รายงานฉบับสมบูรณ์

โครงการเชื่อมโยงการวิจัยภาควิชาวิศวกรรมคอมพิวเตอร์ สู่ภาคอุตสาหกรรม ปี 2547

โครงการย่อยที่สอง

A Design and Development of Test-related Metrics to Assess Software Testing Process

ห้องปฏิบัติการวิศวกรรมซอฟต์แวร์
ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์
จุฬาลงกรณ์มหาวิทยาลัย

คำนำ

เอกสารนี้เป็นรายงานวิจัยฉบับสมบูรณ์ของโครงการเชื่อมโยงการวิจัยภาควิชาวิศวกรรมคอมพิวเตอร์สู่ภาคอุตสาหกรรม ปี 2547 โครงการวิจัยร่วมภาครัฐและภาคเอกชน โครงการย่อยโครงการที่ 2: A Design and Development of Test-related Metrics to Assess Software Testing Process ของภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย คณะดำเนินการวิจัยของโครงการนี้ประกอบไปด้วย

หัวหน้าโครงการ:ผศ. ดร. ธาราทิพย์ สุวรรณศาสตร์

ผู้ช่วยวิจัย นาย ศิรส สุภาวิตา

นางสาว สุมนตรา ปัญจรัตน์

นาย เศรษฐพงศ์ ลีพัหรัตนรักษ์

งานวิจัยนี้มุ่งเน้นถึงการกำหนดข้อมูล และมาตรวัด (Metrics) ที่เกี่ยวข้องกับการทดสอบซอฟต์แวร์ เพื่อใช้ในการประเมินกระบวนการทดสอบว่ามีความก้าวหน้า ถูกดำเนินการไปมากน้อยอย่างไร หรือมีความสามารถแค่ไหน นอกเหนือไปจากการกำหนดมาตรวัดที่เกี่ยวข้องกับกระบวนการทดสอบแล้วงานวิจัยนี้ยังมุ่งเน้นไปในส่วนของการทำให้กระบวนการทดสอบสามารถเริ่มต้นได้ไปพร้อมๆ กับกระบวนการพัฒนาซอฟต์แวร์ ดังนั้นงานวิจัยได้นำแผนภาพบางแผนภาพของยูเอ็มแอลซึ่งเป็นแผนภาพที่ใช้ในการพัฒนาซอฟต์แวร์ เช่น แผนภาพคลาส (Class Diagram) แผนภาพซีเควนซ์ (Sequence Diagram) และแผนภาพยูสเคส (Use Case Diagram) มาเป็นแผนภาพที่ใช้สำหรับสร้างกรณีทดสอบ (Test case) ในกระบวนการทดสอบ หรือใช้สำหรับกำหนดมาตรวัดที่เกี่ยวข้องกับการทดสอบ

ผศ. ดร. ธาราทิพย์ สุวรรณศาสตร์

กรกฎาคม 2549

สารบัญ

สารบัญตาราง

ตาราง หน้า

สารบัญภาพ

ภาพประกอบ หน้า

บทที่ 1
รายละเอียดของโครงการ

1.1 ชื่อโครงการ

(ไทย) การออกแบบและพัฒนามาตรวัดที่เกี่ยวข้องกับการทดสอบเพื่อประเมินกระบวนการทดสอบซอฟต์แวร์

(อังกฤษ) A Design and Development of Test-related Metrics to Assess Software Testing Process

1.2 ชื่อหัวหน้าโครงการ ดร. ธาราทิพย์ สุวรรณศาสตร์

ตำแหน่ง ผู้ช่วยศาสตราจารย์

ที่ทำงาน ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

โทรศัพท์ 02-218-6975 **โทรสาร** 02-218-6955 **e-mail** taratip.s@chula.ac.th

1.3 ระยะเวลาของโครงการ

2 ปี (ตุลาคม 2546 – กันยายน 2548)

1.4 ความเป็นมาของโครงการ

การทดสอบเป็นกระบวนการที่สำคัญในการพัฒนาซอฟต์แวร์ เพื่อเป็นการตรวจสอบว่าซอฟต์แวร์ทำงานได้ถูกต้องและเป็นไปตามความต้องการของผู้ใช้ ซอฟต์แวร์ที่ไม่ได้ผ่านกระบวนการทดสอบมาอย่างดีจะก่อให้เกิดปัญหาตามมาเมื่อผู้ใช้นำซอฟต์แวร์นั้นไปใช้งาน ผู้พัฒนาซอฟต์แวร์โดยส่วนใหญ่จะดำเนินการทดสอบซอฟต์แวร์หลังจากเขียนโปรแกรมเสร็จ ซึ่งในบางครั้งอาจจะช้าไปสำหรับการดำเนินการทดสอบ หรือไม่สามารถดำเนินการทดสอบได้อย่างครบถ้วน นอกจากนั้นแล้วยังไม่สามารถตรวจสอบข้อผิดพลาดบางอย่างของซอฟต์แวร์ได้ ข้อผิดพลาดเหล่านี้อาจจะเกิดขึ้นในขั้นตอนแรกๆ ของการพัฒนาซอฟต์แวร์ เช่นขั้นตอนการวิเคราะห์ความต้องการ หรือขั้นตอนการออกแบบ เป็นต้น ดังนั้นการทดสอบควรจะเริ่มต้นดำเนินการไปพร้อมๆ กับการพัฒนาซอฟต์แวร์ นั้นคือกระบวนการทดสอบควรเป็นกระบวนการที่มีวัฏจักร (Life Cycle) เป็นของตนเอง

ในปัจจุบันมีโมเดลที่ใช้สำหรับพัฒนาซอฟต์แวร์อยู่โมเดลหนึ่งคือ Modified V-Model [1] โดยที่โมเดลนี้สามารถดำเนินกระบวนการทดสอบไปพร้อมๆ กับกระบวนการพัฒนาซอฟต์แวร์ โมเดลนี้

ยังแสดงถึงขั้นตอนต่างๆ ของกระบวนการทดสอบด้วย นอกจากนี้ Modified V-Model ยังถูกเสนออยู่ในอีกรูปแบบหนึ่งใน [2] โดยเรียกรูปแบบนี้ว่าเป็น Extended-Modified V-Model

Extended-Modified V-Model ดังกล่าวข้างต้นนั้นระบุเพียงแต่ว่าในแต่ละขั้นตอนของการพัฒนาซอฟต์แวร์ และขั้นตอนของการทดสอบจะดำเนินกิจกรรมอะไรบ้างแต่ไม่ได้ระบุว่าในขั้นตอนเหล่านั้นจะต้องเก็บข้อมูลและคำนวณมาตรวัด (Metrics) ใดบ้าง ในบางครั้งกระบวนการทดสอบอาจจะต้องถูกประเมิน (Assess) ว่ามีความก้าวหน้า และถูกดำเนินการไปมากน้อยอย่างไร หรือมีความสามารถแค่ไหน ในการติดตามความก้าวหน้าของกระบวนการทดสอบนี้อาจทำได้โดยการตรวจสอบกิจกรรมกรรมต่างๆ ที่ดำเนินการขึ้นมาจริงๆ กับกิจกรรมที่ได้วางแผนอยู่ในแผนการทดสอบ ในงานวิจัยนี้มีแนวความคิดว่า ถ้าได้มีการกำหนดข้อมูลที่จะต้องเก็บและมาตรวัดที่จะต้องคำนวณในแต่ละขั้นตอนของกระบวนการทดสอบแล้ว ก็จะสามารถตรวจสอบความก้าวหน้า หรือ ความสามารถของกระบวนการทดสอบได้

นอกเหนือไปจากการกำหนดมาตรวัดที่เกี่ยวข้องกับกระบวนการทดสอบ งานวิจัยนี้ยังมุ่งเน้นไปในส่วนของการทำให้กระบวนการทดสอบสามารถเริ่มต้นได้ไปพร้อมๆ กับกระบวนการพัฒนาซอฟต์แวร์ดังที่ระบุไว้ใน Extended-Modified V-Model ซึ่งในปัจจุบันกระบวนการพัฒนาซอฟต์แวร์นั้นได้นำเทคโนโลยีเชิงวัตถุ (Object-Oriented Technology) มาใช้ ดังนั้นในขั้นตอนของการพัฒนาซอฟต์แวร์จะต้องมีการวิเคราะห์ ออกแบบ และพัฒนาซอฟต์แวร์ให้เป็นไปตามหลักการของวิศวกรรมซอฟต์แวร์เชิงวัตถุ (Object-Oriented Software Engineering)

ในปัจจุบันยูเอ็มแอล (UML: Unified Modeling Language) [3] เป็นโมเดลที่ใช้กันอย่างแพร่หลายสำหรับการพัฒนาซอฟต์แวร์เชิงวัตถุ โมเดลนี้จะประกอบไปด้วยแผนภาพต่างๆ เพื่อใช้ในการพัฒนาซอฟต์แวร์ ดังนั้นในงานวิจัยนี้มีแนวคิดที่จะนำแผนภาพบางแผนภาพของยูเอ็มแอล เช่น แผนภาพคลาส (Class Diagram) แผนภาพซีเควนซ์ (Sequence Diagram) และแผนภาพยูสเคส (Use Case Diagram) มาเป็นแผนภาพที่ใช้สำหรับสร้างกรณีทดสอบ (Test case) ในกระบวนการทดสอบ หรือใช้สำหรับกำหนดมาตรวัดที่เกี่ยวข้องกับการทดสอบ

วัตถุประสงค์ของโครงการ

1) เพื่อศึกษาโมเดลที่ใช้ในการเพิ่มความสามารถของกระบวนการทดสอบซอฟต์แวร์ (Software Testing Process Improvement Model)

2) เพื่อกำหนดข้อมูล และออกแบบมาตรวัดที่ใช้สำหรับกระบวนการทดสอบ

3) เพื่อออกแบบวิธีการเพื่อใช้สร้างกรณีทดสอบจากโมเดลที่ใช้สำหรับการวิเคราะห์และการออกแบบซอฟต์แวร์ (เช่น ยูเอ็มแอล)

4) เพื่อพัฒนาเครื่องมือสนับสนุนการทดสอบซอฟต์แวร์

1.5 ผลงานที่คาดว่าจะได้รับเมื่อเสร็จสิ้นโครงการ

1) มาตรวัดที่เกี่ยวข้องกับกระบวนการทดสอบซอฟต์แวร์

2) วิธีการสร้างกรณีทดสอบจากโมเดลที่ใช้ในการวิเคราะห์และการออกแบบซอฟต์แวร์

3) เครื่องมือสนับสนุนกระบวนการทดสอบซอฟต์แวร์ เช่น เครื่องมือสร้างข้อมูลสำหรับใช้ทดสอบ เครื่องมือสร้างกรณีทดสอบจากแผนภาพ เป็นต้น

1.6 ประโยชน์ที่คาดว่าจะได้รับ

1) ได้มาตรวัดที่เกี่ยวข้องกับกระบวนการทดสอบซอฟต์แวร์ โดยที่มาตรวัดดังกล่าวจะเป็นสิ่งที่สะท้อนให้เห็นว่าความก้าวหน้าของกระบวนการทดสอบซอฟต์แวร์เป็นอย่างไร เพื่อเป็นแนวทางในการปรับปรุงกระบวนการทดสอบซอฟต์แวร์ของอุตสาหกรรมซอฟต์แวร์ซึ่งนำไปสู่คุณภาพของซอฟต์แวร์ที่ถูกผลิตออกมา

2) ได้วิธีการใหม่สำหรับการสร้างกรณีทดสอบจากโมเดล

3) ได้ต้นแบบเครื่องมือซอฟต์แวร์ที่ช่วยสนับสนุนการทดสอบซอฟต์แวร์ เพื่อนำไปใช้ในอุตสาหกรรมซอฟต์แวร์ และเป็นแนวทางในการพัฒนาเครื่องมือซอฟต์แวร์ที่ความสามารถเพิ่มขึ้นต่อไป

บทที่ 2
การทดสอบซอฟต์แวร์เชิงวัตถุ

2.1 การออกแบบและพัฒนาซอฟต์แวร์เชิงวัตถุ

วิธีการออกแบบและพัฒนาซอฟต์แวร์เชิงวัตถุ เป็นหลักการที่ให้แบ่งซอฟต์แวร์ออกเป็นส่วนย่อยๆ ที่เรียกว่าวัตถุ หรือ ออบเจกต์ (Object) โดยมองออบเจกต์เสมือนเป็นวัตถุในโลกจริง การทำงานของซอฟต์แวร์ จะถูกมองเปรียบเทียบเป็นการทำงานร่วมกันของออบเจกต์ภายในซอฟต์แวร์นั้น โดยออบเจกต์แต่ละตัวจะมีหน้าที่รับผิดชอบในงานเฉพาะอย่าง ด้วยวิธีการสร้างแบบจำลองสำหรับออกแบบและพัฒนาซอฟต์แวร์แบบนี้ จะช่วยให้ผู้ออกแบบซอฟต์แวร์สามารถแบ่งการทำงานของซอฟต์แวร์ได้เป็นสัดส่วน สามารถนำบางส่วนของซอฟต์แวร์กลับมาใช้ซ้ำได้ใหม่สำหรับงานที่ใกล้เคียงกัน และยังช่วยให้สามารถแก้ไขบางส่วนของซอฟต์แวร์ได้ง่าย โดยไม่มีผลกระทบต่อส่วนอื่นๆ ที่ไม่เกี่ยวข้อง

ในระดับของการออกแบบเบื้องต้นของซอฟต์แวร์เชิงวัตถุ การทำงานหน้าที่หนึ่งๆ ของซอฟต์แวร์ จะถูกนำมาวิเคราะห์ และสร้างเป็นแผนภาพ ที่แสดงการทำงานร่วมกันของออบเจกต์ (Interaction Diagram) เช่น แผนภาพซีเควนซ์ (Sequence Diagram) และแผนภาพคอลลาบอเรชัน (Collaboration Diagram) ในยูเอ็มแอล การทำงานร่วมกันของออบเจกต์จะถูกมองเปรียบเป็นการส่งเมสเสจ (Message) ระหว่างออบเจกต์ โดยออบเจกต์ตัวหนึ่งจะส่งเมสเสจไปหาออบเจกต์อีกตัวหนึ่ง เพื่อร้องขอให้ออบเจกต์ที่รับเมสเสจนั้นทำงานบางอย่างซึ่งรับผิดชอบโดยออบเจกต์นั้นๆ ดังนั้นการทำงานโดยภาพรวมของซอฟต์แวร์ที่ออกแบบด้วยวิธีเชิงวัตถุ คือกลุ่มของออบเจกต์ที่ทำงานร่วมกันโดยการส่งเมสเสจถึงกัน

2.2 ความเป็นมาของการทดสอบซอฟต์แวร์เชิงวัตถุ

ในช่วงปลายทศวรรษ 80 และต้นทศวรรษ 90 นักวิจัยได้พยายามที่จะหาคำตอบว่าคุณสมบัติเฉพาะของการพัฒนาซอฟต์แวร์ในเชิงวัตถุ ช่วยลดข้อผิดพลาดที่เกิดจากขึ้นจากผู้พัฒนาซอฟต์แวร์ได้หรือไม่ นอกจากนั้นนักวิจัยยังพยายามที่จะประยุกต์วิธีการทดสอบซอฟต์แวร์เชิงโครงสร้างเพื่อใช้ทดสอบซอฟต์แวร์เชิงวัตถุ [4] Perry และ Kaiser ได้ร่วมกันเสนอผลงานวิจัยชิ้นแรกที่ได้วิเคราะห์ถึงการทดสอบซอฟต์แวร์เชิงวัตถุ โดยการใช้วิธีเชิงรูปนัย (Formal Method) [5] ในงานวิจัยชิ้นนี้ ผู้เขียนได้วิเคราะห์ถึงคุณสมบัติของซอฟต์แวร์เชิงวัตถุ เช่นเอนแคปซูเลชัน (Encapsulation) และ การสืบทอดสมาชิก (Inheritance) ซึ่งมักจะเชื่อกันโดยการสันนิษฐานว่าช่วยลดงานของการทดสอบในบางส่วนลงไปได้ เช่น การนำกรณีทดสอบของซูเปอร์คลาสมาใช้ทดสอบซับคลาส แต่ผลจากการวิเคราะห์กลับตรงกันข้าม คุณสมบัติของซอฟต์แวร์เชิงวัตถุเหล่านี้

ไม่ได้ทำให้งานในการทดสอบลดลง กลับต้องการกรณีทดสอบเฉพาะที่จะสามารถค้นพบข้อผิดพลาดที่เกิดขึ้นจากคุณสมบัติเหล่านั้นได้ งานวิจัยหลังจากนั้น [6,7] ได้ยืนยันข้อสรุปนี้ จากงานวิจัยเหล่านี้ สามารถสรุปได้ว่า หลักการและวิธีการทดสอบสำหรับซอฟต์แวร์เชิงโครงสร้างไม่สามารถนำมาใช้กับการทดสอบซอฟต์แวร์เชิงวัตถุได้ทันที จำเป็นที่จะต้องมีการดัดแปลงวิธีการที่มีอยู่แล้ว หรือคิดค้นวิธีการขึ้นมาใหม่ เพื่อใช้กับการทดสอบซอฟต์แวร์เชิงวัตถุ

2.3 การนำกลับมาใช้ใหม่กับการทดสอบ

การนำกลับมาใช้ใหม่ (Reusability) เป็นจุดประสงค์ที่สำคัญอย่างหนึ่งของ การใช้วิธีเชิงวัตถุในการพัฒนาซอฟต์แวร์ คุณสมบัติเฉพาะในวิธีการเชิงวัตถุ เช่น การสืบทอดสมาชิก และ โพลีมอร์ฟิซึม (Polymorphism) มุ่งเน้นที่จะลดความซ้ำซ้อนในการพัฒนาซอฟต์แวร์ โดยทำให้บางส่วนของซอฟต์แวร์ที่ทำการพัฒนาขึ้น สามารถนำกลับมาใช้ใหม่ได้ แต่อย่างไรก็ตาม ดังที่ได้กล่าวในหัวข้อที่ผ่านมา คำสันนิษฐานว่าการนำกลับมาใช้ใหม่ ไม่จำเป็นต้องมีการทดสอบสำหรับกรณีที่นำมาใช้ใหม่นั้น ไม่ถูกต้อง เมื่อมีการนำซอฟต์แวร์บางส่วนกลับมาใช้ใหม่ จำเป็นที่จะต้องมีการทดสอบซอฟต์แวร์ส่วนนั้น ในการทำงานในบริบทที่นำมาใช้ใหม่ด้วย

จากข้อสรุปนี้ ทำให้ การใช้ เอนแคปซูเลชัน การสืบทอดสมาชิก และ โพลีมอร์ฟิซึม เพื่อนำบางส่วนของซอฟต์แวร์กลับมาใช้ใหม่ จึงมิได้เป็นการลดการทดสอบลง การทดสอบในกรณีที่จะนำกลับมาใช้ใหม่ ยังคงมีความจำเป็นอยู่ ดังนั้น ในการทดสอบและการออกแบบกรณีทดสอบ จำเป็นที่จะต้องพิจารณาถึงปัจจัยนี้ด้วย งานวิจัยชิ้นนี้ ในส่วนของการทดสอบในระดับบูรณาการ (Integration Testing) จะพิจารณาถึงประเด็นนี้ด้วย โดยเสนอให้มีการทดสอบซับคลาสในบริบทของ ซูเปอร์คลาส ถึงแม้ว่าซับคลาสนั้นจะสืบทอดสมาชิกที่ถูกใช้ในการทดสอบมาจากซูเปอร์คลาสซึ่งได้มีการทดสอบไปแล้ว ก็ตาม

2.4 ระดับของการทดสอบสำหรับซอฟต์แวร์เชิงวัตถุ

ระดับของการทดสอบซอฟต์แวร์ โดยทั่วไป จะถูกแบ่งออกเป็น 3 ระดับ คือ การทดสอบระดับหน่วย (Unit Testing) การทดสอบในระดับบูรณาการ และ การทดสอบระดับระบบ (System Testing) การทดสอบระดับหน่วยจะเป็นการทดสอบหน่วยย่อยที่เล็กที่สุดของซอฟต์แวร์ทีละหน่วยโดยลำพัง เพื่อค้นหาข้อผิดพลาดที่อยู่ในหน่วยซอฟต์แวร์นั้น การทดสอบในระดับบูรณาการ คือการทดสอบกลุ่มของหน่วยของซอฟต์แวร์ที่ทำงานร่วมกัน เพื่อค้นหาข้อผิดพลาดที่อยู่ในการทำงานร่วมกันเหล่านั้น เช่น ชนิดหรือขนาดของข้อมูลที่ส่งระหว่างหน่วยของซอฟต์แวร์ ที่อาจจะไม่ถูกต้องตรงกัน หลังจากที่ซอฟต์แวร์ได้ถูกพัฒนาจนสมบูรณ์แล้ว จะต้องนำมาทดสอบในระดับระบบ การทดสอบในระดับระบบจะมุ่งเน้นที่การค้นหาข้อผิดพลาดในภาพรวม ในด้านฟังก์ชันของซอฟต์แวร์

โดยมักจะไม่สนใจถึงรายละเอียดภายในซอฟต์แวร์ การทดสอบทั้งสามระดับต่างมีจุดประสงค์ที่แตกต่างกัน แต่ละระดับต่างมีความสามารถในการค้นหาข้อผิดพลาดต่างชนิดกัน

