Para el cálculo de la nota de la tarea, cada prueba unitaria puede tener asociada una parte de la puntuación total. Por lo tanto, la corrección de las tareas se puede hacer automáticamente con solo ejecutar las pruebas unitarias presentadas en esta sección. Ya que cuando falla una prueba unitaria, mostrará detalles sobre el fallo por pantalla.

Además de las pruebas unitarias, definimos una función externa que se encarga de capturar la salida estándar del código del estudiante. Necesitamos esta función para verificar si el código del estudiante está manejando los errores de la manera especificada por la tarea (con mensajes de error impresos en la pantalla, como "type ERROR"). Una vez que la salida estándar ha sido capturada y almacenada como una cadena, podemos compararlo con el mensaje de error correcto. La función se llama *capt_out()*. En el anexo (Sección 5) puede verse la descripción de esta función.

A continuación, describimos cada tipo de prueba unitaria. Para que sea más claro, resaltamos en negrita los valores importantes que estamos comprobando dentro de cada prueba unitaria.

Constructores. Primero analizamos la correcta actualización de atributos dentro de los constructores: los rangos de valores y sus tipos

Rango de Valores. En cuanto a la verificación del rango de valores, la fuerza de los orcos no puede exceder los límites de los valores permitidos [0,5]. A continuación, mostramos el código del test unitario asociado. Para ello, creamos un orco con fuerza 5.3. Poste-

riormente, comprobamos que el atributo de fuerza se ha truncado al máximo valor (5). Lo mismo debería ocurrir con una fuerza inferior a la mínima permitida (0).

```
def test_values_range_constr_orc(self):
    orc1 = mod.Orc(.ºgrorg", 5.3, True)
    assert orc1.strength == 5
    orc2 = mod.Orc("Grunch", -100.0, False)
    assert orc2.strength == 0
```

Tipos de Valores. Debemos asegurarnos de que, en el código de los estudiantes, cuando se crean nuevos orcos con tipos incorrectos para sus atributos, aparece un mensaje de "type ERROR" en la pantalla. Por ejemplo, los atributos de los orcos son nombre, fuerza y si tiene arma o no. Por esta razón, en el código a continuación, creamos tres orcos, cada uno con un tipo incorrecto para cada atributo.

```
def test_values_types_constr_orc(self):
    with capt_out() as (out, e):
        mod.Orc(1, 4.3, False)
    assert (out.getValue().strip() == "type ERROR")
    with capt_out() as (out, e):
        mod.Orc("Grunch", "Grunch", False)
    assert (out.getValue().strip() == "type ERROR")
    with capt_out() as (out, e):
        mod.Orc(.ºgrorg", 4.3, .ºgrorg")
    assert (out.getValue().strip() == "type ERROR")
```

Propiedades. Presentamos pruebas unitarias asociadas con la actualización de atributos mediante el uso de propiedades. Tal y cómo hicimos dentro del constructor, debemos verificar los tipos de valores y el rango de los valores. Posteriormente, tenemos que asegurarnos de que después de crear cada uno de los orcos, la recuperación y actualización de la información de sus atributos sea correcta.

```
def test_properties_access_orc(self):
    orc = mod.Orc(``Ogrorg'', 4.1, True)
    assert orc.name == ``Ogrorg''
    assert orc.strength == 4.1
```

```
    assert orc.weapon
    orc.name = "Grunch"
    assert orc.name == "Grunch"
    orc.strength = 3.2
    assert orc.strength == 3.2
    orc.weapon = False
    assert not orc.weapon

def test_properties_values_errors_orc(self):
    orc = mod.Orc(``Ogrorg'', 4.1, True)
    with capt_out() as (out, err):
        orc.name = 1.2
    output = out.getvalue().strip()
    assert (output == "type ERROR")
    with capt_out() as (out, err):
        orc.strength = "Grunch"
    output = out.getvalue().strip()
    assert output == "type ERROR"
    with capt_out() as (out, err):
        orc.weapon = 6.8
    output = out.getvalue().strip()
    assert (output == "type ERROR")
    orc.strength = 10.0
    assert orc.strength == 5.0
    orc.strength = -10.0
    assert orc.strength == 0.0
```

4 Resultados y Conclusiones

En esta sección presentamos los resultados y conclusiones del trabajo presentado en este artículo.