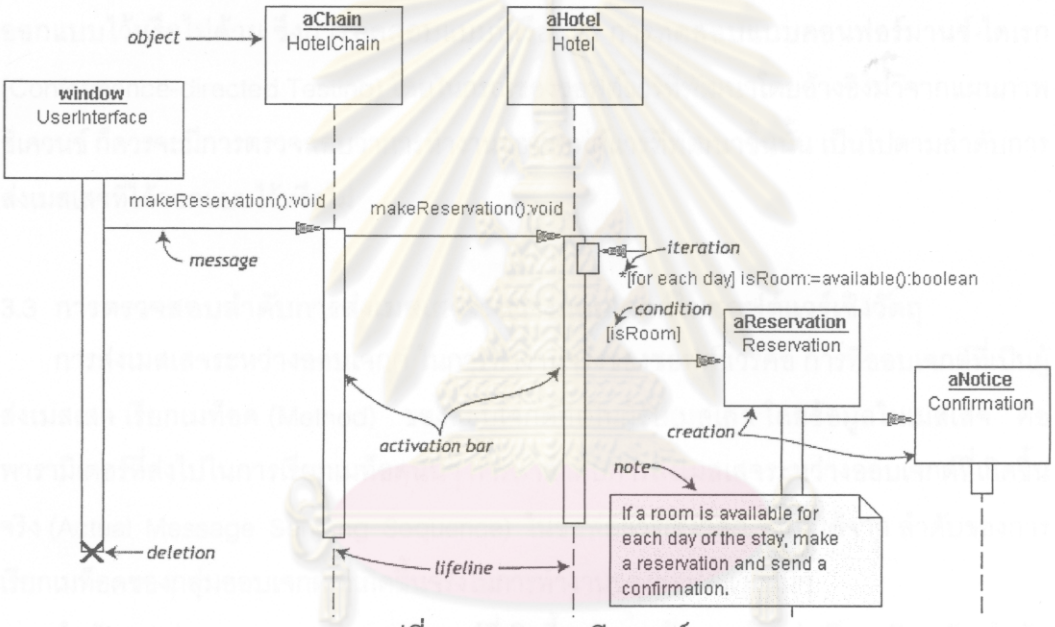
หลักการทั้งในเชิงทฤษฎีและปฏิบัติของการทดสอบซอฟต์แวร์ โดยทั่วไปแล้ว มักให้ความสำคัญกับการทดสอบในระดับหน่วย เพราะเป็นการทดสอบที่สามารถทำได้ทันทีที่หน่วยย่อยหนึ่งๆ ของซอฟต์แวร์ได้ถูกพัฒนาเสร็จ การที่สามารถทดสอบได้เร็ว ย่อมทำให้ค้นพบข้อผิดพลาดได้เร็ว ซึ่งมีผลทำให้สามารถแก้ไขข้อผิดพลาดนั้นได้แต่เนิ่นๆ และมีผลกระทบต่อการพัฒนาส่วนอื่นๆ และแผนการพัฒนาซอฟต์แวร์ น้อยกว่าการค้นพบข้อผิดพลาดในระยะท้ายๆ ของการพัฒนาซอฟต์แวร์ แต่ทว่า การทดสอบในระดับหน่วยของซอฟต์แวร์เชิงวัตถุ ไม่สามารถทำได้โดยง่าย เพราะหน่วยย่อยที่เล็กที่สุดของซอฟต์แวร์เชิงวัตถุ คือ คลาส [4, 7, 8] ไม่อาจทำการทดสอบได้โดยลำพังในหลายๆ กรณี ถึงแม้ว่าการสร้างสตับ (Stub) มาแทนออบเจกต์ของคลาสอื่นๆ ที่จำเป็นในการทำงานของคลาสที่ต้องการทดสอบ จะสามารถแก้ปัญหานี้ได้ วิธีนี้ยังคงไม่สามารถให้ผลที่น่าพอใจได้ เนื่องจากการสร้างสตับค่อนข้างเป็นการสิ้นเปลืองเพราะ นอกจากเพื่อใช้ในการทดสอบแล้ว สตับที่สร้างขึ้นไม่สามารถนำไปใช้ประโยชน์อื่นๆ ได้ นอกจากนั้น สตับในหลายๆ กรณีมีความซับซ้อนค่อนข้างมาก ซึ่งทำให้เสียเวลาในการพัฒนาสตับมาก

เนื่องจากการออกแบบซอฟต์แวร์เชิงวัตถุ เป็นการมองภาพให้เป็นการทำงานร่วมกันของวัตถุ การทดสอบกลุ่มของวัตถุที่ทำงานร่วมกัน จึงสามารถทำได้ง่ายกว่า และน่าจะค้นพบข้อผิดพลาดได้ดีกว่า การทดสอบวัตถุของคลาสใดคลาสหนึ่งโดยลำพัง งานวิจัยบางส่วน [9, 10, 11] มุ่งเน้นที่จัดลำดับการเลือกคลาสเพื่อนำมาทดสอบกลุ่มของคลาสที่ทำงานร่วมกัน โดยใช้วิธีการทดสอบบูรณาการแบบเพิ่ม เพื่อลดจำนวนสตับที่ต้องใช้ในการทดสอบ ในงานวิจัยชิ้นนี้ จะมุ่งให้ความสำคัญกับการทดสอบในระดับบูรณาการเช่นกัน โดยจะเสนอวิธีการทดสอบ และการสร้างกรณีทดสอบสำหรับกลุ่มของคลาสที่ทำงานร่วมกัน โดยจะมุ่งเน้นที่การทดสอบให้ครอบคลุมทุกๆ คลาสและซับคลาสในการทำงานแบบโพลีมอร์ฟิก

บทที่ 3
การทดสอบในระดับบูรณาการ

3.1 แผนภาพซีเควนซ์ในยูเอ็มแอล

แผนภาพซีเควนซ์ในยูเอ็มแอล ดังแสดงในรูป 3.1 ใช้สำหรับแสดงถึงการส่งเมสเสจระหว่างกันในกลุ่มของออบเจกต์ที่ทำงานร่วมกัน โดยแสดงในลักษณะของลำดับก่อนหลัง ที่เรียกว่า ลำดับการส่งเมสเสจระหว่างออบเจกต์ (Message Sending Sequence) การออกแบบและพัฒนาซอฟต์แวร์ในขั้นตอนต่อๆ ไป จะใช้แผนภาพซีเควนซ์เป็นพื้นฐาน เช่น การพัฒนาคลาส (Class) ของแต่ละออบเจกต์ให้ทำงานตามหน้าที่ความรับผิดชอบตามที่ได้ออกแบบไว้ในแผนภาพซีเควนซ์



รูปที่ 3.1 แผนภาพซีเควนซ์

หลังจากการออกแบบในขั้นต้น (Preliminary Design) ด้วยแผนภาพซีเควนซ์แล้ว เราสามารถเริ่มพัฒนาคลาสแต่ละคลาสได้ ตามหน้าที่ของคลาสที่ได้ออกแบบไว้ในแผนภาพซีเควนซ์ โดยอาจจะแยกพัฒนาโดยผู้พัฒนาหลายๆ คนพร้อมๆ กัน หลังจากที่คลาสแต่ละคลาสได้ถูกพัฒนาและทดสอบแล้ว จะต้องถูกนำมาทดสอบร่วมกันในการทดสอบในระดับบูรณาการ

3.2 การทดสอบในระดับบูรณาการโดยอ้างอิงจากแผนภาพซีเควนซ์

การทดสอบในระดับบูรณาการ มีจุดประสงค์เพื่อค้นหาข้อผิดพลาดจากการทำงานร่วมกันของหน่วยย่อยของซอฟต์แวร์ที่แต่ละหน่วยย่อยอาจจะถูกทดสอบมาแล้ว สมมติฐานของการ

ทดสอบในระดับบูรณาการคือ หน่วยย่อยของซอฟต์แวร์จะถูกทดสอบมาแล้ว แต่ถึงแม้จะถูกทดสอบมาเป็นอย่างดีแล้ว เมื่อนำมาทำงานร่วมกันกับคลาสอื่นๆ ยังอาจจะมีข้อผิดพลาดเกิดขึ้นได้ โดยส่วนใหญ่แล้วจะเกิดข้อผิดพลาดขึ้น ณ จุดที่แต่ละหน่วยย่อยต้องติดต่อรับส่งข้อมูลระหว่างกัน

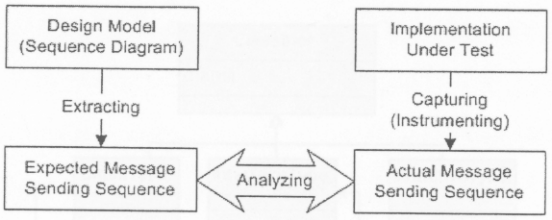
ในการพัฒนาซอฟต์แวร์เชิงวัตถุ การทดสอบในระดับบูรณาการมีความสำคัญอย่างมาก เนื่องจาก การทดสอบในระดับหน่วยย่อยให้ครอบคลุม ทำได้ยากและต้องใช้เวลามาก ซึ่งอาจจะไม่คุ้มค่ากับผลที่ได้ ดังนั้นในหลายๆ กรณี จะลดความสำคัญในการทดสอบในระดับหน่วยย่อยลงแล้วมุ่งเน้นที่การทดสอบในระดับบูรณาการ

ในการทดสอบในระดับบูรณาการ นอกจากจะต้องตรวจสอบว่าผลลัพธ์ของการทำงานตรงตามผลลัพธ์ที่คาดหวังแล้ว ควรจะต้องตรวจสอบด้วยว่าซอฟต์แวร์ที่พัฒนาขึ้นทำงานตามที่ได้ออกแบบไว้หรือไม่ด้วย ซึ่งการทดสอบแบบนี้เรียกว่า การทดสอบแบบคอนฟอร์มานซ์-ไดเรก (Conformance-directed Testing) โดยในกรณีของซอฟต์แวร์ที่พัฒนาโดยอ้างอิงมาจากแผนภาพซีเควนซ์ ก็ควรจะมีการตรวจสอบว่าการทำงานของซอฟต์แวร์ที่พัฒนาขึ้นนั้น เป็นไปตามลำดับการส่งเมสเสจที่ได้ออกแบบไว้หรือไม่

3.3 การตรวจสอบลำดับการส่งเมสเสจระหว่างออบเจกต์ในซอฟต์แวร์เชิงวัตถุ

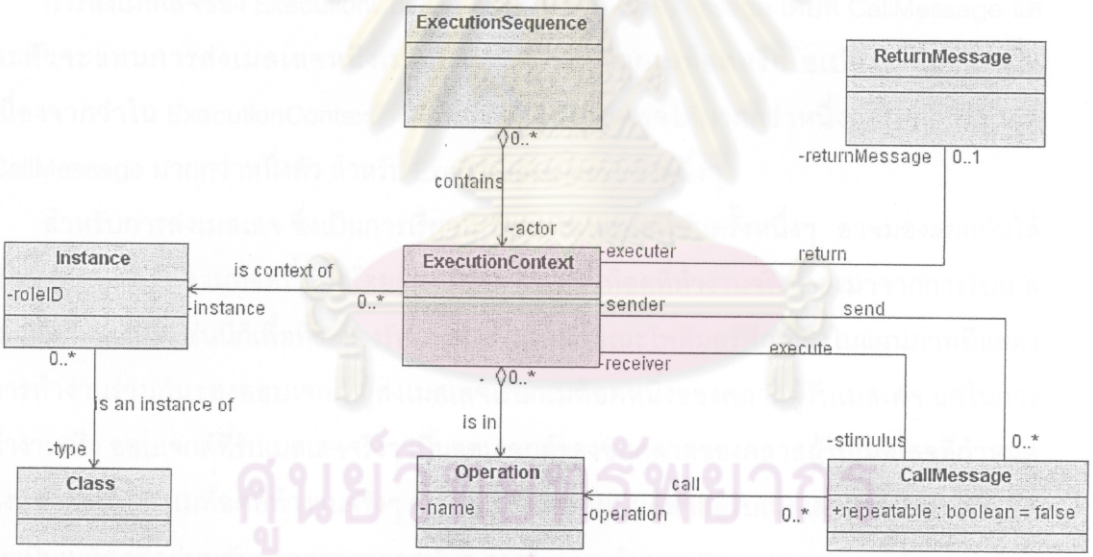
การส่งเมสเสจระหว่างออบเจกต์ ในการทำงานจริงของซอฟต์แวร์คือ การที่ออบเจกต์ที่เป็นผู้ส่งเมสเสจ เรียกเมท็อด (Method) ของออบเจกต์ที่เป็นผู้รับเมสเสจ โดยข้อมูลในเมสเสจ คือ พารามิเตอร์ที่ส่งไปในการเรียกเมท็อดนั้นๆ การหาลำดับการส่งเมสเสจระหว่างออบเจกต์ที่เกิดขึ้นจริง (Actual Message Sending Sequence) ในซอฟต์แวร์เชิงวัตถุ จึงหาได้จาก ลำดับของการเรียกเมท็อดของกลุ่มออบเจกต์ ที่เกิดขึ้นจริงในการทำงานของซอฟต์แวร์นั้นๆ

ลำดับการส่งเมสเสจระหว่างออบเจกต์ที่เกิดขึ้นจริง จะต้องถูกนำไปเปรียบเทียบกับ ลำดับการส่งเมสเสจระหว่างออบเจกต์ที่ควรจะเป็น (Expected Message Sending Sequence) ซึ่งได้มาจากแผนภาพซีเควนซ์ การเปรียบเทียบลำดับการส่งเมสเสจระหว่างออบเจกต์ นอกจากจะต้องพิจารณาถึง ตัวออบเจกต์ผู้ส่งเมสเสจ ตัวออบเจกต์ผู้รับเมสเสจ ตัวเมสเสจ และลำดับของเมสเสจแล้ว ยังต้องพิจารณาถึงปัจจัยอื่นๆ ที่อาจเกี่ยวข้อง เช่น โพลีมอร์ฟิซึม และ รีไฟน์เมนต์ (Refinement) ซึ่งปัจจัยเหล่านี้ต้องการข้อมูลมากกว่าแค่ลำดับการส่งเมสเสจระหว่างออบเจกต์เพื่อใช้พิจารณา ข้อมูลที่ต้องการเพิ่มเติมได้แก่ แผนภาพคลาสเพื่อแสดงความสัมพันธ์ของการสืบทอดสมาชิกระหว่างคลาส และกฎของการทำรีไฟน์เมนต์ รูปที่ 3.2 แสดงกระบวนการเปรียบเทียบการส่งเมสเสจดังที่ได้อธิบาย

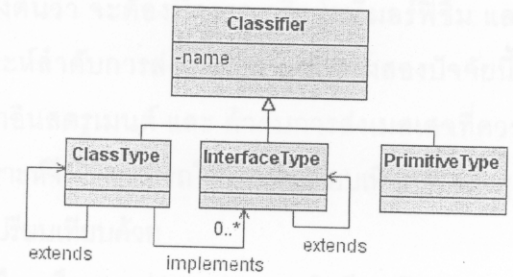


รูปที่ 3.2 กระบวนการเปรียบเทียบลำดับการส่งเมสเสจ

ลำดับการส่งเมสเสจระหว่างอ็อบเจกต์ที่เกิดขึ้นจริง จะถูกอินสตรูเมนต์ (Instrument) มาจากการทำงานของซอฟต์แวร์ ส่วนการลำดับการส่งเมสเสจระหว่างออบเจกต์ที่ควรจะเป็นจะถูกสร้างขึ้นมาจากแผนภาพซีเควนซ์ของสถานการณ์ที่ต้องการ โดยได้มีการสร้างแบบจำลองสำหรับลำดับการส่งเมสเสจทั้งสองประเภท แบบจำลองดังกล่าวแบ่งออกได้เป็นสองส่วนคือ ส่วนที่ใช้แทนลำดับการส่งเมสเสจ และส่วนที่ใช้แทนโครงสร้างของคลาส เพื่อใช้ประกอบการวิเคราะห์ลำดับการส่งเมสเสจ ดังแสดงในรูปที่ 3.3 และ 3.4 ตามลำดับ



รูปที่ 3.3 แบบจำลองสำหรับใช้แทนลำดับการส่งเมสเสจ



รูปที่ 3.4 แบบจำลองสำหรับใช้แทนโครงสร้างของคลาส

จากแบบจำลอง ExecutionContext เป็นส่วนที่ใช้แทนการทำงานของเมท็อดหรือโอเปอเรชัน (operation) ภายใต้สภาวการณ์หนึ่งๆ ในการทำงานนั้น อาจส่งเมสเสจไปยังออบเจกต์อื่นๆ ซึ่งทำให้เกิด ExecutionContext ขึ้นอีกทอดหนึ่ง ดังนั้น ExecutionContext อาจทำหน้าที่เป็น ผู้ส่งเมสเสจ ผู้รับเมสเสจ หรือทั้งสองหน้าที่ในเวลาเดียวกัน ExecutionContext สามารถเกิดขึ้นเป็นระดับได้ ตามระดับการเรียกเมท็อดของออบเจกต์ (ออบเจกต์ตัวหนึ่ง ทำงานโดยเรียกเมท็อดอีกเมท็อดหนึ่ง ซึ่งในการทำงานของเมท็อดนั้น ได้เรียกเมท็อดอีกเมท็อดหนึ่ง)

การส่งเมสเสจของ ExecutionContext จะแสดงด้วย CallMessage โดยที่ CallMessage แต่ละตัวจะแทนการส่งเมสเสจหนึ่งเมสเสจ หรือ การเรียกเมท็อดหรือโอเปอร์เรชันหนึ่งครั้ง เนื่องจากว่าใน ExecutionContext ตัวหนึ่ง อาจส่งเมสเสจได้มากกว่าหนึ่งเมสเสจ จึงอาจมี CallMessage มากกว่าหนึ่งตัว สำหรับ ExecutionContext ตัวหนึ่งๆ

สำหรับการส่งเมสเสจ ซึ่งเป็นการเรียกเมท็อดหรือโอเปอเรชันครั้งหนึ่งๆ อาจมองแยกกันได้เป็นสองมุมมอง คือ เมท็อดที่เรียกโดยผู้ส่งเมสเสจ และเมท็อดที่ทำงานจริงเนื่องมาจากการรับเมสเสจนั้น ที่ต้องเป็นเช่นนี้ก็เพื่อที่จะรองรับการทำงานในลักษณะโพลีมอร์ฟิก ที่ซึ่งในแผนภาพที่แสดงการทำงานร่วมกันของออบเจกต์ ผู้ส่งเมสเสจเรียกเมท็อดหนึ่งของคลาสผู้รับเมสเสจ แต่ในการทำงานจริง ออบเจกต์ที่รับเมสเสจจริงๆ เป็นออบเจกต์ของซับคลาสของคลาสผู้รับเมสเสจที่กำหนด ซึ่งหมายความว่าเมท็อดที่ทำงานจริงๆ จะมิใช่เมท็อดที่อยู่บนคลาสผู้รับเมสเสจที่ได้กำหนดไว้ แต่จะเป็นเมท็อดที่อยู่บนซับคลาสของคลาสนั้นๆ จากในแบบจำลอง Operation จะถูกใช้แทนเมท็อด โดยที่ ทั้ง CallMessage และ ExecutionContext ต่างอ้างอิงถึง Operation ซึ่งต่างมีความหมายแตกต่างกัน โดย Operation ที่อ้างอิงจาก CallMessage หมายถึง เมท็อดที่เรียกโดยผู้ส่งเมสเสจ แต่ Operation ที่อ้างอิงจาก ExecutionContext หมายถึง เมท็อดที่ทำงานจริง

เนื่องจากว่าในการทำงานครั้งหนึ่งๆ เป็นไปได้ว่าสำหรับคลาสหนึ่งๆ อาจมีอ็อบเจ็กต์ของคลาสนั้นหลายๆ ตัวที่ทำงานร่วมกันอยู่ในการทำงานนั้นๆ ดังนั้นเพื่อให้การเปรียบเทียบลำดับการส่งเมสเสจสามารถแยกข้อแตกต่างของการทำงานของออบเจกต์คนละตัวกันได้ จึงต้องใช้ Instance เพื่อแทนออบเจกต์แต่ละตัว และใช้ Classifier เพื่อระบุคลาสของออบเจกต์นั้น

จากที่ได้กล่าวข้างต้นว่า จะต้องพิจารณาถึง โพลีมอร์ฟิซึม และ รีไฟน์เมนต์ ในการเปรียบเทียบเพื่อวิเคราะห์ลำดับการส่งเมสเสจ เพราะทั้งสองปัจจัยนี้ มีผลทำให้ ลำดับการส่งเมสเสจที่ได้จากการทำอินสตรูเมนต์ และ ลำดับการส่งเมสเสจที่ควรจะเป็น ไม่เหมือนกันทุกประการ ดังนั้น การวิเคราะห์จึงไม่สามารถใช้วิธีการเปรียบเทียบกันตรงๆ ได้ จะต้องพิจารณาถึงทั้งสองปัจจัยนี้ร่วมในการเปรียบเทียบด้วย

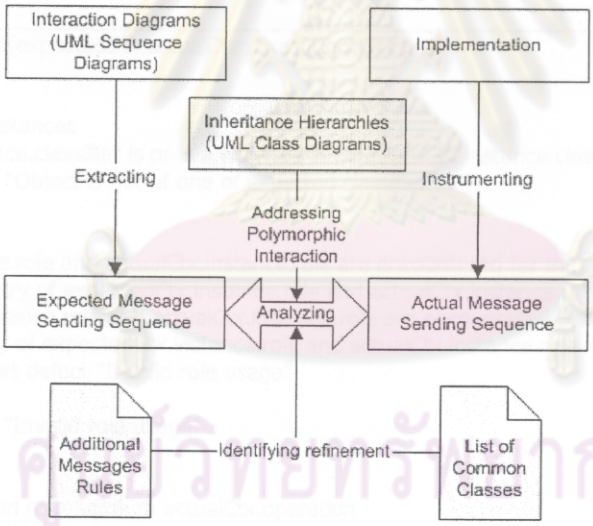
สำหรับโพลีมอร์ฟิซึม หรือ การส่งเมสเสจแบบโพลีมอร์ฟิก คลาสของออบเจกต์ที่เป็นผู้รับเมสเสจในแผนภาพซีเควนซ์ จะไม่ตรงกับ การทำงานที่เกิดขึ้นจริง เนื่องจากออบเจกต์ผู้รับเมสเสจในการทำงานจริงนั้น อาจถูกแทนที่ด้วยออบเจกต์ของซับคลาสของคลาสที่ได้กำหนดไว้แบบจำลองสำหรับใช้แทนลำดับการส่งเมสเสจ ได้ถูกออกแบบไว้เพื่อให้สนับสนุนการแทนลำดับการส่งเมสเสจที่มีการส่งเมสเสจแบบโพลีมอร์ฟิก ตามที่ได้กล่าวไว้แล้ว ดังนั้นการเปรียบเทียบจึงต้องพิจารณาว่า การส่งเมสเสจแต่ละตัว สามารถเกิดโพลีมอร์ฟิซึมได้หรือไม่ ถ้าสามารถเกิดขึ้นได้ จะต้องพิจารณาขณะเปรียบเทียบด้วยว่า ออบเจกต์ผู้รับเมสเสจสามารถเป็นออบเจกต์ของซับคลาสของคลาสที่กำหนดไว้ว่าเป็นคลาสของออบเจกต์ผู้รับเมสเสจในแผนภาพซีเควนซ์ด้วย ซึ่งในการพิจารณาการส่งเมสเสจว่าเป็นการส่งเมสเสจแบบโพลีมอร์ฟิกหรือไม่ จะต้องอาศัยข้อมูลจากลำดับชั้นการสืบทอดคลาส (Class Inheritance Hierarchy) ซึ่งได้มาจากแผนภาพคลาส

รีไฟน์เมนต์ คือ การที่ผู้พัฒนาซอฟต์แวร์ ไม่ได้สร้างซอฟต์แวร์ขึ้นมาตรงตามที่ได้ออกแบบไว้ทุกประการ แต่มีการเพิ่มเติมหรือดัดแปลงเล็กน้อย จากข้อกำหนดที่ได้ออกแบบไว้ ที่ต้องเป็นเช่นนั้น เพราะว่า บ่อยครั้งในการออกแบบซอฟต์แวร์ไม่สามารถระบุรายละเอียดของข้อกำหนดที่เกี่ยวกับตัวซอฟต์แวร์ได้ทั้งหมด โดยเฉพาะส่วนที่เกี่ยวกับรายละเอียดปลีกย่อยในการพัฒนา การวิเคราะห์ลำดับการส่งเมสเสจจึงต้องพิจารณาถึงการทำรีไฟน์เมนต์ด้วย

การทำรีไฟน์เมนต์ในแต่ละสถานการณ์ อาจมีความแตกต่างกันไปตาม วิธีการออกแบบ สภาพแวดล้อมในการทำงาน และข้อตกลงร่วมกันในการออกแบบ (Design Convention) ของแต่ละองค์กรที่พัฒนาซอฟต์แวร์ การทำรีไฟน์เมนต์โดยทั่วไป และที่จะพิจารณาในงานวิจัยนี้ จะอยู่ในลักษณะของการเพิ่มการส่งเมสเสจเข้าไป จากที่ได้กำหนดไว้ในการออกแบบ ดังนั้นในการวิเคราะห์เปรียบเทียบลำดับการส่งเมสเสจ จะต้องพิจารณาว่าหากการส่งเมสเสจใด เป็นการส่งเมสเสจที่เกิดขึ้นมาจากทำรีไฟน์เมนต์ จะต้องไม่นำการส่งเมสเสจนั้นไปพิจารณาเปรียบเทียบกับลำดับการส่งเมสเสจที่คาดหวังที่ได้มาจากแผนภาพซีเควนซ์ เนื่องจากการส่งเมสเสจนั้น ไม่ได้ถูกออกแบบไว้ในแผนภาพดังกล่าว

ในการจะพิจารณาว่าการส่งเมสเสจใด เป็นการส่งเมสเสจที่เกิดขึ้นจากการทำรีไฟน์เมนต์ จะต้องอาศัยหลักการหรือกฎเพื่อใช้ในการพิจารณา ซึ่งขึ้นอยู่กับวิธีการทำรีไฟน์เมนต์ที่ใช้ ตัวอย่างเช่น การทำรีไฟน์เมนต์อาจจะเกิดจากการเพิ่มการส่งเมสเสจ หรือเรียกเมท็อดบนคลาสที่

ให้บริการพื้นฐาน (Common Class) เช่น ข้อมูลประเภทอักขระ หรือข้อความ และข้อมูลประเภทเลขจำนวนเต็ม ซึ่งคลาสเหล่านี้ มักจะถูกละไว้จากแผนภาพที่ใช้ในการออกแบบการทำงานร่วมกัน เนื่องจากเป็นรายละเอียดที่ไม่มีความสำคัญในหลายๆ กรณี ดังนั้นหลักการวิเคราะห์ลำดับการส่งเมสเสจ จะต้องพิจารณาว่า หากการส่งเมสเสจใดเป็นผลให้มีการเรียกเมท็อดบนคลาสที่ให้บริการพื้นฐาน และการส่งเมสเสจนั้นไม่ตรงกับในลำดับการส่งเมสเสจที่คาดหวังจากแผนภาพซีเควนซ์ จะถือว่าการส่งเมสเสจนั้นเกิดขึ้นมาจากการทำรีไฟน์เมนต์ ซึ่งหมายความว่า จะไม่นำการส่งเมสเสจนั้น ไปร่วมพิจารณาเปรียบเทียบลำดับการส่งเมสเสจ อีกหลักการหนึ่ง ที่เป็นไปได้คือ การพิจารณาการส่งเมสเสจที่เป็นการเรียก ไพรเวท-เมท็อด (Private Method) ที่ไม่ตรงกับลำดับการส่งเมสเสจที่คาดหวัง ว่าเป็นการส่งเมสเสจที่เกิดจากการทำรีไฟน์เมนต์ รูปที่ 3.5 แสดงการวิเคราะห์เปรียบเทียบลำดับการส่งเมสเสจ โดยนำโพลีมอร์ฟิซึม และ รีไฟน์เมนต์ เข้ามาร่วมพิจารณาด้วย จากรูป จะเห็นได้ว่า สามารถเพิ่มเติมกฎหรือ หลักการในการพิจารณาการส่งเมสเสจที่เป็นรีไฟน์เมนต์ได้ จึงสามารถประยุกต์กระบวนการทดสอบนี้ ให้เข้ากับการทำรีไฟน์เมนต์ที่แตกต่างออกไปได้



รูปที่ 3.5 การเปรียบเทียบลำดับการส่งเมสเสจที่พิจารณาถึงโพลีมอร์ฟิซึมและรีไฟน์เมนต์

การวิเคราะห์เปรียบเทียบลำดับการส่งเมสเสจ จะเริ่มต้นที่ ExecutionSequence ของทั้งลำดับการส่งเมสเสจที่คาดหวัง และลำดับการส่งเมสเสจที่เกิดขึ้นจริง ซึ่งเป็นต้นกำเนิดของลำดับการส่งเมสเสจ โดยไล่ลงมาตามลำดับชั้นของ ExecutionContext และ CallMessage ที่เชื่อมโยงถึงกัน สำหรับแต่ละ ExecutionContext ของแต่ละลำดับการส่งเมสเสจ จะต้องพิจารณาถึง

Classifier Instance Operation CallMessage และ ReturnMessage ที่เชื่อมโยงกับ ExecutionContext นั้น โดยที่

- Classifier ในลำดับการส่งเมสเสจที่เกิดขึ้นจริง ต้องตรงกับ หรือเป็นซับคลาสของ Classifier ในลำดับการส่งเมสเสจที่คาดหวัง
- บทบาท (role) ของ Instance ในลำดับการส่งเมสเสจทั้งสอง จะต้องตรงกัน (ในขั้นตอนของการวิเคราะห์ลำดับการส่งเมสเสจ จะต้องมีการจดจำแต่ละ Instance เอาไว้ เพื่อที่จะวิเคราะห์ได้เมื่อพบ Instance นั้นอีกในระหว่างการวิเคราะห์ครั้งนั้น)
- Operation ในลำดับการส่งเมสเสจทั้งสอง จะต้องตรงกัน
- ถ้า ExecutionContext ตัวใดตัวหนึ่งมี CallMessage (มีการเรียกเมท็อดอื่น) จะต้องวิเคราะห์เปรียบเทียบ CallMessage ตามขั้นตอนที่จะกล่าวถึงหลังจากนี้
- ReturnMessage ในลำดับการส่งเมสเสจทั้งสอง จะต้องตรงกัน

ขั้นตอนการวิเคราะห์ ExecutionContext แสดงเป็นสูโดโคด (Pseudo Code) ดังในรูปที่ 3.6

```
verifyExecutionContext(expectedCtx,actualCtx: ExecutionContext)
begin

# verify classifier of instances
if not (actualCtx.instance.classifier is or is a subclass of expectedCtx.instance.classifier)
        report defect "Object is not of one of expected classifier"

# verify role usage
if expectedCtx.instance.role and actualCtx.instance.role are encountered for the first time
        put a map entry of expectedCtx.instance.role and actualCtx.instance.role in rolemap
else if expectedCtx.instance.role and actualCtx.instance.role are encountered before
        if a map entry of expectedCtx.instance.role and actualCtx.instance.role is not in rolemap
                report defect "Invalid role usage"
else
        report defect "Invalid role usage"

# verify operation
if expectedCtx.operation not match to actualCtx.operation
        report defect "Mismatched operation signature"

# verify call message
if expectedCtx.sentMessages or actualCtx.sentMessages are not empty
        verifyCallMessages(expectedCtx.sentMessages, actualCtx.sentMessages)

# verify return message
if expectedCtx.returnMessage not match actualCtx.returnMessage
        report defect "Mismatched return message"

end
```

รูปที่ 3.6 ขั้นตอนการวิเคราะห์ ExecutionContext

สำหรับแต่ละกลุ่มของ CallMessage ที่อยู่ใน ExecutionContext จะต้องนำแต่ละ CallMessage จากทั้งสองลำดับการส่งเมสเสจมาวิเคราะห์เทียบกัน โดยที่ การส่งเมสเสจในลำดับการส่งเมสเสจที่เกิดขึ้นจริง ที่เป็นผลมาจากการทำรีไฟน์เมนต์จะถูกข้ามไป หากไม่ตรงกับการส่งเมสเสจในลำดับการส่งเมสเสจที่คาดหวัง เมื่อวิเคราะห์แต่ละคู่ของ CallMessage แล้ว จะต้องวิเคราะห์แต่ละคู่ของ ExecutionContext ซึ่งแทนการทำงานที่เป็นผลจากการส่งเมสเสจ จากทั้งสองลำดับการส่งเมสเสจด้วย โดยที่ขั้นตอนในการวิเคราะห์คู่ ExecutionContext นี้ จะเป็นดังขั้นตอนที่ได้กล่าวมาแล้ว (รูปที่ 3.6) ขั้นตอนการวิเคราะห์ CallMessage แสดงเป็นสูโดโคด ดังในรูปที่ 3.7

```
verifyCallMessages(expectedMsgs,actualMsgs: CallMessage[])
begin

# verify each call message in expected call message list
foreach exptMsg in expectedMsgs
begin foreach
        # fetch call message from actual call message, skipping refinement message
        actlMsg = next actualMsgs element
        while exptMsg.operation not match to actlMsg.operation
                if actlMsg is refinement message
                        actlMsg = next actualMsgs element
                else
                        report defect "Mismatched call message"

        # verify receivers of the messages
        verifyExecutionContext(exptMsg.receiver, actlMsg.receiver)
end foreach

end
```

รูปที่ 3.7 ขั้นตอนการวิเคราะห์ CallMessage

จากหลักการที่ได้นำเสนอ ผู้วิจัยได้พัฒนาเครื่องมือเพื่ออินสตรูเมนต์การทำงานของซอฟต์แวร์ที่พัฒนาด้วยภาษาจาวา เพื่อสร้างลำดับการส่งเมสเสจระหว่างออบเจกต์ที่เกิดขึ้นจริงในการทำงานของซอฟต์แวร์ ซึ่งลำดับการส่งเมสเสจที่ได้ สามารถนำไปเปรียบเทียบกับลำดับการส่งเมสเสจที่ควรจะเป็นจากแผนภาพซีเควนซ์ได้

3.4 มาตรวัดความครอบคลุมของการทดสอบด้วยลำดับการส่งเมสเสจระหว่างออบเจกต์

จากวิธีการทดสอบด้วยลำดับการส่งเมสเสจระหว่างออบเจกต์ สามารถทำการวัดความครอบคลุมของการทดสอบได้ โดยสามารถประยุกต์หลักการและทฤษฎีเกี่ยวกับการทดสอบด้วยคอนโทรลโฟลวกราฟ (Contorl Flow Graph) มาใช้วัดความครอบคลุมได้ในหลายๆ แง่มุม เช่น การวัดความครอบคลุมโดยการประยุกต์ใช้มาตรวัดความครอบคลุมแบบทุกทางเลือก (Branch Coverage) ก็จะเป็นการวัดว่าทุกๆ ทางเลือกในแผนภาพซีเควนซ์ ได้ถูกทดสอบครบถ้วนแล้ว

งานวิจัยนี้ได้เสนอมาตรวัดสำหรับการทดสอบ ซึ่งนอกจากจะแสดงถึงความครอบคลุมของการทดสอบ ซึ่งแสดงถึงประสิทธิภาพของการทดสอบแล้ว ยังสามารถใช้ร่วมกับผลการทดสอบซึ่งรวบรวมไว้ระหว่างการทดสอบ เพื่อแสดงความก้าวหน้าในการทดสอบได้ด้วย โดยนับเป็นอัตราส่วนจำนวนคลาสและซับคลาสที่ได้ผ่านการทดสอบอย่างสมบูรณ์แล้ว เทียบกับจำนวนคลาสทั้งหมดที่จะต้องทำการทดสอบ ดังนี้

$$Progress = N_{tested} / N_{total}$$

Progress = ความก้าวหน้าในการทดสอบ

N_{tested} = จำนวนคลาสที่ผ่านการทดสอบแล้ว

N_{total} = จำนวนคลาสทั้งหมดที่ต้องทดสอบ

มาตรวัดที่งานวิจัยนี้ได้เสนอ อยู่ในรูปแบบของเงื่อนไขในการวัดความครอบคลุมเฉพาะสำหรับการทดสอบซอฟต์แวร์เชิงวัตถุ 5 แบบ ดังแสดงในตารางที่ 3.1 โดยจะพิจารณาถึงคุณสมบัติของโพลีมอร์ฟิซึม ซึ่งเป็นคุณสมบัติเฉพาะของซอฟต์แวร์เชิงวัตถุ เงื่อนไขเหล่านี้สามารถนำมาใช้สร้างมาตรวัด เพื่อแสดงว่าการทดสอบซอฟต์แวร์นั้นทำได้ครอบคลุมมากน้อยเพียงใด โดยใช้สมมติฐานที่ว่าในการทำงานร่วมกันของออบเจกต์กลุ่มหนึ่งๆ จะต้องทดสอบคลาสของออบเจกต์ในกลุ่มนั้น และอาจจะรวมถึงซับคลาสทั้งหมดของคลาสในกลุ่มนั้น ภายใต้สถานการณ์การทำงานนั้นๆ โดยมาตรวัดนี้จะแสดงว่าคลาสและซับคลาสเหล่านั้นได้ถูกทดสอบไปเป็นอัตราส่วนเท่าใดต่อจำนวนคลาสที่จะต้องทดสอบทั้งหมด

ตารางที่ 3.1 เงื่อนไขการวัดความครอบคลุมสำหรับการทดสอบซอฟต์แวร์เชิงวัตถุ

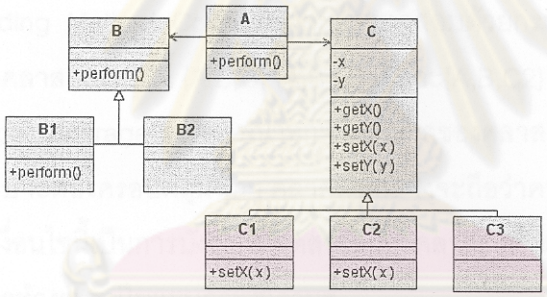
ชื่อเงื่อนไข	คำนิยามของเงื่อนไข
Base Class Coverage	ทุกๆ คลาสที่ถูกระบุอยู่ในการทำงานร่วมกันของอ็อบเจ็กนั้น ต้องถูกทดสอบ
Overriding Method Coverage	นอกจากคลาสที่ถูกระบุอยู่ในการทำงานร่วมกันของอ็อบเจ็กแล้ว ซับคลาสที่โอเวอร์ไรด์ เมท็อดที่อยู่ในการทำงานร่วมกันของอ็อบเจ็กนั้น ต้องถูกทดสอบด้วย
Inheritance Coverage	ทุกๆ คลาสที่ถูกระบุอยู่ในการทำงานร่วมกันของอ็อบเจ็กนั้น และซับคลาสทุกๆ ตัวของคลาสเหล่านี้ ต้องถูกทดสอบ

ตารางที่ 3.1 เงื่อนไขการวัดความครอบคลุมสำหรับการทดสอบซอฟต์แวร์เชิงวัตถุ (ต่อ)

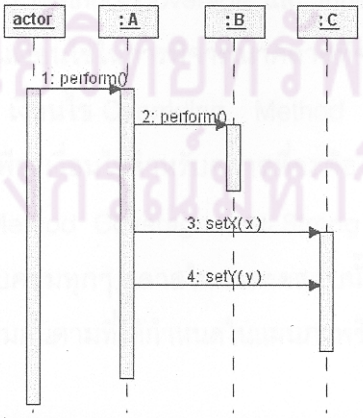
Strong Overriding Method Coverage*	คลาสที่ต้องทดสอบเหมือนกับ Overriding Method Coverage แต่ต้องทดสอบสำหรับทุกๆ ผลคูณคาร์ทีเซียนของแต่ละลำดับชั้นการสืบทอดคลาส
Strong Inheritance Coverage*	คลาสที่ต้องทดสอบเหมือนกับ Inheritance Coverage แต่ต้องทดสอบสำหรับทุกๆ ผลคูณคาร์ทีเซียนของแต่ละลำดับชั้นการสืบทอดคลาส

หมายเหตุ * เงื่อนไขนี้ใช้สำหรับการทำงานร่วมกันของอ็อบเจ็ก ที่มีคลาสที่มี Inheritance Hierarchy สองตัวขึ้นไป

ตัวอย่างการใช้เงื่อนไขทั้ง 5 นี้ จะแสดงโดยใช้ตัวอย่างแผนภาพคลาสและแผนภาพซีเควนซ์ ดังแสดงในรูปที่ 3.8 และ 3.9 ตามลำดับ



รูปที่ 3.8 แผนภาพคลาสสำหรับตัวอย่างการใช้มาตรวัดในการวัดความครอบคลุม



รูปที่ 3.9 แผนภาพซีเควนซ์สำหรับตัวอย่างการใช้มาตรวัดในการวัดความครอบคลุม

จากรูปที่ 3.9 จะเห็นได้ว่า คลาส A B และ C ต่างเป็นคลาสของอ็อบเจ็กต์ที่รับเมสเสจ เมื่อพิจารณาแผนภาพคลาสในรูปที่ 3.8 จะเห็นว่ามีเพียงคลาส B และ C เท่านั้นที่มีซับคลาส ดังนั้นการทำงานแบบโพลีมอร์ฟิกจะเกิดขึ้นเฉพาะกับการส่งเมสเสจไปหาคลาส B และ C ซึ่งตรงกับเมสเสจลำดับที่ 2 3 และ 4 ในแผนภาพซีเควนซ์ในรูปที่ 4 เท่านั้น

เงื่อนไข Base Class Coverage กำหนดความต้องการขั้นต่ำให้ทดสอบคลาสที่กำหนดในแผนภาพซีเควนซ์เท่านั้น ดังนั้นจึงต้องการกรณีทดสอบเพียงตัวเดียว เพื่อทดสอบคลาส A B และ C

สำหรับเงื่อนไข Overriding Method Coverage จะต้องพิจารณาซับคลาสของคลาสที่กำหนดไว้ในแผนภาพซีเควนซ์ด้วย โดยจะต้องทดสอบทุกๆ ซับคลาสที่โอเวอร์ไรด์เมท็อดที่ใช้ในแผนภาพซีเควนซ์นั้น ในกรณีของตัวอย่างนี้ คลาส B1 ซึ่งเป็นซับคลาสของคลาส B ได้โอเวอร์ไรด์เมท็อด perform และคลาส C1 และ C2 ซึ่งเป็นซับคลาสของคลาส C ได้โอเวอร์ไรด์เมท็อด setX ซึ่งทั้งสองเมท็อดนี้อยู่ในการทำงานในแผนภาพซีเควนซ์ ดังนั้น นอกจากคลาส A B และ C แล้วจะต้องทดสอบคลาส B1 C1 และ C2 ด้วย โดยต้องสร้างกรณีทดสอบให้คลาสเหล่านี้ทำงานแทนที่ซุเปอร์คลาสของพวกมันด้วย ดังนั้นสำหรับตัวอย่างนี้ การทดสอบเพื่อให้ได้ตามเงื่อนไขการครอบคลุมแบบ Overriding Method Coverage จะต้องทดสอบด้วยกรณีทดสอบ 3 ตัว เพื่อให้เกิดการทำงานของกลุ่มคลาสดังต่อไปนี้ {A,B,C} {A,B1,C1} และ{A,B,C2} [1]

เงื่อนไข Inheritance Coverage จะพิจารณาทุกๆ ซับคลาสของคลาสที่กำหนดไว้ในแผนภาพซีเควนซ์ โดยการทดสอบจะต้องครอบคลุมทุกๆ คลาสที่ว่านี้ จึงจะถือว่าครอบคลุมได้ตามเงื่อนไขพิจารณาในอีกแง่หนึ่ง เงื่อนไขนี้เป็นการบังคับให้ทดสอบทุกๆ คลาสในลำดับชั้นการสืบทอดคลาสของคลาสที่ถูกกำหนดว่าทำงานอยู่ในแผนภาพซีเควนซ์

เงื่อนไข Strong Overriding Method Coverage และ Strong Inheritance Coverage จะคล้ายคลึงกับเงื่อนไข Overriding Method Coverage และ Inheritance Coverage ตามลำดับ โดยที่จะมีความคล้ายคลึงกันในเงื่อนไขในการพิจารณาคลาสที่จะต้องครอบคลุมในการทดสอบ แต่มีความแตกต่างกันตรงที่ว่า เงื่อนไข Overriding Method Coverage และ Inheritance Coverage ต่างกำหนดไว้แค่เพียงเงื่อนไขสำหรับคลาสที่จะต้องถูกครอบคลุมเท่านั้น ในขณะที่เงื่อนไข Strong Overriding Method Coverage และ Strong Inheritance Coverage จะมีเงื่อนไขเพิ่มเติมว่า จะต้องครอบคลุมทุกๆ คลาสในการทดสอบนั้น โดยการจัดกลุ่มโดยใช้ผลคูณคาร์ทีเซียนของคลาสที่ทำงานร่วมกันตามที่ได้กำหนดในแผนภาพซีเควนซ์

[1] กลุ่มของคลาสที่แสดงเป็นเพียงตัวอย่างเท่านั้น การจัดกลุ่มคลาสอาจต่างไปจากตัวอย่างนี้ได้ เนื่องจากเงื่อนไขกำหนดเพียงให้ทดสอบครอบคลุมคลาสที่กำหนดเท่านั้น

ตัวอย่างการจัดกลุ่มคลาสเพื่อสร้างกรณีทดสอบให้ครอบคลุมตามเงื่อนไขแต่ละชนิด ได้แสดงไว้ในตารางที่ 3.2

ตารางที่ 3.2 ตัวอย่างการจัดกลุ่มคลาสที่ต้องครอบคลุมตามเงื่อนไข

ชื่อเงื่อนไข	กลุ่มคลาสที่ต้องครอบคลุม
Base Class Coverage	{A,B,C}
Overriding Method Coverage	{A,B,C} {A,B1,C1} {A,B,C2}
Inheritance Coverage	{A,B,C} {A,B1,C1} {A,B2,C2} {A,B,C3}
Strong Overriding Method Coverage	{A,B,C} {A,B,C1} {A,B,C2} {A,B1,C} {A,B1,C1} {A,B1,C2}
Strong Inheritance Coverage	{A,B,C} {A,B,C1} {A,B,C2} {A,B,C3} {A,B1,C} {A,B1,C1} {A,B1,C2} {A,B1,C3} {A,B2,C} {A,B2,C1} {A,B2,C2} {A,B2,C3}

เงื่อนไข และมาตรวัดเหล่านี้ได้นำเสนอในการประชุมวิชาการนานาชาติ ดังที่ได้แนบมาในภาคผนวก ข

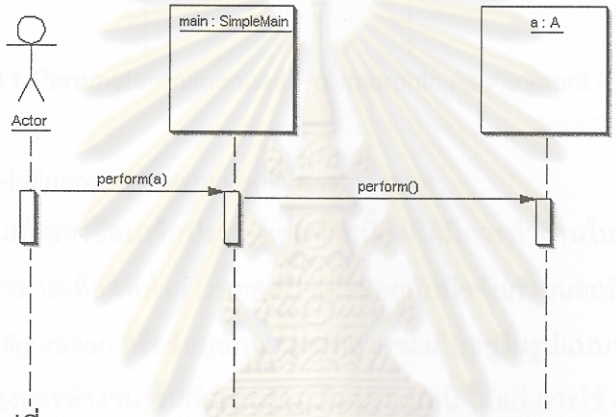
3.5 การทดสอบการทำงานแบบโพลีมอร์ฟิกในแผนภาพซีเควนซ์