4.1 Resultados

El enfoque presentado en este documento ha sido evaluado como una tarea en un curso universitario de programación de segundo año de la University College Cork (UCC), llamado programacion intermedia, el cual contiene gran parte del temario de POO. Para evaluar la efectividad del trabajo presentado en este artículo, la evaluamos con alumnos de la asignatura que cumplian las siguientes condiciones: i) no tenían experiencia previa en POO, ii) eran capaces de entender los conceptos de POO y iii) estaban dispuestos a responder una encuesta.

Antes de realizar la tarea presentada en este artículo, observamos que la mayoría de nuestros estudiantes, incluso con buenos conocimientos teóricos, todavía encontraban difícil y desafiante el diseño y la implementación de una solución para un problema del mundo real mediante el uso y la aplicación de los principios de POO. Sin embargo, después de desarrollar la tarea presentada en este artículo y obtener su retroalimentación automática asociada (la generada por los test automáticos presentados) la mejora de sus conocimientos en cuanto la applicación de los principios de POO ha sido muy notoria.

Algo que también nos gustaría mencionar es que al realizar esta tarea, se pueden proporcionar alguna parte de las pruebas unitarias a los estudiantes. De esta forma, podrán entender con más claridad el proceso de EA. Especialmente, a los estudiantes que no están familiarizados con la EA (y que tal vez no confien aún en la EA). Además, otra ventaja de compartir alguna parte de las pruebas unitarias es que los estudiantes pueden aprender mejor cómo funcionan las pruebas unitarias. Específicamente, para la evaluación realizada en este artículo, se les proporcionó a los estudiantes algunas muestras de pruebas unitarias.

A continuación presentamos los resultados de la encuesta realizada a los alumnos que experimentaron la herramienta de EA y la tarea del caso real presentada en este artículo. Después de pedirle a la clase que completara una encuesta, respondieron 36 estudiantes. La figura 1 muestra los resultados. Lo primero que se puede observar es que no hay alumnos que tengan una opinión fuertemente negativa (color azul). Solo 2(5,56 %)/3(8,33 %) alumnos opinaron negativamente (color rojo) y las opiniones neutrales son de 4(11,11 %)/2 (5,56 %) alumnos (color amarillo). Sin embargo, los estudiantes tuvieron opiniones más positivas (color verde) y muy fuertemente positivas (color morado) (29/32 estudiantes, lo que supone un 80,56 %/88,89 %).

Además, los estudiantes que realizaron la tarea del caso real de videojuego y fueron EA mediante los test unitarios presentados en este artículo, escribieron comentarios positivos acerca de la tarea cuando se les preguntó mediante una encuesta. Entre los comentarios, los estudiantes mencionaron: "la tarea fue muy buena para llegar a ser competente en POO", "me ayudó a aprender los conceptos enseñados en clase", "la tarea es una buena manera para aprender y desarrollar habilidades", etc.

Creemos que los resultados y comentarios tan positivos de la encuesta realizada y mencionados anteriormente se deben a varias razones. En primer lugar, la tarea representa una aplicación real de videojuegos, que es más motivadora que una tarea teórica. Los estudiantes aprenden a aplicar los POO codificándolos en un problema real. Además, los test unitarios les permiten recibir valoración instantánea del trabajo realizado.

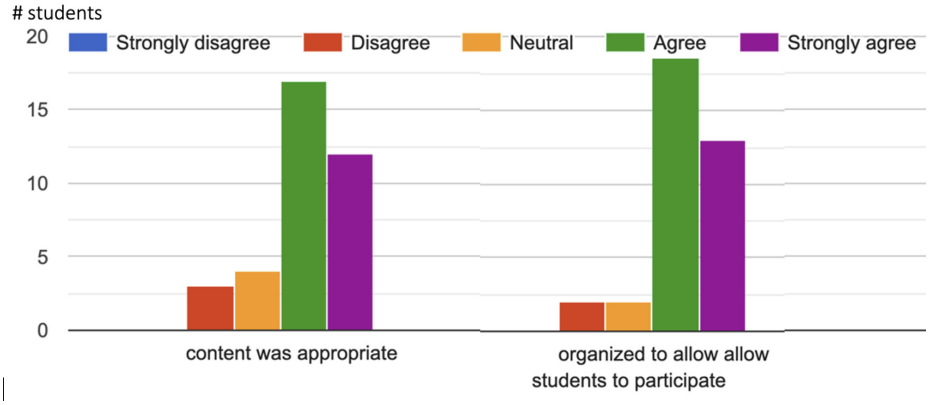

Fig. 1: Resultados de la encuesta realizada a los alumnos que evaluaron la herramienta de EA y caso de estudio.