เนื่องจากการทำงานแบบโพลีมอร์ฟิก ควรจะต้องมีการทดสอบทุกๆ ซับคลาสของคลาสที่เป็นส่วนหนึ่งของการทำงานของกลุ่มออบเจกต์นั้นๆ ดังนั้นการสร้างกรณีทดสอบสำหรับการทำงานของกลุ่มของออบเจกต์ที่มีการใช้คุณสมบัติโพลีมอร์ฟิซึม จะต้องพิจารณาถึงประเด็นนี้ นั่นคือต้องพยายามสร้างกรณีทดสอบให้ครอบคลุมซับคลาสด้วย

ในงานวิจัยนี้ ผู้วิจัยได้เสนอวิธีในการพิจารณาการทำงานของกลุ่มออบเจกต์ในแผนภาพซีเควนซ์ เพื่อหาเงื่อนไขของกรณีทดสอบที่สามารถทดสอบกลุ่มคลาสที่ต้องการได้ ซึ่งเป็นหัวใจที่สำคัญในการสร้างกรณีทดสอบให้ครอบคลุมทุกๆ ซับคลาส วิธีดังกล่าวนี้จะบ่งบอกว่าการทำงานของกลุ่มออบเจกต์ที่พิจารณานี้ จะเกิดการทำงานแบบโพลีมอร์ฟิกขึ้นตามปัจจัยใด โดยในงานวิจัยนี้สามารถแบ่งการทำงานของกลุ่มออบเจกต์ที่เกิดโพลีมอร์ฟิซึมออกเป็น 3 กลุ่ม ตามชนิดของปัจจัยที่ผลต่อการทำงานแบบโพลีมอร์ฟิก ปัจจัยดังกล่าวคือ พารามิเตอร์ของการทำงานที่เป็นออบเจกต์ที่รับเมสเสจที่เป็นโพลีมอร์ฟิก พารามิเตอร์ของการทำงานที่มีผลต่อการเลือกออกเจกต์ที่รับเมสเสจที่เป็นโพลีมอร์ฟิก และการตั้งค่าและสภาพแวดล้อมของการทำงานที่มีผลต่อการเลือกออปเจ็กต์ที่รับเมสเสจที่เป็นโพลีมอร์ฟิก ชนิดของปัจจัยเหล่านี้ มีดังต่อไปนี้

3.5.1 Simple Polymorphic Assignment

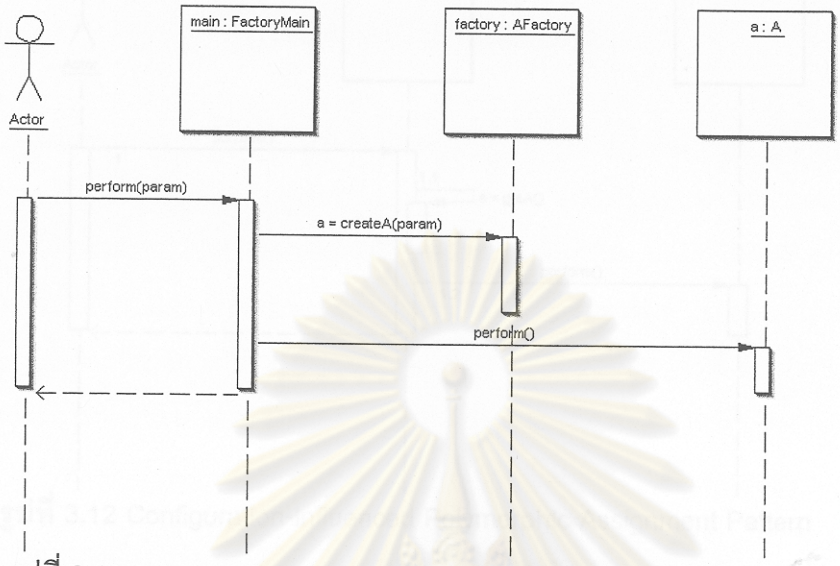
การทำงานร่วมกันของอ็อบเจ็ก ประเภทนี้ จะมีการทำงานแบบโพลีมอร์ฟิกที่ขึ้นอยู่กับชนิดของออบเจกต์ที่เป็นพารามิเตอร์ของเมท็อดที่ต้องการทดสอบโดยตรง กล่าวคือ ออบเจกต์ที่เป็นผู้รับเมสเสจที่เป็นโพลีมอร์ฟิกจะถูกส่งเข้ามาเป็นพารามิเตอร์ของเมท็อดที่ทำการทดสอบ ดังนั้นจึงสามารถควบคุมผู้รับเมสเสจเป็นออบเจกต์ของคลาสที่ต้องการได้โดยส่ง ออบเจกต์ของคลาสนั้นเข้ามาทางพารามิเตอร์ การทดสอบการทำงานร่วมกันของอ็อบเจ็กที่มีลักษณะนี้ ก็ใช้หลักการเดียวกันนี้ในการสร้างกรณีทดสอบ โดยแต่ละกรณีทดสอบจะใช้ออบเจกต์ของแต่ละคลาสที่ต้องการทดสอบเป็นพารามิเตอร์สำหรับเมท็อดที่จะทดสอบ ลักษณะของการทำงานร่วมกันของอ็อบเจ็กประเภทนี้ แสดงในรูปที่ 3.10



รูปที่ 3.10 Simple Polymorphic Assignment Pattern

3.5.2 Parameter-Influenced Polymorphic Assignment

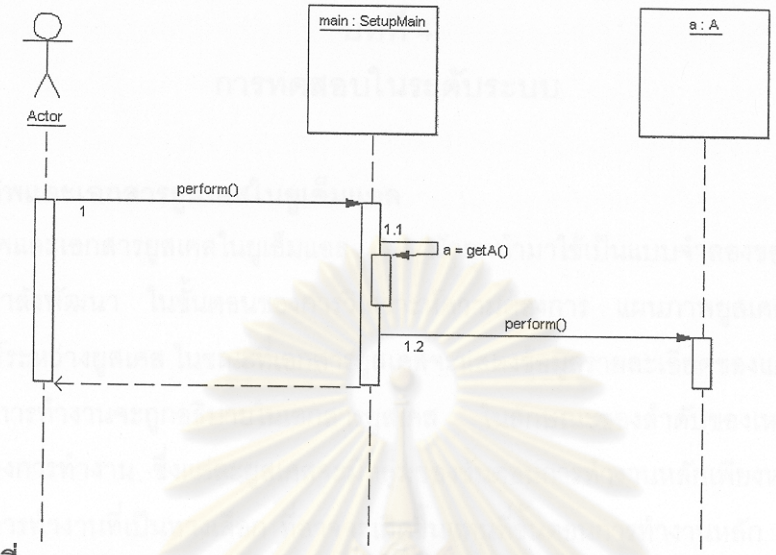
การทำงานร่วมกันของอ็อบเจ็กประเภทที่สองนี้ มีการทำงานขึ้นอยู่กับพารามิเตอร์ของเมท็อดที่จะทำการทดสอบเช่นกัน แต่เป็นผลทางอ้อม โดยที่ พารามิเตอร์ที่ส่งเข้ามายังเมท็อดนั้น จะไม่ใช่ออบเจกต์ที่รับเมสเสจที่เป็นโพลีมอร์ฟิก แต่จะเป็นค่าที่มีผลต่อการสร้าง หรือทำให้ได้มาซึ่งออบเจกต์ที่เป็นผู้รับเมสเสจที่เป็นโพลีมอร์ฟิก ตัวอย่างที่ชัดเจน เช่น กรณีการทำงานร่วมกันของอ็อบเจ็กที่เป็นไปตามหนึ่งในรูปแบบของ Factory Method pattern ที่ชื่อ parameterized factory method [12] การที่จะควบคุม หรือ ทดสอบการทำงานร่วมกันของอ็อบเจ็กประเภทนี้ จึงเป็นการปรับเปลี่ยนค่าพารามิเตอร์ที่ส่งผลนั้น ให้ส่งผลในการสร้างออบเจกต์ของคลาสที่ต้องการ วิธีการสร้างกรณีทดสอบจะคล้ายคลึงกับกรณี Simple Polymorphic Assignment แต่ทว่า อาจมีความซับซ้อนมากกว่า เนื่องจาก อาจจะไม่สามารถระบุได้อย่างชัดเจนจากแผนภาพซีเควนซ์ว่าพารามิเตอร์ใดบ้างที่ผล และมีผลอย่างไรต่อออบเจกต์ที่รับเมสเสจที่เป็นโพลีมอร์ฟิก รูปที่ 3.11 แสดง ลักษณะของการทำงานร่วมกันของอ็อบเจ็กประเภทนี้



รูปที่ 3.11 Parameter-Influenced Polymorphic Assignment Pattern

3.5.3 Configuration-Influenced Polymorphic Assignment

การทำงานร่วมกันของอ็อบเจ็กประเภทสุดท้ายนี้ ไม่ได้มีการทำงานในเชิงโพลีมอร์ฟิซึมที่ขึ้นอยู่กับพารามิเตอร์ของเมท็อดที่ทำการทดสอบ แต่ขึ้นอยู่กับปัจจัยภายนอกที่ถูกกำหนดขึ้นด้วยวิธีใดวิธีหนึ่ง เช่น configuration หรือข้อมูลภายนอกที่อาจจะเก็บอยู่ในรูปแบบของแฟ้มข้อมูลหรือฐานข้อมูล ตัวอย่างของการทำงานร่วมกันของอ็อบเจ็กประเภทนี้ ได้แก่ การใช้ configuration เป็นตัวกำหนดการเลือกใช้ซับคลาสตัวใดตัวหนึ่งในการทำงานบางอย่าง การควบคุมการทำงานร่วมกันของอ็อบเจ็กลักษณะนี้ จึงเป็นการปรับเปลี่ยนปัจจัยภายนอกเหล่านี้ให้ส่งผลถึงออบเจกต์ที่เป็นผู้รับเมสเสจที่เป็นโพลีมอร์ฟิก การทำเช่นนี้ อาจมีความยากลำบากเช่นเดียวกับในกรณี Parameter-Influenced Polymorphic Assignment เนื่องจาก จากแผนภาพซีเควนซ์ ไม่สามารถบอกได้ว่าปัจจัยใดบ้างที่มีผลต่อออบเจกต์นั้น และมีผลอย่างไร ลักษณะโดยทั่วไปของการทำงานร่วมกันของอ็อบเจ็กประเภทนี้ แสดงในรูปที่ 3.12



รูปที่ 3.12 Configuration-Influenced Polymorphic Assignment Pattern

ด้วยการควบคุมปัจจัยเหล่านี้ เราสามารถควบคุมให้การทดสอบการทำงานของกลุ่มออบเจกต์ที่เราพิจารณา ครอบคลุมถึงซับคลาสที่เราต้องการได้ ดังนั้น การสร้างกรณีทดสอบสำหรับการทำงานของกลุ่มออบเจกต์ที่เกิดโพลีมอร์ฟิซึม จึงพิจารณาถึงปัจจัยเหล่านี้เสมือนเป็นหนึ่งในข้อมูลนำเข้าสำหรับการทดสอบ จากหลักการนี้ สามารถประยุกต์ร่วมกับหลักการในการสร้างกรณีทดสอบอื่นๆ ที่มีอยู่แล้ว เพื่อสร้างกรณีทดสอบจากแผนภาพซีเควนซ์เพื่อให้ครอบคลุมทั้งเงื่อนไขความครอบคลุมของการทดสอบสำหรับการทำงานแบบโพลีมอร์ฟิกและเงื่อนไขอื่นๆ เช่น ความครอบคลุมแบบทุกทางเลือกได้ หลักการนี้ได้นำเสนอในงานประชุมวิชาการนานาชาติดังที่ได้แนบมาในภาคผนวก ค

บทที่ 4
การทดสอบในระดับระบบ

4.1 แผนภาพและเอกสารยูสเคสในยูเอ็มแอล

แผนภาพและเอกสารยูสเคสในยูเอ็มแอล มักถูกนำมาใช้เป็นแบบจำลองของการใช้งานซอฟต์แวร์ที่กำลังพัฒนา ในขั้นตอนของการวิเคราะห์ความต้องการ แผนภาพยูสเคสจะแสดงถึงความสัมพันธ์ระหว่างยูสเคส ในขณะที่เอกสารยูสเคสจะแสดงข้อมูลรายละเอียดของแต่ละยูสเคส

ขั้นตอนการทำงานจะถูกอธิบายในเอกสารยูสเคส ในลักษณะของลำดับของเหตุการณ์ที่จะเกิดขึ้นระหว่างการทำงาน ซึ่งแต่ละยูสเคสจะมีกลุ่มของขั้นตอนการทำงานหลักเพียงหนึ่งชุด และจะมีขั้นตอนการทำงานที่เป็นทางเลือก ที่อาจจะเกิดขึ้นแทนที่ขั้นตอนการทำงานหลัก ณ จุดหนึ่งๆ ซึ่งขั้นตอนการทำงานที่เป็นทางเลือกจะต้องมีเงื่อนไขกำกับว่า ด้วยเงื่อนไขอย่างไร การทำงานจึงจะเป็นไปตามขั้นตอนการทำงานที่เป็นทางเลือก แทนที่จะเป็นขั้นตอนหลัก

นอกจากเงื่อนไขของขั้นตอนการทำงานที่เป็นทางเลือกแล้ว ความสัมพันธ์ระหว่างยูสเคสในแผนภาพยูสเคสเป็นอีกหนึ่งปัจจัยที่มีผลต่อการทำงานของยูสเคส เช่น ยูสเคสหนึ่งๆ สามารถรวมการทำงานของยูสเคสตัวอื่นๆ เข้ามาได้ ในความสัมพันธ์ประเภทอินคลูด (Include) การทำความเข้าใจการทำงานของซอฟต์แวร์จากยูสเคส จะต้องพิจารณาทั้งจากขั้นตอนการทำงานของแต่ละยูสเคส และความสัมพันธ์ระหว่างยูสเคสประกอบกัน

4.2 การทดสอบโดยอ้างอิงแผนภาพและเอกสารยูสเคส

การทดสอบซอฟต์แวร์ในระดับระบบในมุมมองของฟังก์ชัน จะมุ่งเน้นที่การทดสอบความถูกต้องของการทำงาน โดยการทดสอบระดับระบบ มักจะทดสอบโดยใช้วิธีการทดสอบแบบแบล็กบ็อกซ์ (Black-Box Testing Technique) ซึ่งเป็นการทดสอบโดยพิจารณาเฉพาะคุณสมบัติภายนอกของซอฟต์แวร์ เช่น ข้อมูลนำเข้า และผลลัพธ์ของการทำงาน โดยไม่สนใจโครงสร้างภายในของซอฟต์แวร์

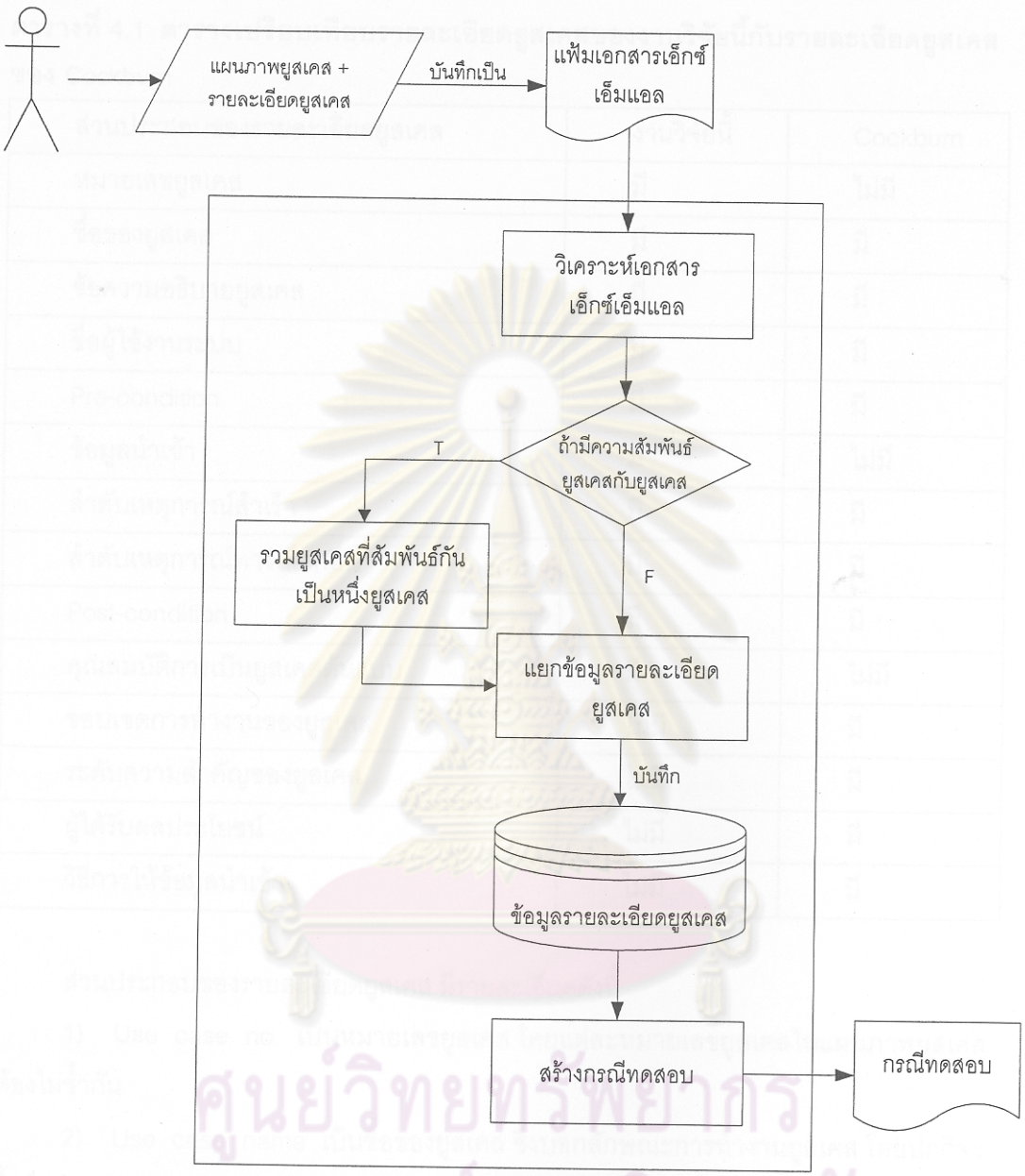
สำหรับซอฟต์แวร์ที่สร้างมาจากยูสเคส แผนภาพและเอกสารยูสเคสของซอฟต์แวร์นั้นๆ เป็นแหล่งข้อมูลที่ดีที่จะใช้ในการสร้างกรณีทดสอบสำหรับการทดสอบระดับระบบ เนื่องจากแผนภาพและเอกสารยูสเคสอธิบายการทำงานของซอฟต์แวร์ในลักษณะของการตอบสนองต่อเงื่อนไขของข้อมูลนำเข้าและสิ่งแวดล้อมอื่นๆ เป็นขั้นตอนการทำงาน กรณีทดสอบที่สร้างขึ้นเป็นการป้อนค่าข้อมูลนำเข้าตามเงื่อนไขต่างๆ ที่ระบุในยูสเคส แล้วพิจารณาผลลัพธ์ที่เกิดขึ้นเทียบกับเงื่อนไขของยูสเคส

ในงานวิจัยนี้ วิธีการสร้างกรณีทดสอบจากแผนภาพและเอกสารยูสเคสอย่างเป็นระบบได้ถูกออกแบบขึ้น ซึ่งกรณีทดสอบที่ได้จะประกอบไปด้วยข้อมูลนำเข้าที่จะใช้ในการการทดสอบ และผลลัพธ์ที่คาดว่าจะได้จากกรณีทดสอบนั้นๆ กรณีทดสอบที่สร้างขึ้นจะครอบคลุมทุกๆ เงื่อนไขของขั้นตอนการทำงานที่เป็นทางเลือก ผู้ทดสอบสามารถทดสอบซอฟต์แวร์ได้โดยทำตามขั้นตอนในกรณีทดสอบ และตรวจสอบผลลัพธ์ของการทำงานเทียบกับผลที่ควรจะได้จากกรณีทดสอบ โดยมีกระบวนการดังนี้ การเขียนแผนภาพยูสเคสและรายละเอียดยูสเคส การรวมยูสเคสที่มีความสัมพันธ์กัน และการสร้างกรณีทดสอบโดยอัตโนมัติ โดยภาพรวมของแนวคิดของงานวิจัยนี้แสดงดังรูปที่ 4.1

ขั้นตอนการทำงานเริ่มต้นจากผู้วิเคราะห์ระบบออกแบบแผนภาพยูสเคสพร้อมทั้งกำหนดรายละเอียดยูสเคส แล้วบันทึกเป็นเอกสารเอ็กซ์เอ็มแอล จากนั้นวิเคราะห์เอกสารหาความสัมพันธ์ระหว่างยูสเคสกับยูสเคส ถ้าพบความสัมพันธ์จะรวมยูสเคสเหล่านั้นเป็นหนึ่งยูสเคส แล้วแยกรายละเอียดยูสเคสลงฐานข้อมูล และสร้างกรณีทดสอบจากข้อมูลในฐานข้อมูล โดยกรณีทดสอบจะอยู่ในรูปแบบเอกสารเอ็ชทีเอ็มแอล (HTML) ประกอบด้วยข้อมูลทดสอบ และผลลัพธ์ที่คาดหวัง

4.2.1 การเขียนแผนภาพยูสเคสและรายละเอียดยูสเคส

งานวิจัยนี้ได้กำหนดรูปแบบรายละเอียดข้อมูลยูสเคสเพิ่มเติมสำหรับใช้สร้างกรณีทดสอบได้โดยอัตโนมัติ โดยมีส่วนประกอบบางส่วนซึ่งนำมาจากรายละเอียดยูสเคสของ Cockburn [5] (ได้แก่ ชื่อของยูสเคส ชื่อผู้ใช้งานระบบ Precondition ลำดับเหตุการณ์สำเร็จ ลำดับเหตุการณ์ทางเลือกอื่น และ Postcondition) และส่วนประกอบที่จำเป็นซึ่งงานวิจัยกำหนดขึ้น (ได้แก่ หมายเลขยูสเคส ข้อมูลนำเข้าสำหรับยูสเคส ประโยคเงื่อนไขของลำดับเหตุการณ์ หมายเลขประโยคเงื่อนไข หมายเลขลำดับการทำงาน และคุณสมบัติการเป็นยูสเคสต้นแบบ) ซึ่งส่วนประกอบของรายละเอียดยูสเคสของงานวิจัยนี้เปรียบเทียบกับของ Cockburn แสดงได้ดังตารางที่ 4.1



รูปที่ 4.1 ภาพรวมของแนวคิดการสร้างกรณีทดสอบจากยูสเคส

ตารางที่ 4.1 ตารางเปรียบเทียบรายละเอียดยูสเคสของงานวิจัยนี้กับรายละเอียดยูสเคสของ Cockburn

ส่วนประกอบของรายละเอียดยูสเคส	งานวิจัยนี้	Cockburn
หมายเลขยูสเคส	มี	ไม่มี
ชื่อของยูสเคส	มี	มี
ข้อความอธิบายยูสเคส	มี	มี
ชื่อผู้ใช้งานระบบ	มี	มี
Pre-condition	มี	มี
ข้อมูลนำเข้า	มี	ไม่มี
ลำดับเหตุการณ์สำเร็จ	มี	มี
ลำดับเหตุการณ์ทางเลือกอื่น	มี	มี
Post-condition	มี	มี
คุณสมบัติการเป็นยูสเคสต้นแบบ	มี	ไม่มี
ขอบเขตการทำงานของยูสเคส	ไม่มี	มี
ระดับความสำคัญของยูสเคส	ไม่มี	มี
ผู้ได้รับผลประโยชน์	ไม่มี	มี
วิธีการให้ข้อมูลนำเข้า	ไม่มี	มี

ส่วนประกอบของรายละเอียดยูสเคส มีรายละเอียดดังนี้

1) Use case no. เป็นหมายเลขยูสเคส โดยแต่ละหมายเลขยูสเคสในแผนภาพยูสเคสต้องไม่ซ้ำกัน

2) Use case name เป็นชื่อของยูสเคส ซึ่งบอกลักษณะการทำงานยูสเคส โดยปกติจะขึ้นต้นด้วยคำกริยา ตัวอย่างเช่น “Add contact” เป็นต้น

3) Description เป็นข้อความอธิบายยูสเคสเพิ่มเติม เพื่อให้เข้าใจยิ่งขึ้น

4) Actor เป็นชื่อผู้ใช้งานระบบ หรือระบบที่ติดต่อใช้งานยูสเคส ซึ่งเป็นผู้ให้ข้อมูลนำเข้าสำหรับให้ยูสเคสเริ่มต้นทำงานได้

5) Pre-condition เป็นข้อความอธิบายกิจกรรมที่ต้องปฏิบัติก่อนเริ่มการทำงานยูสเคส

6) Required-item เป็นข้อมูลนำเข้าสำหรับยูสเคสโดย Actor เป็นผู้ให้ข้อมูลเหล่านี้ ซึ่ง Required-item ประกอบด้วย

- Item name เป็นชื่อของข้อมูลนำเข้าของยูสเคส
- Item type เป็นชนิดของข้อมูลนำเข้าของยูสเคส โดยงานวิจัยนี้กำหนดให้ชนิดของข้อมูลนำเข้ามีทั้งหมด 4 ชนิดได้แก่
  - Integer เป็นข้อมูลชนิดตัวเลขจำนวนเต็ม
  - Float เป็นข้อมูลชนิดตัวเลขจำนวนจริง
  - String เป็นข้อมูลชุดอักขระ หรือข้อความ
  - Boolean เป็นข้อมูลชนิดตรรกะ มีค่าเป็นไปได้สำหรับข้อมูลชนิดนี้ 2 แบบคือ ค่าจริง (True) หรือเท็จ (False)
- Item size เป็นขนาดของข้อมูลนำเข้าของยูสเคส ซึ่งการกำหนดขนาดของข้อมูลนำเข้าขึ้นอยู่กับชนิดของข้อมูลนำเข้าดังนี้
  - ข้อมูลชนิดตัวเลขจำนวนเต็ม ให้ใส่ตัวเลขกำหนดจำนวนหลักสูงสุดที่เป็นไปได้สำหรับการสร้างข้อมูลทดสอบ ถ้าไม่ได้กำหนดตัวเลขจำนวนหลักสูงสุดแล้วจำนวนหลักของข้อมูลทดสอบจะถูกสุ่มโดยอัตโนมัติ
  - ข้อมูลชนิดตัวเลขจำนวนจริง ให้ใส่ตัวเลขกำหนดจำนวนหลักหลังจุดทศนิยมของข้อมูลทดสอบ ถ้าไม่กำหนดจำนวนหลักหลังจุดทศนิยมของข้อมูลทดสอบจะถูกสุ่มโดยอัตโนมัติ
  - ข้อมูลชุดอักขระ ให้ใส่ตัวเลขกำหนดความยาวของข้อมูลชุดอักขระ ความยาวของข้อมูลทดสอบจะถูกสุ่มโดยอัตโนมัติในกรณีที่ไม่ได้กำหนดความยาวของข้อมูลชุดอักขระ
  - ข้อมูลชนิดตรรกะไม่ต้องระบุขนาดของข้อมูลนำเข้า
- Max value เป็นค่าสูงสุดที่เป็นไปได้ของข้อมูลนำเข้าของยูสเคส ซึ่งข้อมูลทดสอบที่สุ่มได้ต้องมีค่าไม่มากกว่าค่านี้
- Min value เป็นค่าต่ำสุดที่เป็นไปได้ของข้อมูลนำเข้าของยูสเคส ซึ่งข้อมูลทดสอบที่สุ่มได้ต้องมีค่าไม่น้อยกว่าค่านี้

7) Success scenario เป็นลำดับเหตุการณ์สำเร็จของการทำงานยูสเคส ที่เกิดขึ้นหลังจากที่ผู้ใช้งานระบบให้ข้อมูลนำเข้าซึ่งสอดคล้องกับเงื่อนไขปกติของยูสเคส โดย Success scenario ต้องระบุรายละเอียดต่อไปนี้

- Condition เป็นประโยคเงื่อนไข (Condition clause) ซึ่งใช้ตรวจสอบข้อมูลนำเข้าก่อนการทำงานยูสเคส ถ้าข้อมูลนำเข้าสอดคล้องกับประโยคเงื่อนไขแล้วลำดับเหตุการณ์ทำงานของยูสเคสจะเริ่มทำงาน

- Condition no. เป็นหมายเลขของประโยคเงื่อนไข เนื่องจากในแต่ละยูสเคสมีลำดับเหตุการณ์สำเร็จเพียงหนึ่งลำดับเหตุการณ์ ดังนั้นจึงกำหนดให้หมายเลขของประโยคเงื่อนไขของลำดับเหตุการณ์สำเร็จเป็นหมายเลข 0 เสมอ

- Step เป็นหมายเลขลำดับการทำงานของยูสเคส ซึ่งการทำงานของยูสเคสอธิบายเพิ่มเติมใน Action และการเขียนหมายเลขลำดับการทำงานของยูสเคสอธิบายเพิ่มเติมในหัวข้อ 4.2.2

- Action เป็นการทำงานของยูสเคส เมื่อประโยคเงื่อนไขของลำดับเหตุกาณ์เป็นจริง โดยแต่ละลำดับเหตุการณ์สำเร็จของยูสเคสอาจมีมากกว่า 1 การทำงาน

8) Alternative scenario เป็นลำดับเหตุการณ์ทางเลือกอื่นของการทำงานยูสเคส ซึ่งเกิดขึ้นเมื่อการทำงานบางขั้นตอนของลำดับเหตุการณ์สำเร็จทำงานไม่เป็นไปตามปกติ ตัวอย่างเช่น การทำงานเพื่อจัดการข้อผิดพลาด (Exception handling) ของข้อมูลนำเข้า เป็นต้น หรือการทำงานบางขั้นตอนของลำดับเหตุการณ์สำเร็จสามารถเลือกทำงานได้มากกว่า 1 วิธี โดย Alternative scenario ต้องระบุรายละเอียดเช่นเดียวกับ Success scenario แต่แตกต่างกันตรงวิธีการกำหนดหมายเลขของประโยคเงื่อนไข และหมายเลขลำดับการทำงานของยูสเคส

9) Post-condition เป็นข้อความอธิบายผลลัพธ์หลังจากการทำงานยูสเคส ซึ่งแต่ละยูสเคสอาจมีหลายการทำงาน (ทั้งการทำงานสำเร็จ และการทำงานทางเลือกอื่น) ทำให้ผลลัพธ์อาจแตกต่างกัน ดังนั้น Post-condition จึงแยกเป็นผลลัพธ์ของแต่ละการทำงานยูสเคส โดยหมายเลขของผลลัพธ์ของลำดับเหตุการณ์สำเร็จ และเหตุการณ์ทางเลือกอื่นเป็นหมายเลขเดียวกับหมายเลขประโยคเงื่อนไขของลำดับเหตุการณ์สำเร็จ และเหตุการณ์ทางเลือกอื่นตามลำดับ

10) Is abstract เป็นคุณสมบัติที่กำหนดการเป็น abstract use case หรือยูสเคสที่เป็นต้นแบบสำหรับยูสเคสอื่น โดยในงานวิจัยนี้ระบุให้เป็น 0 เมื่อต้องการกำหนดให้เป็นยูสเคสต้นแบบ แต่ถ้าไม่ต้องการกำหนดให้เป็นยูสเคสต้นแบบให้ระบุเป็น 1 ในงานวิจัยนี้จะไม่สร้างกรณีทดสอบจากยูสเคสที่เป็นยูสเคสต้นแบบ

งานวิจัยนี้กำหนดรูปแบบรายละเอียดยูสเคสตามที่กล่าวมา ซึ่งการกำหนดหมายเลขประโยคเงื่อนไข หมายเลขลำดับของการทำงานยูสเคส และการเขียนประโยคเงื่อนไขของการทำงานยูสเคส เป็นส่วนสำคัญมากในการสร้างกรณีทดสอบ มีรายละเอียดดังนี้

1) วิธีการกำหนดหมายเลขประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่น

จากที่ได้กล่าวแล้วว่าหมายเลขประโยคเงื่อนไขของลำดับเหตุการณ์สำเร็จของการทำงานยูสเคสถูกกำหนดให้เป็นหมายเลข 0 เสมอ อันเนื่องจากในแต่ละยูสเคสมีลำดับเหตุการณ์สำเร็จได้เพียงหนึ่งลำดับเหตุการณ์ แต่การกำหนดหมายเลขประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่นนั้นแตกต่างกับของลำดับเหตุการณ์สำเร็จ โดยหมายเลขประโยคเงื่อนไขของลำดับ

เหตุการณ์ทางเลือกอื่นนี้ถูกกำหนดขึ้นเพื่อเป็นจุดอ้างอิงถึงเหตุการณ์ หรือการทำงานในลำดับเหตุการณ์สำเร็จที่ไม่สามารถทำงานได้ตามปกติอันเนื่องจากข้อมูลนำเข้า การกำหนดหมายเลขประโยคเงื่อนไขมีรูปแบบดังรูปที่ 4.2

<Order no. of Success scenario> . <Running no.>

รูปที่ 4.2 รูปแบบของหมายเลขประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่น

จากรูปที่ 4.2 รูปแบบของหมายเลขประโยคเงื่อนไขประกอบด้วยหมายเลข 2 ชุดซึ่งคั่นด้วยเครื่องหมายมหัพภาค ได้แก่

- Order no. of Success scenario เป็นหมายเลขลำดับการทำงานสำเร็จที่ไม่สามารถทำงานได้ตามปกติ หรือหมายเลขลำดับการทำงานสำเร็จที่เลือกทำงานได้มากกว่า 1 วิธี
- Running no. เป็นหมายเลขซึ่งเริ่มต้นตั้งแต่หมายเลข 1 เป็นต้นไป

2) วิธีการกำหนดหมายเลขลำดับการทำงานยูสเคส

หมายเลขลำดับการทำงานยูสเคสเป็นหมายเลขแสดงลำดับการทำงานแต่ละขั้นตอนของยูสเคส โดยต้องกำหนดหมายเลขลำดับการทำงานให้กับการทำงานแรกจนกระทั่งการทำงานสุดท้าย ซึ่งการกำหนดหมายเลขลำดับการทำงานยูสเคสแบ่งออกเป็น 2 แบบคือ

- หมายเลขลำดับการทำงานของลำดับเหตุการณ์สำเร็จ

เป็นหมายเลขลำดับการทำงานซึ่งอยู่ในลำดับเหตุการณ์สำเร็จ โดยกำหนดให้หมายเลขลำดับการทำงานสำเร็จเริ่มต้นตั้งแต่หมายเลข 1 เป็นต้นไป ดังนั้นถ้าลำดับเหตุการณ์สำเร็จประกอบด้วยการทำงาน 5 ขั้นตอน หมายเลขลำดับการทำงานจะเริ่มตั้งแต่หมายเลข 1 ถึงหมายเลข 5

- หมายเลขลำดับการทำงานของลำดับเหตุการณ์ทางเลือกอื่น

เป็นหมายเลขลำดับการทำงานซึ่งอยู่ในลำดับเหตุการณ์ทางเลือกอื่น โดยหมายเลขลำดับการทำงานทางเลือกอื่นมีรูปแบบดังรูปที่ 4.3

<Condition no.> . <Running no.>

รูปที่ 4.3 รูปแบบของหมายเลขลำดับการทำงานทางเลือกอื่น

จากรูปที่ 4.3 รูปแบบของหมายเลขลำดับการทำงานทางเลือกอื่นประกอบด้วยหมายเลข 2 ชุดคือ Condition no. ซึ่งเป็นหมายเลขประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือก

อื่นและ Running no. เป็นหมายเลขซึ่งเริ่มต้นตั้งแต่หมายเลข 1 เป็นต้นไป โดยที่คั่นด้วยเครื่องหมายมหัพภาค

3) การเขียนประโยคเงื่อนไข

ในแต่ละยูสเคสอาจมีลำดับเหตุการณ์ที่เกิดขึ้นได้หลายลำดับเหตุการณ์ การสร้างข้อมูลทดสอบเพื่อใช้ทดสอบลำดับเหตุการณ์เหล่านั้นจึงต้องกำหนดประโยคเงื่อนไขซึ่งใช้เป็นข้อกำหนดในการร่วมสร้างข้อมูลทดสอบ การเขียนประโยคเงื่อนไขของงานวิจัยนี้กำหนดให้เป็นไปตามข้อกำหนดดังนี้

- กำหนดให้ข้อมูลนำเข้าที่ระบุไว้ใน required-item 1 ตัวคือตัวแปร 1 ตัวแปร
- ชื่อตัวแปรในประโยคเงื่อนไขต้องถูกระบุไว้ใน required-item และต้องสะกดเหมือนกัน
- แต่ละพจน์ในประโยคเงื่อนไขต้องเขียนไว้ภายในวงเล็บเท่านั้น เช่น (card_id==1234) เป็นต้น
- เครื่องหมายเปรียบเทียบที่ใช้ในประโยคเงื่อนไขมีดังตารางที่ 4.2

ตารางที่ 4.2 เครื่องหมายเปรียบเทียบที่ใช้ในประโยคเงื่อนไข

เครื่องหมาย	คำอธิบาย	ชนิดข้อมูลของนำเข้า			
		Integer	Float	String	Boolean
==	เท่ากับ	ใช้ได้	ใช้ได้	ใช้ได้	ใช้ได้
<>	ไม่เท่ากับ	ใช้ได้	ใช้ได้	ใช้ได้	ใช้ไม่ได้
>	มากกว่า	ใช้ได้	ใช้ได้	ใช้ไม่ได้	ใช้ไม่ได้
>=	มากว่าหรือเท่ากับ	ใช้ได้	ใช้ได้	ใช้ไม่ได้	ใช้ไม่ได้
<	น้อยกว่า	ใช้ได้	ใช้ได้	ใช้ไม่ได้	ใช้ไม่ได้
<=	น้อยกว่าหรือเท่ากับ	ใช้ได้	ใช้ได้	ใช้ไม่ได้	ใช้ไม่ได้

- การระบุขนาดความยาวของข้อมูลนำเข้าชนิดชุดอักขระ ให้เขียนชื่อข้อมูลนำเข้าตามด้วยเครื่องหมายจุลภาค และตามด้วยคำว่า length เช่น (FirstName.length <= 20) เป็นต้น
- ค่าของข้อมูลนำเข้าชนิดชุดอักขระ ต้องกำหนดอยู่ภายในเครื่องหมายอัญประกาศ (") เท่านั้น เช่น (UnivName=="chula") เป็นต้น

- สำหรับข้อมูลนำเข้าชนิดตรรกะ กำหนดให้แทนค่าจริงด้วย True และแทนค่าเท็จด้วย False
- ถ้าประโยคเงื่อนไขมีจำนวนพจน์มากกว่า 1 พจน์สามารถเชื่อมพจน์ได้ 2 แบบคือ แบบและ (&&) และแบบหรือ (||) เช่น ((X<=0)||(X>=10)) เป็นต้น
- กำหนดให้ใช้เครื่องหมายทางคณิตศาสตร์ในประโยคเงื่อนไขได้ดังนี้บวก (+) ลบ (-) คูณ (*) และหาร (/)
- แต่ละพจน์ต้องประกอบด้วยชื่อข้อมูลนำเข้าหรือการคำนวณทางคณิตศาสตร์ เครื่องหมายเปรียบเทียบ และค่าขอบเขตของข้อมูลนำเข้า ตัวอย่างเช่น (Height – Weight < 110) เป็นต้น
- จำนวนวงเล็บในประโยคเงื่อนไขต้องไม่เกิน 5 วงเล็บ และวงเล็บต้องซ้อนกันอย่างถูกต้อง เช่น ((X<=0) || (X>=11)) วงเล็บของประโยคเงื่อนไขนี้ถูกต้องเพราะมีจำนวนวงเล็บ 3 วงเล็บและซ้อนกันอย่างถูกต้อง แต่ ((X<=0) || (X>=11) วงเล็บของประโยคเงื่อนไขนี้ไม่ถูกต้องเพราะมีวงเล็บเปิด 3 อันแต่มีวงเล็บปิด 2 อัน ส่วน ((X<=0) (&& X>=11)) แม้ว่าวงเล็บของประโยคเงื่อนไขนี้มีวงเล็บ 3 วงเล็บ และซ้อนกันอย่างถูกต้อง แต่ตำแหน่งของวงเล็บไม่ถูกต้อง เป็นต้น
- ประโยคเงื่อนไขมีพจน์ที่เป็นการคำนวณทางคณิตศาสตร์ได้เพียงพจน์เดียวเท่านั้น เช่น ((Weight > 30) && (Height – Weight < 110)) เป็นต้น

4) การกำหนดจุดเชื่อมโยงยูสเคส

ความสัมพันธ์ระหว่างยูสเคสกับยูสเคสในแผนภาพยูสเคสมี 2 แบบคือความสัมพันธ์แบบอินคลูด และความสัมพันธ์แบบเอ็กซ์เทน ซึ่งแสดงด้วยเส้นเชื่อมโยงระหว่างยูสเคสหนึ่งไปอีกยูสเคสหนึ่งโดยแสดงชนิดความสัมพันธ์บนเส้นเชื่อมโยง

เอกสารข้อกำหนดของยูเอ็มแอล (UML Specification) [6] ระบุว่ายูสเคสที่มีความสัมพันธ์กันแบบเอ็กซ์เทนต้องมีการกำหนดจุดอ้างอิง (Extension point) ไว้ที่ยูสเคสหลัก (Base use case) เพื่อให้ยูสเคสที่เอ็กซ์เทนยูสเคสหลักอ้างอิงถึงขั้นตอนการทำงานในยูสเคสหลักได้ โดยจุดอ้างอิงในยูสเคสอาจมีได้หลายตำแหน่งในลำดับเหตุการณ์ที่เกิดขึ้นตามที่ผู้ออกแบบกำหนด ในงานวิจัยนี้ไม่ได้มีการใช้จุดอ้างอิงเพื่อให้ยูสเคสอื่นอ้างอิงถึงตามเอกสารข้อกำหนดของยูเอ็มแอล เนื่องจากรูปแบบของรายละเอียดยูสเคสและวิธีการสร้างกรณีทดสอบโดยอัตโนมัติที่งานวิจัยนี้นำเสนอไม่รองรับกับการกำหนดจุดอ้างอิง ดังนั้นงานวิจัยนี้ออกแบบจุดเชื่อมโยงยูสเคสเพื่อระบุว่ายูสเคสที่ต้องการเชื่อมโยง โดยจุดเชื่อมโยงจะระบุอยู่ในลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสหลักเพื่อให้ยูสเคสที่เชื่อมโยงทำงานแทนยูสเคสหลัก

การกำหนดจุดเชื่อมโยงความสัมพันธ์ของยูสเคสในรายละเอียดยูสเคสของงานวิจัยนี้ทำได้ดังนี้

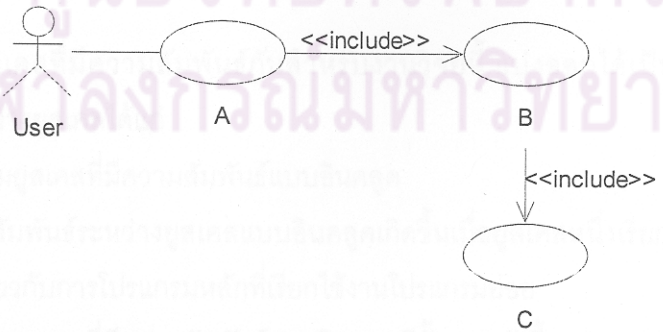
- จุดเชื่อมโยงยูสเคสสำหรับความสัมพันธ์แบบอินคลูด

สำหรับความสัมพันธ์แบบอินคลูด งานวิจัยนี้กำหนดให้จุดเชื่อมโยงไปยังยูสเคสอื่นสามารถกำหนดได้เฉพาะในลำดับเหตุการณ์สำเร็จเท่านั้น การเขียนจุดเชื่อมโยงทำได้โดยเขียนเครื่องหมายปีกกาเปิด และ “UC” แล้วระบุหมายเลขยูสเคสที่ต้องการเรียกใช้ และปิดด้วยเครื่องหมายปีกกาปิด สมมติว่าการทำงานข้อที่ 1 ของลำดับเหตุการณ์สำเร็จมีการเรียกใช้งานยูสเคสหมายเลข 2 จึงเขียนจุดเชื่อมโยงเป็น “{UC2}”

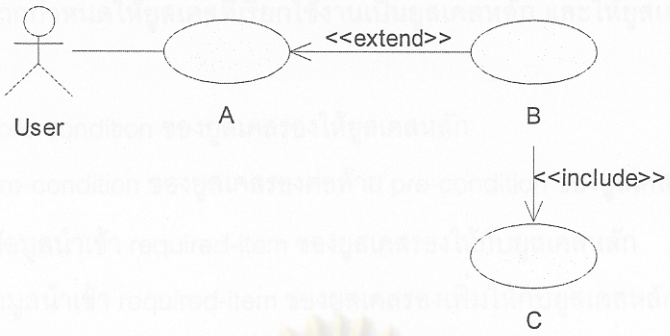
- จุดเชื่อมโยงยูสเคสสำหรับความสัมพันธ์แบบเอ็กซ์เทน