Además, una comparación de los resultados que se logran con la EA de pruebas unitarias presentada, con respecto a la evaluación realizada por inspección visual del código de los estudiantes, confirma su efectividad.

4.2 Conclusiones

En este artículo presentamos una herramienta de EA para la enseñanza y evaluación de la POO en python. De esta forma, contribuimos a la literatura al presentar una herramienta de EA en python que evalúa conceptos de POO para la enseñanza. Además, la EA también contribuye a la retroalimentación de los estudiantes; y que los estudiantes aprendan pruebas unitarias y cómo aplicarlas a problemas reales de POO.

Los test unitarios evalúan que los detalles de la implementación de los alumnos se haya realizado correctamente. Específicamente, se comprueba que los tipos de los datos sean correctos, que los datos se encuentren dentro del rango especificado y que la modificación de los atributos sea la adecuada. Además, cuando se produce un error, el código de los estudiantes tiene que ser capaz de detectar el error y manejarlo correctamente.

En resumen, el trabajo presentado en este documento es especialmente útil para los estudiantes, ya que, suelen tener dificultades para aplicar los principios de POO a problemas reales. La evaluación realizada en un curso de segundo año de universidad, junto con las opiniones proporcionadas por los estudiantes, confirman la efectividad de nuestra propuesta.

Como trabajo futuro, consideramos que el escenario del caso real de videojuego se tendría que extender con otros personajes. Por ejemplo, arqueros, caballeros, elfos, etc. Además, se podrían implementar acciones entre ellos, por ejemplo, luchar entre ellos, formar batallones, etc. Sería muy interesante también explorar la introducción del diseño de clases con herencia, especialización, composición, agregación, sobrecarga de operadores, etc.

5 Anexo

En este anexo presentamos la función que sirve para capturar la salida estándar y comprobarla en los test unitarios. Primero, especificamos las librerias necesarias y posteriormente definimos la función.

```
from contextlib import contextmanager
   try: # Python 2
      from StringIO import StringIO
   except ImportError: # Python 3
      from io import StringIO

@contextmanager
def capt_out():
   new_o, new_e = StringIO(), StringIO()
   old_o, old_e = sys.stdout, sys.stderr
   try:
      sys.stdout, sys.stderr = new_o, new_e
      yield sys.stdout, sys.stderr
   finally:
      sys.stdout = old_o
      sys.stderr = old_e
```

Referencias bibliográficas

Barriocanal, E. G., Urbán, M.-Á. S., Cuevas, I. A., & Pérez, P. D. (2002). An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin*, *34*(4), 125-128.

Chen, W.-K., & Cheng, Y. C. (2007). Teaching object-oriented programming laboratory with computer game programming. *IEEE Transactions on Education*, *50*(3), 197-203.

Combéfis, S., & Paques, A. (2015). Pythia reloaded: An intelligent unit testing-based code grader for education. *Proceedings of the 1st International Workshop on Code Hunt Workshop on Educational Software Engineering*, 5-8.

Hsiao, I.-H., Sosnovsky, S., & Brusilovsky, P. (2009). Adaptive navigation support for parameterized questions in object-oriented programming. *European Conference on Technology Enhanced Learning*, 88-98.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli calling international conference on computing education research*, 86-93.

Ribeiro, J. C. B., Zenha-Rela, M. A., & de Vega, F. F. (2009). Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, *51*(11), 1534-1548.

Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, *4*(12), 354-357.

Torchiano, M., & Bruno, G. (2018). Integrating software engineering key practices into an oop massive in-classroom course: An experience report. *Proceedings of the 2nd International Workshop on Software Engineering Education for Millennials*, 64-71.

Torchiano, M., & Morisio, M. (2009). A fully automatic approach to the assessment of programming assignments. *The International journal of engineering education*, *25*(4), 814-829.

Touré, F., & Badri, M. (2018). Prioritizing Unit Testing Effort Using Software Metrics and Machine Learning Classifiers (S). *SEKE*, 653-652.

Wappler, S., & Wegener, J. (2006). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 1925-1932.

Whalley, J. L., & Philpott, A. (2011). A unit testing approach to building novice programmers' skills and confidence. *Proceedings of the Thirteenth Australasian Computing Education Conference*, *114*, 113-118.