การเขียนจุดเชื่อมโยงสำหรับความสัมพันธ์แบบเอ็กซ์เทนมีรูปแบบเช่นเดียวกับการเขียนจุดเชื่อมโยงสำหรับความสัมพันธ์แบบอินคลูด โดยงานวิจัยนี้กำหนดให้จุดเชื่อมโยงสำหรับความสัมพันธ์แบบเอ็กซ์เทนสามารถกำหนดได้เฉพาะในลำดับเหตุการณ์ทางเลือกอื่นเท่านั้น และไม่ต้องระบุประโยคเงื่อนไขสำหรับลำดับเหตุการณ์ทางเลือกอื่นที่มีจุดเชื่อมโยง สมมติว่าการกำหนดจุดเชื่อมโยงเพื่อเรียกใช้งานยูสเคสหมายเลข 3 จึงเขียนจุดเชื่อมโยงเป็น “{UC3}” และไม่ระบุประโยคเงื่อนไข

สำหรับงานวิจัยนี้กำหนดให้ความสัมพันธ์ระหว่างยูสเคสกับยูสเคสต้องไม่ซ้อนกัน กล่าวคือถ้ายูสเคสหนึ่งเรียกใช้งานอีกยูสเคสหนึ่ง แล้วยูสเคสที่ถูกเรียกใช้งานต้องไม่มีการเรียกใช้งานยูสเคสอื่นดังตัวอย่างรูปที่ 4.4 ยูสเคส B ถูกเรียกใช้โดยยูสเคส A แต่ยูสเคส B มีการเรียกใช้ยูสเคส C ซึ่งทำให้เกิดการซ้อนกัน หรือยูสเคสที่ถูกเรียกใช้ซึ่งความสัมพันธ์กันแบบเอ็กซ์เทนต้องไม่มีลำดับการทำงานทางเลือกอื่น และไม่มีการเรียกใช้งานยูสเคสอื่นดังตัวอย่างรูปที่ 4.5 ยูสเคส B ถูกเรียกใช้โดยยูสเคส A ความสัมพันธ์แบบเอ็กซ์เทน แต่ยูสเคส B มีการเรียกใช้งานยูสเคส C ซึ่งทำให้เกิดการซ้อนเช่นกัน



รูปที่ 4.4 ภาพแสดงยูสเคสที่มีความสัมพันธ์ที่ซ้อนกันแบบที่ 1



รูปที่ 4.5 ภาพแสดงยูสเคสที่มีความสัมพันธ์ที่ซ้อนกันแบบที่ 2

ก่อนการสร้างกรณีทดสอบจะต้องรวมยูสเคสที่มีความสัมพันธ์กันให้เป็นหนึ่งยูสเคสแล้วจึงสามารถสร้างกรณีทดสอบได้ ซึ่งการรวมยูสเคสที่มีความสัมพันธ์ได้กล่าวไว้ในหัวข้อ 4.2.2 ต่อไป

4.2.2 การรวมยูสเคสที่มีความสัมพันธ์กัน

ในขั้นตอนการวิเคราะห์นักวิเคราะห์ระบบมักเริ่มต้นเขียนแผนภาพยูสเคสแบบง่ายแสดงการทำงานหลักของระบบก่อน กล่าวคือแผนภาพยูสเคสจะประกอบด้วยผู้ใช้งานระบบ ยูสเคสของการทำงานหลัก และเส้นแสดงความสัมพันธ์ระหว่างยูสเคสกับผู้ใช้งานระบบ หลังจากผ่านขั้นตอนการวิเคราะห์ระบบแล้วนักวิเคราะห์ระบบจะปรับให้แผนภาพยูสเคสมีความละเอียดยิ่งขึ้นซึ่งเรียกว่ากระบวนการนี้ว่า Refinement โดยการเพิ่มยูสเคสของการทำงานย่อย และความสัมพันธ์ระหว่างยูสเคสเพื่อให้เข้าใจยิ่งขึ้น ซึ่งเป็นประโยชน์ต่อการพัฒนาระบบ

จากที่ได้กล่าวมาสังเกตได้ว่าแผนภาพยูสเคสที่ทำการปรับแก้ไขแล้วนั้นมีที่มาจากแผนภาพยูสเคสอย่างง่าย ดังนั้นก่อนการสร้างกรณีทดสอบจึงต้องรวมความสัมพันธ์ระหว่างยูสเคสกับยูสเคสเข้าด้วยกันก่อนเพื่อแปลงให้เป็นแผนภาพยูสเคสอย่างง่ายที่มีเฉพาะยูสเคสของการทำงานหลัก

การรวมยูสเคสที่มีความสัมพันธ์กันสำหรับงานวิจัยนี้ แบ่งออกได้เป็น 2 แบบตามความสัมพันธ์ระหว่างยูสเคสได้แก่

1) การรวมยูสเคสที่มีความสัมพันธ์แบบอินคลูด

ความสัมพันธ์ระหว่างยูสเคสแบบอินคลูดเกิดขึ้นเมื่อยูสเคสหนึ่งเรียกใช้งานอีกยูสเคสหนึ่งในลักษณะเดียวกับการโปรแกรมหลักที่เรียกใช้งานโปรแกรมย่อย

การรวมยูสเคสที่มีความสัมพันธ์แบบอินคลูดมีขั้นตอนดังนี้

- กำหนดยูสเคสหลักและยูสเคสรอง

เริ่มจากกำหนดให้ยูสเคสที่เรียกใช้งานเป็นยูสเคสหลัก และให้ยูสเคสที่ถูกเรียกใช้งานเป็นยูสเคสรอง

- เพิ่ม pre-condition ของยูสเคสรองให้ยูสเคสหลัก

นำ pre-condition ของยูสเคสรองต่อท้าย pre-condition ของยูสเคสหลัก

- เพิ่มข้อมูลนำเข้า required-item ของยูสเคสรองให้กับยูสเคสหลัก

นำข้อมูลนำเข้า required-item ของยูสเคสรองเพิ่มให้กับยูสเคสหลักเพื่อเป็นข้อมูลนำเข้าของยูสเคสหลักด้วย

- รวมประโยคเงื่อนไขของลำดับเหตุการณ์สำเร็จ

นำประโยคเงื่อนไขของลำดับเหตุเหตุการณ์สำเร็จของยูสเคสรองรวมกับประโยคเงื่อนไขของลำดับเหตุการณ์สำเร็จของยูสเคสหลัก โดยเชื่อมตัวดำเนินการทางตรรกะ “และ (&&)”

- เพิ่มการทำงานของลำดับเหตุการณ์สำเร็จของยูสเคสรองให้ยูสเคสหลัก

เพิ่มการทำงานของลำดับเหตุการณ์สำเร็จของยูสเคสรองให้กับการทำงานของลำดับเหตุการณ์สำเร็จของยูสเคสหลัก สำหรับการทำงานใหม่ให้แทรกแทนจุดเชื่อมโยงไปยังยูสเคสรอง โดยหมายเลขลำดับของการทำงานใหม่ให้ขึ้นต้นด้วยหมายเลขลำดับของการทำงานของจุดเชื่อม แล้วตามด้วยเครื่องหมายยติภังค์ (“-“) และหมายเลขลำดับของการทำงานเดิมของลำดับเหตุการณ์สำเร็จของยูสเคสรอง

- เพิ่มลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสรองให้ยูสเคสหลัก

เพิ่มลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสรองให้ยูสเคสหลักเพื่อเป็นลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสหลัก โดยหมายเลขของประโยคเงื่อนไขใหม่ของลำดับเหตุการณ์ทางเลือกอื่นให้ขึ้นต้นด้วยหมายเลขลำดับการทำงานของจุดเชื่อมโยงไปยูสเคสรอง แล้วตามด้วยเครื่องหมาย “-” และหมายเลขลำดับของการทำงานเดิมของลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสรอง และหมายเลขลำดับการทำงานใหม่ของลำดับเหตุการณ์ทางเลือกอื่นมีรูปแบบเช่นเดียวกับหมายเลขของประโยคเงื่อนไขใหม่

- เพิ่ม post-condition ของลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสรองให้กับยูสเคสหลัก

นำ post-condition ของลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสรองเพิ่มให้กับยูสเคสหลัก โดยระบุหมายเลขของผลลัพธ์เป็นหมายเลขของประโยคเงื่อนไขใหม่ของลำดับเหตุการณ์ทางเลือกอื่น

2) การรวมยูสเคสที่มีความสัมพันธ์แบบเอ็กซ์เทน

ความสัมพันธ์ระหว่างยูสเคสแบบเอ็กซ์เทนเกิดขึ้นเมื่อยูสเคสหนึ่งไม่สามารถทำงานได้ตามปกติ จึงเรียกใช้งานอีกยูสเคสหนึ่งเพื่อทำงานแทน

การรวมยูสเคสที่มีความสัมพันธ์แบบเอ็กซ์เทนมีขั้นตอนดังนี้

- กำหนดยูสเคสหลักและยูสเคสรอง

เริ่มจากกำหนดให้ยูสเคสที่เรียกใช้งาน (ยูสเคสที่เอ็กซ์เทน) เป็นยูสเคสหลัก และให้ยูสเคสที่ถูกเรียกใช้งาน (ยูสเคสที่ถูกเอ็กซ์เทน) เป็นยูสเคสรอง

- เพิ่ม pre-condition ของยูสเคสรองให้ยูสเคสหลัก

นำ pre-condition ของยูสเคสรองต่อท้าย pre-condition ของยูสเคสหลัก

- เพิ่มข้อมูลนำเข้า required-item ของยูสเคสรองให้กับยูสเคสหลัก

นำข้อมูลนำเข้า required-item ของยูสเคสรองเพิ่มให้กับยูสเคสหลักเพื่อเป็นข้อมูลนำเข้าของยูสเคสหลักด้วย

- เพิ่มประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่น

นำประโยคเงื่อนไขของลำดับเหตุการณ์สำเร็จของยูสเคสรองกำหนดให้กับประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่นที่จุดเชื่อมโยงยูสเคส และใช้หมายเลขของประโยคเงื่อนไขของเหตุการณ์ทางเลือกอื่นเป็นหมายเลขของประโยคเงื่อนไขใหม่

- นำการทำงานของลำดับเหตุการณ์สำเร็จของยูสเคสรองเป็นลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสหลัก

โดยนำการทำงานของลำดับเหตุการณ์สำเร็จของยูสเคสรองแทนที่การทำงานของลำดับเหตุการณ์ทางเลือกอื่นที่มีจุดเชื่อมโยงยูสเคส โดยหมายเลขของการทำงานใหม่ขึ้นต้นด้วยหมายเลขของประโยคเงื่อนไข แล้วตามด้วยเครื่องหมายจุลภาค และหมายเลขตั้งแต่ 1 เป็นต้นไป

- เพิ่ม post-condition ของลำดับเหตุการณ์สำเร็จของยูสเคสรองให้กับยูสเคสหลัก

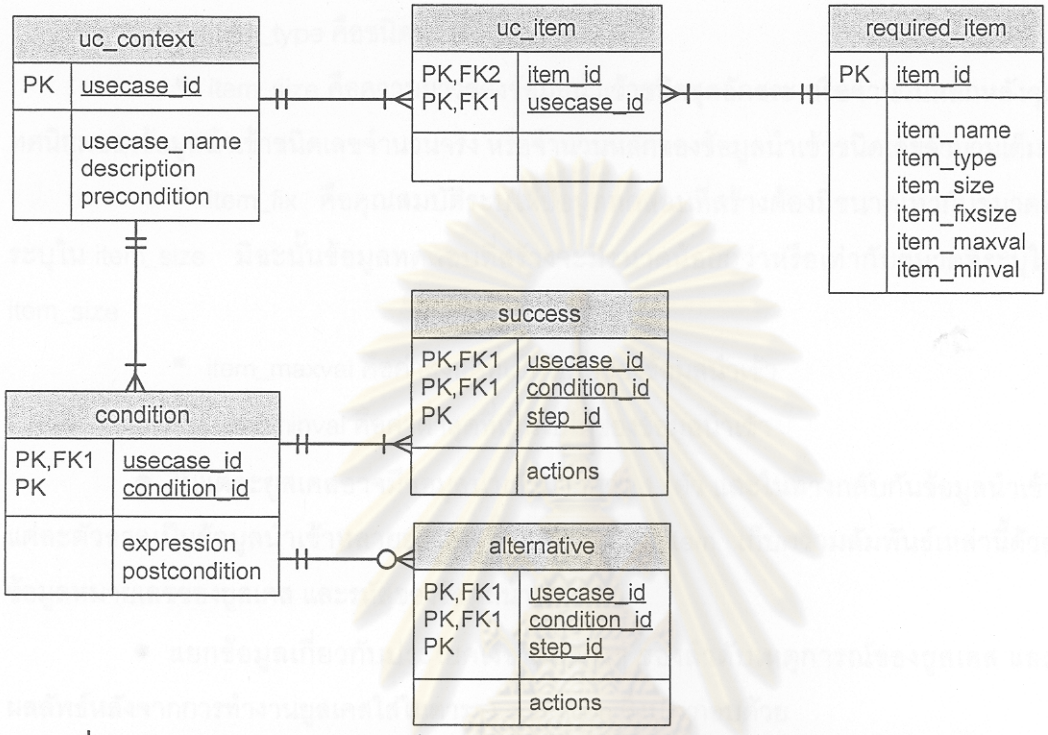
นำ post-condition ของลำดับเหตุการณ์สำเร็จของยูสเคสรองเพิ่มให้กับยูสเคสหลัก โดยนำมาใช้เป็นผลลัพธ์ของลำดับเหตุการณ์ทางเลือกอื่น ดังนั้นหมายเลขของผลลัพธ์เป็นหมายเลขของประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่น

4.2.3 การสร้างกรณีทดสอบโดยอัตโนมัติ

หลังจากกำหนดรูปแบบรายละเอียดยูสเคสเพื่อใช้ในการสร้างกรณีทดสอบ และวิธีการรวมยูสเคสที่มีความสัมพันธ์กัน จากนั้นเข้าสู่ขั้นตอนการสร้างกรณีทดสอบจากยูสเคสที่เตรียมไว้ ซึ่งแบ่งขั้นตอนหลักออกเป็น 2 ส่วนได้แก่ การแยกรายละเอียดยูสเคสลงฐานข้อมูล และการสร้างกรณีทดสอบ

1) การแยกรายละเอียดยูสเคสลงฐานข้อมูล

ขั้นตอนนี้เป็นการแยก และเตรียมข้อมูลรายละเอียดยูสเคสที่จำเป็นสำหรับการสร้างกรณีทดสอบ โดยข้อมูลที่ได้จะถูกจัดเก็บในฐานข้อมูล โครงสร้างของฐานข้อมูลที่ใช้จัดเก็บแสดงได้ดังรูปที่ 4.6



รูปที่ 4.6 แผนภาพความสัมพันธ์ของเอนทิตี (ER-Diagram) ของรายละเอียดยูสเคส

จากรูปที่ 4.6 ซึ่งเป็นแผนภาพแสดงความสัมพันธ์ของข้อมูลรายละเอียดยูสเคสซึ่งแสดงโครงสร้างการจัดเก็บข้อมูลในฐานข้อมูล โดยข้อมูลรายละเอียดยูสเคสต้องแยกออกและจัดเก็บตามโครงสร้างฐานข้อมูลมีรายละเอียดดังนี้

- พิจารณาทีละยูสเคสในแผนภาพยูสเคส โดยพิจารณาเฉพาะยูสเคสที่ไม่ได้เป็นยูสเคสต้นแบบสำหรับยูสเคสอื่น กล่าวคือพิจารณายูสเคสที่คุณสมบัติ Is abstract ไม่เท่ากับ 0
- แยกข้อมูลทั่วไปของยูสเคส และใส่ลงในตาราง uc_context โดยข้อมูลทั่วไปประกอบด้วย
  - usecase_id คือหมายเลขยูสเคส
  - usecase_name คือชื่อของยูสเคส
  - description คือข้อความอธิบายยูสเคส
  - precondition คือกิจกรรมที่ต้องปฏิบัติก่อนเริ่มทำงานยูสเคส

- แยกข้อมูลนำเข้าของยูสเคส แล้วใส่ลงตาราง required_item ซึ่งข้อมูลนำเข้าของยูสเคสต้องประกอบด้วย
  - item_id คือรหัสของข้อมูลนำเข้า ซึ่งต้องตัวเลขที่ไม่ซ้ำกันเพื่อใช้เป็นรหัสประจำข้อมูลนำเข้า
  - item_name คือชื่อของข้อมูลนำเข้า
  - item_type คือชนิดของข้อมูลนำเข้า
  - item_size คือความยาวของข้อมูลนำเข้าชนิดชุดอักขระ หรือจำนวนหลักหลังจุดทศนิยมของข้อมูลนำเข้าชนิดเลขจำนวนจริง หรือจำนวนหลักของข้อมูลนำเข้าชนิดเลขจำนวนเต็ม
  - item_fix คือคุณสมบัติระบุให้ข้อมูลทดสอบที่สร้างต้องมีขนาดเท่ากับขนาดที่ระบุใน item_size มิฉะนั้นข้อมูลทดสอบที่สร้างจะมีขนาดน้อยกว่าหรือเท่ากับขนาดที่ระบุใน item_size
  - item_maxval คือค่าสูงสุดที่เป็นไปได้ของข้อมูลนำเข้า
  - item_minval คือค่าต่ำสุดที่เป็นไปได้ของข้อมูลนำเข้า
- ในแต่ละยูสเคสอาจมีข้อมูลนำเข้ามากกว่า 1 ตัว และในทางกลับกันข้อมูลนำเข้าแต่ละตัวอาจเป็นข้อมูลนำเข้าหลายยูสเคส ซึ่งในตาราง uc_item เก็บความสัมพันธ์เหล่านี้ด้วยข้อมูลหมายเลขของยูสเคส และรหัสของข้อมูลนำเข้า
- แยกข้อมูลเกี่ยวกับประโยคเงื่อนไขต่างๆ ของลำดับเหตุการณ์ของยูสเคส และผลลัพธ์หลังจากการทำงานยูสเคสใส่ในตาราง condition ซึ่งประกอบด้วย
  - usecase_id คือหมายเลขของยูสเคส
  - condition_id คือหมายเลขของประโยคเงื่อนไขการเกิดลำดับเหตุการณ์ของยูสเคส
  - expression คือประโยคเงื่อนไขการเกิดลำดับเหตุการณ์ของยูสเคส
  - postcondition คือผลลัพธ์หลังจากการทำงานตามลำดับเหตุการณ์ของยูสเคส
- แยกข้อมูลลำดับเหตุการณ์สำเร็จของยูสเคสใส่ในตาราง success ซึ่งแต่ละยูสเคสมีลำดับเหตุการณ์สำเร็จเพียงหนึ่งลำดับเหตุการณ์เท่านั้น ข้อมูลลำดับเหตุการณ์สำเร็จให้แยกแต่ละขั้นตอนการทำงานของลำดับเหตุการณ์ซึ่งประกอบด้วย
  - usecase_id คือหมายเลขของยูสเคส
  - condition_id คือหมายเลขของประโยคเงื่อนไขการเกิดลำดับเหตุการณ์ของยูสเคส
  - step_id คือหมายเลขลำดับการทำงานของลำดับเหตุการณ์สำเร็จ
  - actions คือรายละเอียดการทำงานของลำดับเหตุการณ์สำเร็จ

- ในกรณีที่ยูสเคสมีลำดับเหตุการณ์ทางเลือกอื่น ให้แยกข้อมูลลำดับเหตุการณ์ทางเลือกอื่นของยูสเคสใส่ในตาราง alternative แต่ละยูสเคสอาจลำดับเหตุการณ์ทางเลือกได้มากกว่าหนึ่งลำดับเหตุการณ์ ข้อมูลลำดับเหตุการณ์ทางเลือกอื่นให้แยกแต่ละขั้นตอนการทำงานของลำดับเหตุการณ์เช่นเดียวกับการแยกข้อมูลลำดับเหตุการณ์สำเร็จ

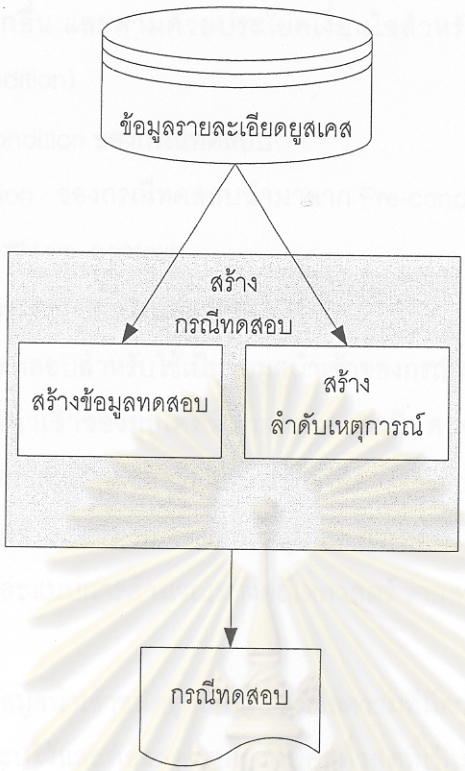
2) การสร้างกรณีทดสอบ

ขั้นตอนนี้เป็นขั้นตอนที่นำข้อมูลรายละเอียดยูสเคสที่แยกลงฐานข้อมูลมาสร้างกรณีทดสอบ ซึ่งการสร้างกรณีทดสอบนั้นพิจารณาจากประโยคเงื่อนไขของแต่ละยูสเคส ซึ่งประโยคเงื่อนไขหนึ่งประโยคทำให้เกิดลำดับเหตุการณ์ได้หนึ่งลำดับเหตุการณ์ และสร้างกรณีทดสอบได้หนึ่งกรณีทดสอบเพื่อใช้ทดสอบลำดับเหตุการณ์ โดยจำนวนกรณีทดสอบที่สร้างได้มีจำนวนเท่ากับผลรวมของประโยคเงื่อนไขของแต่ละยูสเคส

กรณีทดสอบในงานวิจัยนี้ ประกอบด้วย Test case id, Test case name, Description, Pre-condition, Input, Expected output และ Post-condition ส่วนประกอบของกรณีทดสอบมีรายละเอียดดังนี้

- Test case id เป็นหมายเลขของกรณีทดสอบ
- Test case name เป็นข้อความแสดงชื่อกรณีทดสอบ
- Description เป็นข้อความแสดงรายละเอียดของกรณีทดสอบ
- Pre-condition เป็นข้อความแสดงสิ่งที่ต้องปฏิบัติก่อนเริ่มต้นทดสอบ
- Input เป็นข้อมูลทดสอบซึ่งประกอบด้วย ชื่อของข้อมูลทดสอบ และค่าของข้อมูลทดสอบ
- Expected output เป็นผลลัพธ์คาดหวังของกรณีทดสอบ โดยระบุเป็นลำดับเหตุการณ์ที่คาดว่าจะเกิดขึ้นเมื่อทดสอบด้วยข้อมูลทดสอบ
- Post-condition เป็นข้อความแสดงผลลัพธ์หลังจากการทำงานตามเงื่อนไข

การสร้างกรณีทดสอบแบ่งออกเป็น 2 ส่วนหลักคือ การสุ่มสร้างข้อมูลทดสอบ และการสร้างลำดับเหตุการณ์ที่คาดว่าจะเกิดขึ้น ดังแสดงในรูปที่ 4.7



รูปที่ 4.7 ภาพรวมขั้นตอนการสร้างกรณีทดสอบ

ขั้นตอนการสร้างกรณีทดสอบมีขั้นตอนดังนี้

- พิจารณาสร้างกรณีทดสอบทีละประโยคเงื่อนไขของแต่ละยูสเคส

โดยเริ่มต้นพิจารณายูสเคสทีละยูสเคส และพิจารณาทีละประโยคเงื่อนไขของยูสเคส ซึ่งประโยคเงื่อนไขของยูสเคสหนึ่งประโยคสามารถสร้างกรณีทดสอบได้หนึ่งกรณีทดสอบ

- กำหนดหมายเลขของกรณีทดสอบ

หมายเลขของกรณีทดสอบนำมาจากหมายเลขของยูสเคสที่นำมาสร้างกรณีทดสอบ ตามด้วยเครื่องหมายจุลภาค และตัวเลขที่ไม่ซ้ำกันตั้งแต่ 1 เป็นต้นไป

- ระบุชื่อของกรณีทดสอบ

ชื่อของกรณีทดสอบนำมาจากชื่อของยูสเคสที่นำมาสร้างกรณีทดสอบ ซึ่งก็คือข้อมูล usecase_name ในตาราง uc_context

- ระบุข้อความอธิบายกรณีทดสอบ

ข้อความอธิบายกรณีทดสอบให้ระบุว่า Success Scenario สำหรับกรณีที่เป็นกรณีทดสอบลำดับเหตุการณ์สำเร็จ หรือระบุว่า Alternative Scenario สำหรับกรณีที่เป็นกรณีทดสอบ

ลำดับเหตุการณ์ทางเลือกอื่น และตามด้วยประโยคเงื่อนไขสำหรับลำดับเหตุการณ์ (ข้อมูล expression ในตาราง condition)

- ระบุ Pre-condition ของกรณีทดสอบ

Pre-condition ของกรณีทดสอบนำมาจาก Pre-condition ของยูสเคส ซึ่งก็คือข้อมูล precondition ในตาราง uc_context

- สร้างข้อมูลทดสอบของกรณีทดสอบ

สร้างข้อมูลทดสอบสำหรับใช้เป็นข้อมูลนำเข้าของกรณีทดสอบ โดยต้องสร้างข้อมูลทดสอบตามจำนวนข้อมูลนำเข้าของยูสเคสที่นำมาสร้างกรณีทดสอบ การสร้างค่าของข้อมูลทดสอบต้องพิจารณาจากรายละเอียดของข้อมูลนำเข้า (ในตาราง required_item) และประโยคเงื่อนไข (ข้อมูล expression ในตาราง condition) ซึ่งรูปแบบของพจน์ในประโยคเงื่อนไขมี 2 แบบ คือแบบการเปรียบเทียบ และแบบการคำนวณทางคณิตศาสตร์ รายละเอียดของการสร้างค่าของข้อมูลทดสอบมีดังนี้

- สำหรับข้อมูลนำเข้าชนิดชุดอักขระ สุ่มข้อความที่มีความยาวของตัวอักษรไม่เกินความยาวของตัวอักษรที่ระบุไว้และต้องสอดคล้องกับค่าต่ำสุดที่เป็นไปได้ ค่าสูงสุดที่เป็นไปได้ รวมทั้งประโยคเงื่อนไข
- สำหรับข้อมูลนำเข้าชนิดตรรกะ สุ่มค่าจริงหรือเท็จซึ่งต้องสอดคล้องกับประโยคเงื่อนไข
- สำหรับข้อมูลนำเข้าชนิดตัวเลขจำนวนเต็มและกำหนดจำนวนหลักของข้อมูลนำเข้า สุ่มเลขที่มีจำนวนหลักไม่เกินจำนวนหลักที่กำหนดไว้
- สำหรับข้อมูลนำเข้าชนิดตัวเลขจำนวนเต็มหรือจำนวนจริงและรูปแบบของพจน์ในประโยคเงื่อนไขเป็นแบบการเปรียบเทียบ สุ่มเลขที่มีค่าสอดคล้องกับการเปรียบเทียบนั้น
- สำหรับข้อมูลนำเข้าชนิดตัวเลขจำนวนเต็มหรือจำนวนจริงและรูปแบบของพจน์ในประโยคเงื่อนไขเป็นการคำนวณทางคณิตศาสตร์ สุ่มเลขที่มีค่าสอดคล้องกับการคำนวณทางคณิตศาสตร์
- สำหรับข้อมูลนำเข้าอื่นที่เกี่ยวข้องกับยูสเคส แต่ไม่ได้เป็นส่วนหนึ่งของประโยคเงื่อนไข สุ่มค่าของข้อมูลทดสอบตามชนิดของข้อมูลนำเข้า ขนาดของข้อมูลนำเข้า และมีค่าสอดคล้องกับค่าต่ำสุดที่เป็นไปได้และค่าสูงสุดที่เป็นไปได้ของข้อมูลนำเข้า

- สร้างลำดับเหตุการณ์ที่คาดว่าจะเกิดขึ้นเพื่อใช้เป็นผลลัพธ์คาดหวังของกรณีทดสอบ

ผลลัพธ์คาดหวังของกรณีทดสอบนำมาจากลำดับเหตุการณ์ของยูสเคสเมื่อประโยคเงื่อนไขเป็นจริง โดยระบุลำดับเหตุการณ์ที่คาดว่าจะเกิดขึ้นเป็นข้อเริ่มต้นตั้งแต่ข้อ 1 เป็นต้นไป ผลลัพธ์คาดหวังของกรณีทดสอบแบ่งได้เป็น 2 แบบคือ

- ผลลัพธ์คาดหวังของกรณีทดสอบสำหรับทดสอบลำดับเหตุการณ์สำเร็จนำลำดับการทำงานมาจากข้อมูล actions ในตาราง success
- ผลลัพธ์คาดหวังของกรณีทดสอบสำหรับทดสอบลำดับเหตุการณ์ทางเลือกอื่น ให้พิจารณาจากหมายเลขของประโยคเงื่อนไขของลำดับเหตุการณ์ทางเลือกอื่นที่เป็นจริงเมื่อทดสอบด้วยข้อมูลทดสอบที่สร้างขึ้น โดยหมายเลขหน้าเครื่องหมายมหัพภาคจะแสดงหมายเลขลำดับการทำงานในลำดับเหตุการณ์สำเร็จ ให้นำการทำงานของลำดับเหตุการณ์สำเร็จ (ข้อมูล actions ในตาราง success) ตั้งแต่การทำงานลำดับที่ 1 จนถึงการทำงานลำดับหมายเลขหน้าเครื่องหมายมหัพภาค และตามด้วยการทำงานในลำดับเหตุการณ์ทางเลือกอื่น (ข้อมูล actions ในตาราง alternative) สำหรับกรณีที่การทำงานในลำดับเหตุการณ์ทางเลือกอื่นมีจุดเชื่อมโยงกลับไปสู่ลำดับเหตุการณ์สำเร็จ ต้องตามด้วยลำดับการทำงานที่เกิดขึ้น ณ จุดดังกล่าวจนถึงลำดับการทำงานสุดท้ายของลำดับเหตุการณ์สำเร็จ

- ระบุ Post-condition

Post-codition ของกรณีทดสอบนำมาจาก Post-condition ของยูสเคสตามลำดับเหตุการณ์ที่เกิดขึ้น ซึ่งก็คือข้อมูล postcondition ในตาราง condition

4.3 มาตรวัดสำหรับการทดสอบโดยอ้างอิงแผนภาพและเอกสารยูสเคส

จากกรณีทดสอบที่ได้จากวิธีการในหัวข้อที่ผ่านมา ความก้าวหน้าของการทดสอบสามารถถูกวัดได้จากอัตราส่วนระหว่างจำนวนกรณีทดสอบที่ได้ทดสอบที่ได้ถูกทดสอบไปแล้วกับจำนวนกรณีทดสอบทั้งหมดที่ได้สร้างขึ้น เนื่องจากกรณีทดสอบได้ถูกสร้างขึ้นจากวิธีการที่เป็นระบบ จึงสามารถทราบถึงจำนวนกรณีทดสอบทั้งหมดที่สร้างขึ้นได้อย่างแน่นอน การวัดความก้าวหน้าของการทดสอบจึงขึ้นอยู่กับการบันทึกผลของการนำกรณีทดสอบที่สร้างขึ้นไปใช้ในการทดสอบ

ประสิทธิภาพหรือความครอบคลุมของการทดสอบสามารถวัดได้ในลักษณะเดียวกับความก้าวหน้าของการทดสอบ เพราะกรณีทดสอบได้ถูกสร้างมาอย่างมีระบบให้ครอบคลุมทุกเงื่อนไขในแต่ละยูสเคส ดังนั้นเมื่อทดสอบครบตามกรณีทดสอบทั้งหมดที่ได้ถูกสร้างขึ้น ก็จะถือว่าครอบคลุมทุกๆ เงื่อนไขการทำงานของยูสเคส

มาตรวัดของประสิทธิผลของการทดสอบสามารถวัดได้จากอัตราส่วนของข้อผิดพลาดที่พบระหว่างการทดสอบกับหน่วยของการปฏิบัติระหว่างการทดสอบ เช่น จำนวนกรณีทดสอบ

ระยะเวลาที่ใช้ในการทดสอบ ฯลฯ ซึ่งข้อมูลประเภทนี้สามารถเก็บรวบรวมได้ระหว่างการทดสอบ เช่นเดียวกับข้อมูลที่ใช้วัดความก้าวหน้าของการทดสอบ

กรณีทดสอบที่สร้างได้จากวิธีการสร้างกรณีทดสอบโดยอ้างอิงแผนภาพ และเอกสารยูสเคส กรณีทดสอบที่สร้างได้จะมีจำนวนดังนี้

จำนวนกรณีทดสอบ = (1 + จำนวน Alternative Scenario ของยาเคสนั้น) + จำนวน Alternative Scenario ของยูสเคสที่อินคลูด + จำนวน Alternative Scenario ที่เอ็กซ์เทน

4.4 เครื่องมือสำหรับการสร้างกรณีทดสอบโดยอ้างอิงแผนภาพ และเอกสารยูสเคส

จากวิธีการทดสอบในหัวข้อ 4.2 ผู้วิจัยได้สร้างซอฟต์แวร์เพื่อเป็นเครื่องมือในการสร้างกรณีทดสอบจากเอกสารยูสเคส โดยในเอกสารยูสเคสจะต้องระบุรายการข้อมูลนำเข้าทั้งหมด พร้อมทั้งชนิดของข้อมูล เครื่องมือจะวิเคราะห์เงื่อนไขในการทำงานของยูสเคสประกอบกับชนิดของข้อมูลนำเข้า แล้วสร้างกรณีทดสอบพร้อมด้วยตัวอย่างข้อมูลนำเข้าสำหรับกรณีทดสอบ โดยตัวอย่างข้อมูลนำเข้าได้มาจากการสุ่มค่าตามเงื่อนไขของข้อมูลนำเข้าสำหรับกรณีทดสอบนั้นๆ

ต่อไปจะกล่าวถึงการพัฒนาเครื่องมือสร้างกรณีทดสอบโดยอัตโนมัติจากยูสเคส โดยกล่าวถึงสภาพแวดล้อมที่ใช้ในการพัฒนาเครื่องมือ ฐานข้อมูลของเครื่องมือ โครงสร้างของเครื่องมือ และตัวอย่างการสร้างกรณีทดสอบจากเอกสารยูสเคส ซึ่งมีรายละเอียดดังนี้

4.4.1 สภาพแวดล้อมที่ใช้ในการพัฒนาเครื่องมือ

1) ฮาร์ดแวร์ (Hardware)

ฮาร์ดแวร์ที่ใช้ในการพัฒนาเครื่องมือประกอบด้วย

- เครื่องคอมพิวเตอร์ส่วนบุคคล หน่วยประมวลผลเพนเทียมโฟร์ 1.6 กิกะเฮิรตซ์ (Pentium 4 1.6 GHz.)
- หน่วยความจำหลัก (RAM) 256 เมกะไบต์ (256 MB)
- ฮาร์ดดิสก์ (Hard disk) 40 กิกะไบต์ (40 GB)

2) ซอฟต์แวร์ (Software)

ซอฟต์แวร์ที่ใช้ในการพัฒนาเครื่องมือประกอบด้วย

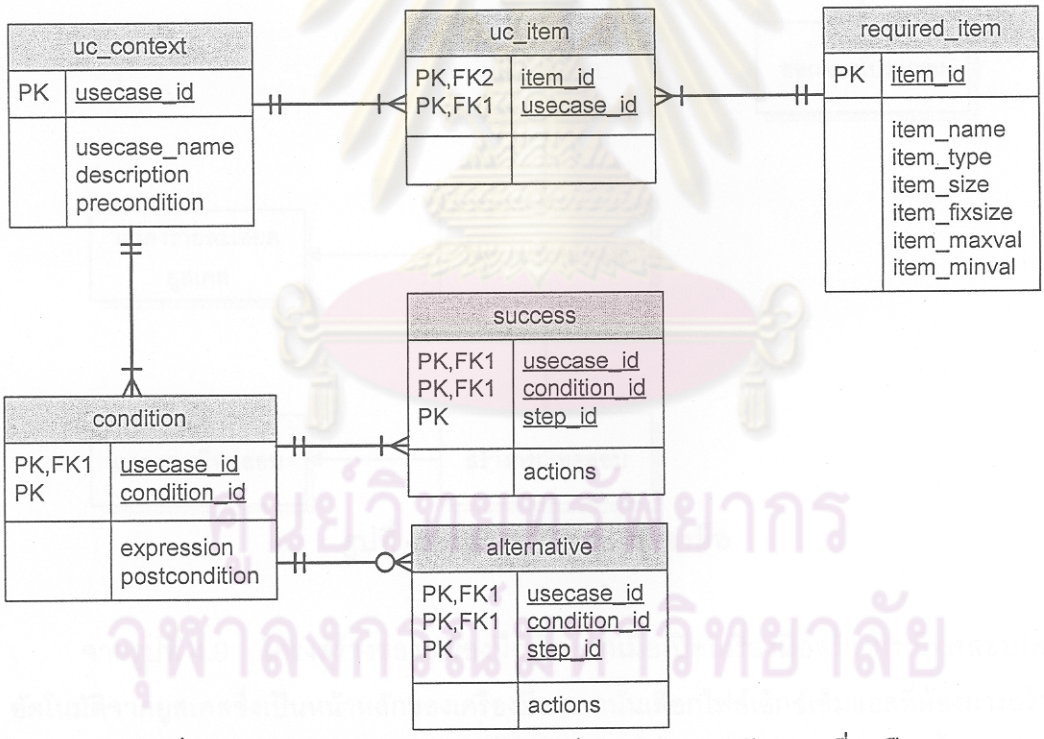
- ระบบปฏิบัติการ (Operating system) ไมโครซอฟท์วินโดวส์เอ็กซ์พี โพรเฟชชันแนล (Microsoft Windows XP Professional)
- ระบบจัดการฐานข้อมูล (Database management system) ไมโครซอฟท์ออฟฟิซแอ็กเซส 2003 (Microsoft Office Access 2003)

- พัฒนาเครื่องมือด้วยภาษาวิชัวล์เบสิกดอทเน็ตโดยใช้โปรแกรมไมโครซอฟท์วิชัวล์สตูดิโอดอทเน็ต 2003 (Microsoft Visual Studio .NET 2003)
- เครื่องมือช่วยสร้างเอกสารแผนภาพยูเอ็มแอล เรชันนอลโรสเอ็นเตอร์ไพรส์ (Rational Rose Enterprise Edition) เวอร์ชัน 2002
- เครื่องมือแปลงเอกสารแผนภาพยูเอ็มแอลเป็นไฟล์เอ็กซ์เอ็มแอล ยูนิซีสเอ็กซ์เอ็มไอ 1.3 แอดอิน (Unisys XMI 1.3 add-ins)

4.4.2 ฐานข้อมูลของเครื่องมือ

ฐานข้อมูลของเครื่องมือสร้างกรณีทดสอบโดยอัตโนมัติจากยูสเคสสามารถแสดงได้โดยใช้แผนภาพความสัมพันธ์ระหว่างเอนทิตี ซึ่งเป็นแผนภาพที่ใช้แสดงข้อมูลที่เก็บในฐานข้อมูล รวมทั้งแสดงความสัมพันธ์ระหว่างตารางต่างๆ ในฐานข้อมูล

แผนภาพความสัมพันธ์ระหว่างเอนทิตีของเครื่องมือนี้แสดงดังรูปที่ 4.8



รูปที่ 4.8 แผนภาพแสดงความสัมพันธ์ระหว่างเอนทิตีของเครื่องมือ

จากรูปที่ 4.8 ฐานข้อมูลของเครื่องมือประกอบด้วย 6 ตาราง ซึ่งมีรายละเอียดดังนี้

1) ตาราง uc_context เป็นตารางที่เก็บข้อมูลทั่วไปของยูสเคส
2) ตาราง required_item เป็นตารางที่เก็บข้อมูลทั่วไปของข้อมูลนำเข้าของยูสเคส

3) ตาราง uc_item เป็นตารางที่เก็บความสัมพันธ์ระหว่างข้อมูลนำเข้ากับยูสเคส เพื่อระบุว่าเป็นข้อมูลนำเข้าของยูสเคสใดบ้าง

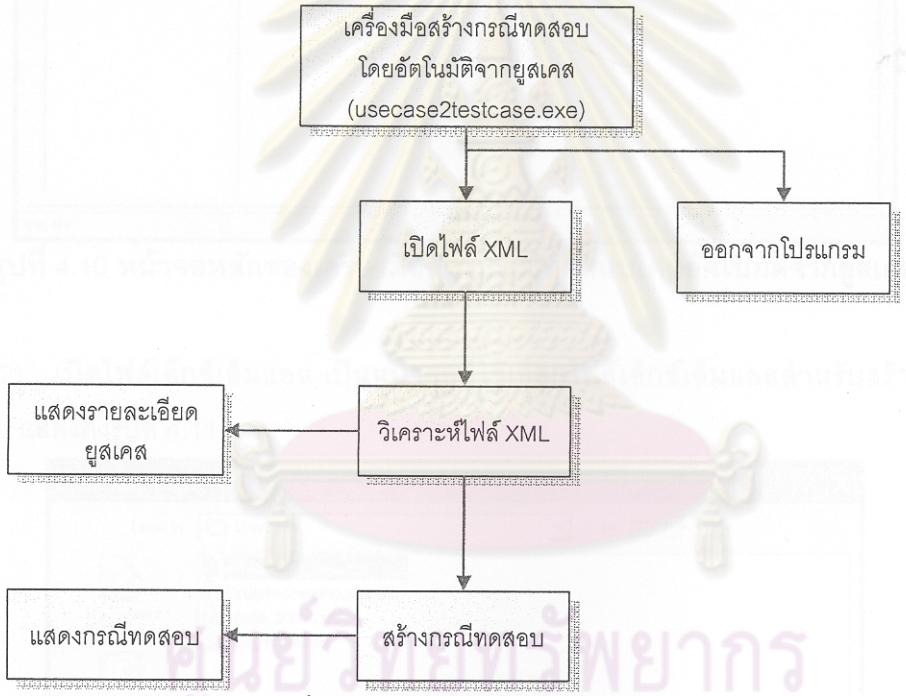
4) ตาราง condition เป็นตารางที่เก็บประโยคเงื่อนไขของลำดับเหตุการณ์ที่เกิดขึ้นในยูสเคส

5) ตาราง success เป็นตารางที่เก็บลำดับการทำงานของลำดับเหตุการณ์สำเร็จ

6) ตาราง alternative เป็นตารางที่เก็บลำดับการทำงานของลำดับเหตุการณ์ทางเลือกอื่น

4.4.3 โครงสร้างของเครื่องมือ

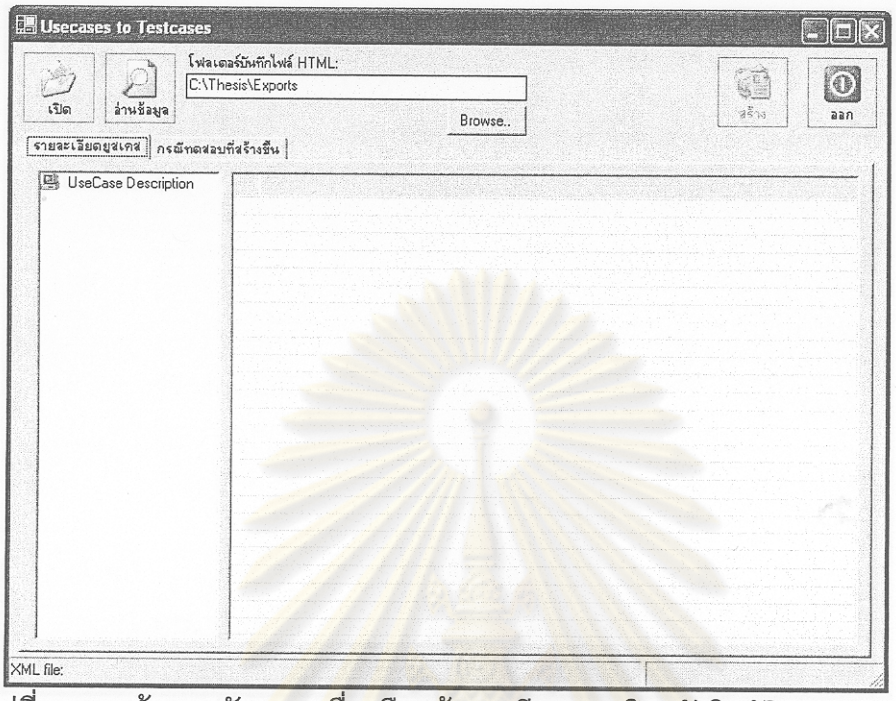
โครงสร้างของเครื่องมือสร้างกรณีทดสอบโดยอัตโนมัติจากยูสเคสแสดงได้ดังรูปที่ 4.9



รูปที่ 4.9 โครงสร้างของเครื่องมือ

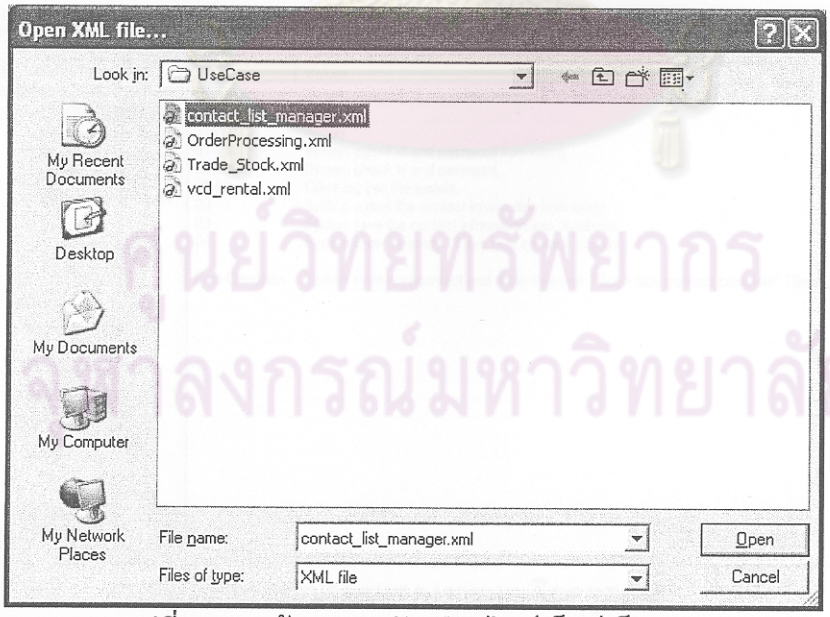
จากรูปที่ 4.9 โครงสร้างของเครื่องมือเริ่มจากเมื่อเปิดเครื่องมือสร้างกรณีทดสอบโดยอัตโนมัติจากยูสเคสซึ่งเป็นหน้าหลักของเครื่องมือ จากนั้นเลือกไฟล์เอ็กซ์เอ็มแอลที่ต้องการสร้างกรณีทดสอบ ต่อมาเลือกการอ่านข้อมูลเพื่อวิเคราะห์ไฟล์เอ็กซ์เอ็มแอล จากนั้นจะแสดงรายละเอียดยูสเคส หลังจากนั้นกำหนดที่เก็บกรณีทดสอบแล้วเลือกสร้างกรณีทดสอบ และขั้นตอนสุดท้ายจะได้กรณีทดสอบที่สร้างขึ้นพร้อมทั้งแสดงรายละเอียดกรณีทดสอบทั้งหมด ซึ่งในแต่ละส่วนมีหน้าจอดังนี้

1) เครื่องเครื่องมือสร้างกรณีทดสอบโดยอัตโนมัติจากยูสเคส หน้าจอหลักแสดงดังรูปที่ 4.10



รูปที่ 4.10 หน้าจอหลักของเครื่องมือสร้างกรณีทดสอบโดยอัตโนมัติจากยูสเคส

2) เปิดไฟล์เอ็กซ์เอ็มแอล เป็นหน้าจอที่ใช้เลือกไฟล์เอ็กซ์เอ็มแอลสำหรับสร้างกรณีทดสอบ แสดงดังรูปที่ 4.11

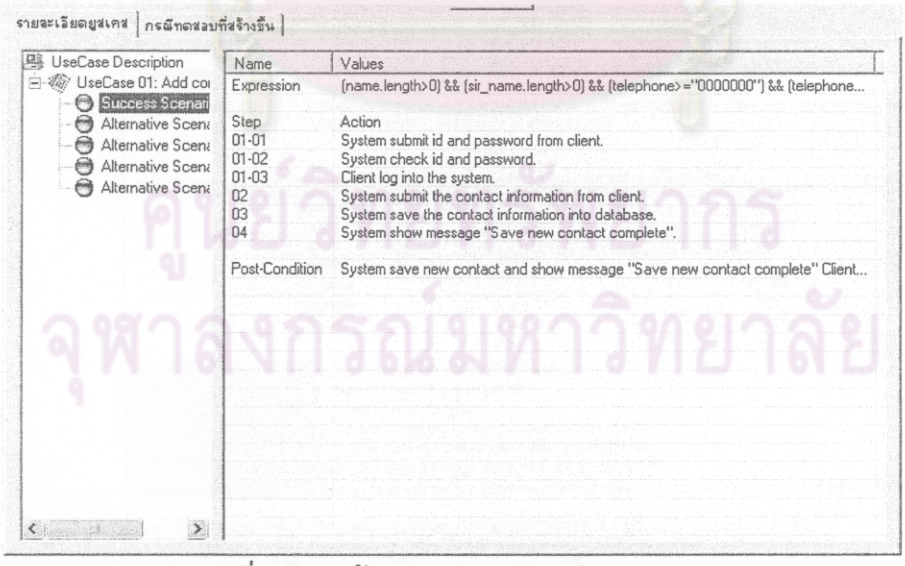


รูปที่ 4.11 หน้าจอสำหรับเลือกไฟล์เอ็กซ์เอ็มแอล

3) วิเคราะห์ไฟล์เอ็กซ์เอ็มแอล เป็นส่วนที่ใช้สำหรับวิเคราะห์ไฟล์เอ็กซ์เอ็มแอล แสดงได้ดังรูปที่ 4.12

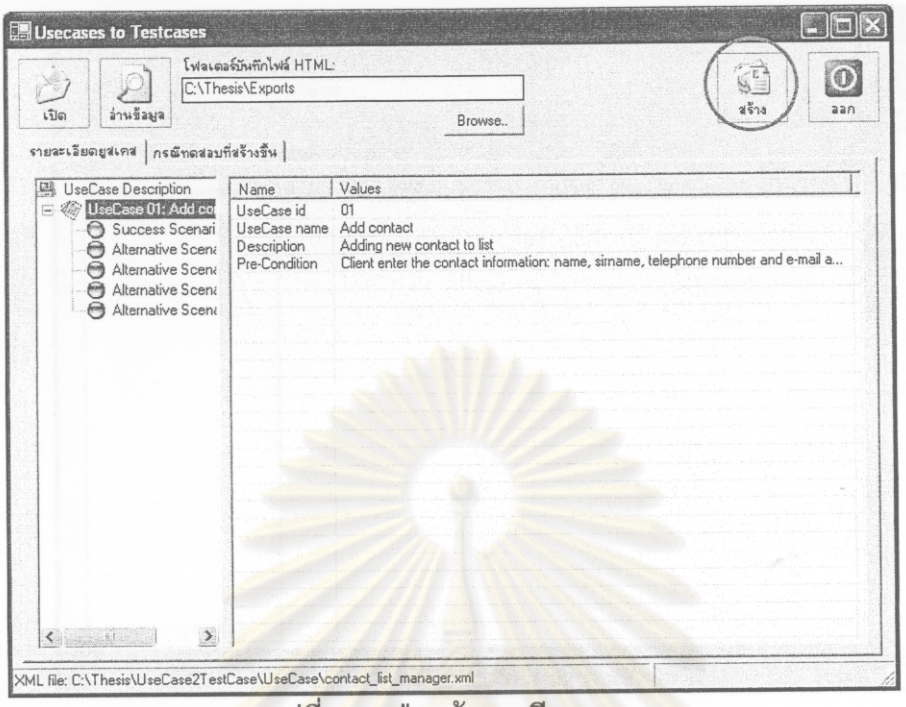
รูปที่ 4.12 ปุ่มอ่านข้อมูล (สำหรับวิเคราะห์ไฟล์)

4) แสดงรายละเอียดยูสเคส เป็นส่วนที่แสดงรายละเอียดของยูสเคสหลังจากที่เครื่องมือวิเคราะห์ไฟล์เอ็กซ์เอ็มแอล แสดงได้ดังรูปที่ 4.13



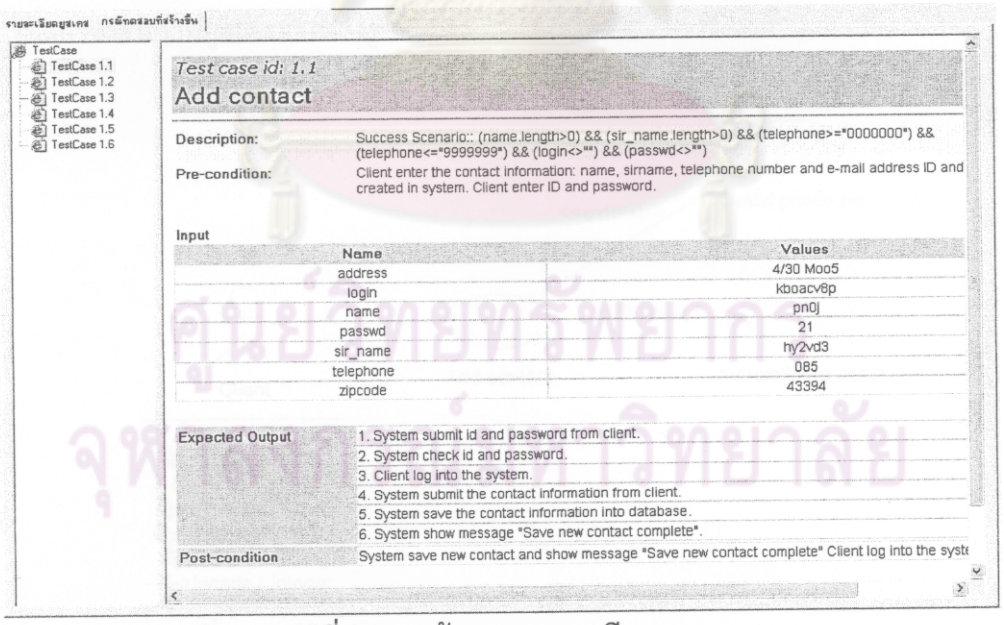
รูปที่ 4.13 หน้าจอแสดงรายละเอียดยูสเคส

5) สร้างกรณีทดสอบ เป็นส่วนที่ใช้สำหรับสร้างกรณีทดสอบ แสดงได้ดังรูปที่ 4.14



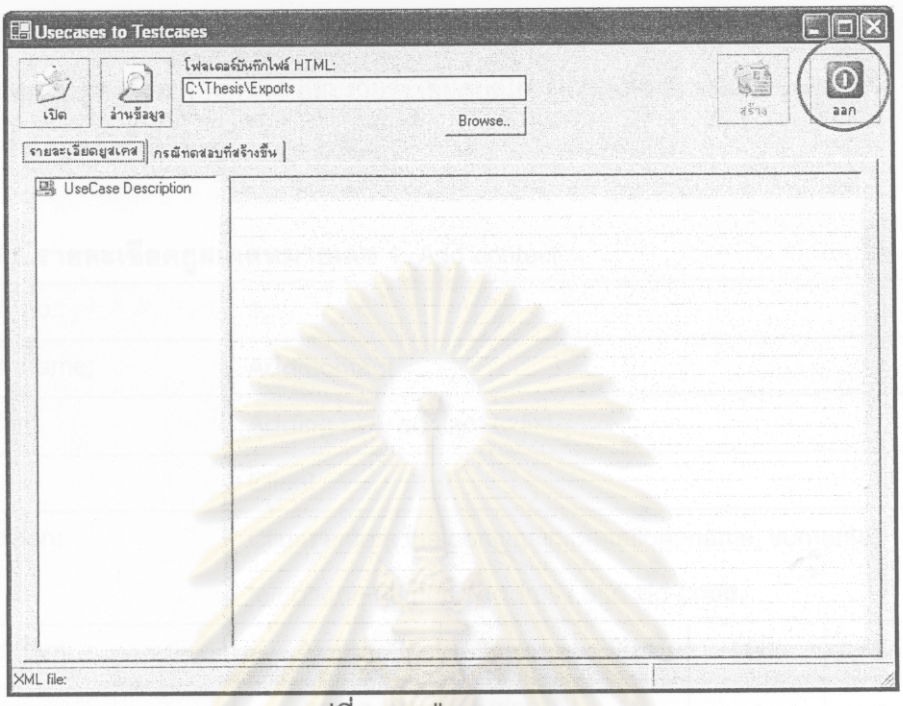
รูปที่ 4.14 ปุ่มสร้างกรณีทดสอบ

6) แสดงกรณีทดสอบ เป็นส่วนที่ใช้แสดงกรณีทดสอบที่สร้างขึ้นโดยเครื่องมือ แสดงได้ดังรูปที่4.15



รูปที่ 4.15 หน้าจอแสดงกรณีทดสอบ

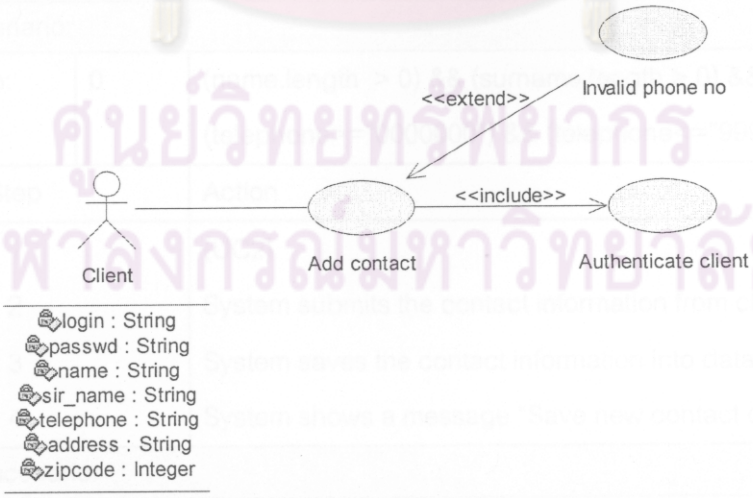
7) ออกจากโปรแกรม เป็นส่วนที่ใช้ปิดการใช้งานเครื่องมือ แสดงได้ดังรูปที่ 4.16



รูปที่ 4.16 ปุ่มออก

4.4.4 ตัวอย่างการสร้างกรณีทดสอบจากเอกสารยูสเคส

แผนภาพยูสเคสที่นำมาเป็นตัวอย่างคือตัวอย่างแผนภาพยูสเคส Contact list manager โดยตัวอย่างแผนภาพยูสเคสแสดงดังรูปที่ 4.17



รูปที่ 4.17 ตัวอย่างแผนภาพยูสเคส: Contact list manager

จากรูปที่ 4.17 รายละเอียดยูสเคสหมายเลข 1: Add contact ของยูสเคส Add contact ในแผนภาพยูสเคสแสดงดังตารางที่ 4.3 รายละเอียดยูสเคสหมายเลข 2: Authenticate client ในแผนภาพยูสเคสแสดงดังตารางที่ 4.4 และรายละเอียดยูสเคสหมายเลข 3: Invalid phone no ในแผนภาพยูสเคสแสดงดังตารางที่ 4.5

ตารางที่ 4.3 รายละเอียดยูสเคสหมายเลข 1: Add contact

<table>
<tr><td colspan="2">Use case no.:</td><td colspan="5">1</td></tr>
<tr><td colspan="2">Use case name:</td><td colspan="5">Add contact</td></tr>
<tr><td colspan="2">Description:</td><td colspan="5">Adding new contact to list</td></tr>
<tr><td colspan="2">Actor:</td><td colspan="5">Client</td></tr>
<tr><td colspan="2">Pre-condition:</td><td colspan="5">Client enters the contact information: name, surname, telephone number, address and zip code.</td></tr>
<tr><td colspan="2" rowspan="6">Required-item:</td><td>Name</td><td>Type</td><td>Size</td><td>Max</td><td>Min</td></tr>
<tr><td>name</td><td>String</td><td>15</td><td>-</td><td>-</td></tr>
<tr><td>surname</td><td>String</td><td>20</td><td>-</td><td>-</td></tr>
<tr><td>telephone</td><td>String</td><td>7</td><td>-</td><td>-</td></tr>
<tr><td>address</td><td>String</td><td>20</td><td>-</td><td>-</td></tr>
<tr><td>zipcode</td><td>Integer</td><td>5</td><td>10000</td><td>99999</td></tr>
<tr><td colspan="2">Is abstract:</td><td colspan="5">0</td></tr>
<tr><td colspan="7">Success scenario:</td></tr>
<tr><td>Condition no:</td><td>0</td><td colspan="5">(name.length > 0) && (surname.length > 0) && (telephone>="0000000") && (telephone<="9999999")</td></tr>
<tr><td colspan="2">Step</td><td colspan="5">Action</td></tr>
<tr><td colspan="2">1
2
3
4</td><td colspan="5">{UC2}
System submits the contact information from client.
System saves the contact information into database.
System shows a message “Save new contact complete”.</td></tr>
<tr><td colspan="7">Alternative scenario:</td></tr>
<tr><td>Condition no:</td><td>2.1</td><td colspan="5">(name.length <= 0)</td></tr>
<tr><td colspan="2">Step</td><td colspan="5">Action</td></tr>
<tr><td colspan="2">2.1.1</td><td colspan="5">System shows an error message “Please enter name”.</td></tr>
</table>

ตารางที่ 4.3 รายละเอียดยูสเคสหมายเลข 1: Add contact (ต่อ)

Condition no:	2.2	(surname.length <= 0)
Step		Action
2.2.1		System shows an error message “Please enter surname”
Condition no:	2.3	n/a
Step		Action
2.3		{UC3}
Post-condition	0	System saves new contact and shows a message “Save new contact complete”.
	2.1	System shows an error message “Please enter name”
	2.2	System shows an error message “Please enter surname”

ตารางที่ 4.4 รายละเอียดยูสเคสหมายเลข 2: Authenticate client

Use case no.:	2				
Use case name:	Authenticate client				
Description:	Authenticate client before using the system.				
Actor:	Client				
Pre-condition:	Login name and password are created in system. Client enters login name and password.				
Required-item:	Name	Type	Size	Max	Min
	login	String	8	-	-
	passwd	String	5	-	-
Is abstract	0				
Success scenario:					
Condition no: 0	(login <> “”) && (passwd <> “”)				
Order	Action				
1	System submits login name and password form client.				
2	System checks login name and password.				
3	Client logs in to the system.				

ตารางที่ 4.4 ตัวอย่างรายละเอียดยูสเคสหมายเลข 2: Authenticate client (ต่อ)

Alternative scenario:		
Condition no:	2.1	(login == "")
Step		Action
2.1.1		System shows an error message "Please enter login".
Condition no:	2.2	(passwd == "")
Step		Action
2.2.1		System shows an error message "Please enter password".
Post-condition:	0	Client logs into the system.
	2.1	System shows an error message "Please enter login".
	2.2	System shows an error message "Please enter password".

ตารางที่ 4.5 ตัวอย่างรายละเอียดยูสเคสหมายเลข 3: Invalid phone no

Use case no.:		3				
Use case name:		Invalid phone no				
Description:		Telephone number is invalid				
Actor:		Client				
Pre-condition:		Client enters telephone number.				
Required-item:		Name	Type	Size	Max	Min
		telephone	String	7	-	-
Is abstract		0				
Success scenario:						
Condition no:	0	(telephone < "0000000") || (telephone > "9999999")				
Step		Action				
1		System shows error message "telephone number length must be 7"				
Post-condition:	0	System shows error message "telephone number length must be 7"				

เนื่องจากยูสเคส Authenticate client และยูสเคส Invalid phone no มีความสัมพันธ์กับยูสเคส Add contact แบบอินครูด และเอ็กซ์เทนตามลำดับจึงต้องรวมยูสเคสทั้งสองเข้ากับยูสเคส Add contact ก่อนสร้างกรณีทดสอบซึ่งยูสเคสที่รวมแล้ว มีรายละเอียดยูสเคส Add contact ใหม่ได้ดังตารางที่ 4.6

ตารางที่ 4.6 รายละเอียดยูสเคสที่รวมความสัมพันธ์ยูสเคสหมายเลข 1: Add contact

Use case no.:	1				
Use case name:	Add contact				
Description:	Adding new contact to list				
Actor:	Client				
Pre-condition:	Client enters the contact information: name, surname, telephone number, address and zip code. Login name and password are created in system. Client enters login name and password.				
Required-item:	Name	Type	Size	Min	Max
	login	String	8	-	-
	passwd	String	5	-	-
	name	String	15	-	-
	surname	String	20	-	-
	telephone	String	7	-	-
	address	String	20	-	-
	zipcode	Integer	5	10000	99999
Is abstract	0				

ตารางที่ 4.6 รายละเอียดยูสเคสที่รวมความสัมพันธ์ยูสเคสหมายเลข 1: Add contact (ต่อ)

Success scenario:		
Condition no:	0	(name.length>0) && (surname.length>0) && (telephone>="0000000") && (telephone<="9999999") && (login <> "") && (passwd <> "")
Step		Action
1-1		System submits login name and password form client.
1-2		System checks login name and password.
1-3		Client logs in to the system.
2		System submits the contact information form client.
3		System saves the contact information into database.
4		System shows a message "Save new contact complete".
Alternative scenario:		
Condition no:	1-2.1	(login == "")
Step		Action
1-2.1.1		System shows an error message "Please enter login".
Condition no:	1-2.2	(passwd == "")
Step		Action
1-2.2.1		System shows an error message "Please enter password".
Condition no:	2.1	(name.length<=0)
Step		Action
2.1.1		System shows an error message "Please enter name".
Condition no:	2.2	(surname.length<=0)
Step		Action
2.2.1		System shows an error message "Please enter surname".
Condition no:	2.3	(telephone < "0000000") || (telephone > "9999999")
Step		Action
2.3.1		System shows an error message "telephone number length must be 7".

ตารางที่ 4.6 รายละเอียดยูสเคสที่รวมความสัมพันธ์ยูสเคสหมายเลข 1: Add contact (ต่อ)

Post-condition:	0	Client logs into the system. System saves new contact and shows a message “Save new contact complete”.
	1-2.1	System shows an error message “Please enter login”.
	1-2.2	System shows an error message “Please enter password”.
	2.1	System shows an error message “Please enter name”.
	2.2	System shows an error message “Please enter sir name”.
	2.3	System shows an error message “telephone number length must be 7”.

จากยูสเคส Add contact สามารถสร้างกรณีทดสอบได้ 6 กรณีทดสอบดังนี้

ตารางที่ 4.7 กรณีทดสอบหมายเลข 1.1 ของยูสเคส Add Contract

Test case id	1.1	
Test case name	Add contact	
Description	Success Scenario: (len(name)>0) AND (len(sir_name)>0) AND (telephone>=”0000000”) AND (telephone<=”9999999”) AND (login <> “”) AND (passwd <> “”)	
Pre-condition	Client enter the contact information: name, sir name, telephone number and e-mail address. ID and password are created in system. Client enter ID and password.	
Input	Name	Value
	id	setapong
	passwd	note
	name	andryi
	sir_name	shevchenko
	telephone	1234567
	address	Italy

ตารางที่ 4.7 กรณีทดสอบหมายเลข 1.1 ของยูสเคส Add Contract (ต่อ)

Expected Output	System submit id and password form client System check id and password Client log into the system System submit the contact information form client System save the contact information into database System show message "Save new contact complete"
Post-condition	Client log into the system. System save new contact and show message "Save new contact complete"

ตารางที่ 4.8 กรณีทดสอบหมายเลข 1.2 ของยูสเคส Add Contract

Test case id	1.2	
Test case name	Add contact	
Description	Alternative Scenario: (login = "")	
Pre-condition	Client enter the contact information: name, sir name, telephone number and e-mail address. ID and password are created in system. Client enter ID and password.	
Input	Name	Value
	id	<blank>
	passwd	note
	name	andryi
	sir_name	shevchenko
	telephone	1234567
	address	Italy
Expected Output	System submit id and password form client System check id and password System show an error message "Please enter ID".	
Post-condition	System show an error message "Please enter ID".	

ตารางที่ 4.9 กรณีทดสอบหมายเลข 1.3 ของยูสเคส Add Contract

Test case id	1.3	
Test case name	Add contact	
Description	Alternative Scenario: (passwd = "")	
Pre-condition	Client enter the contact information: name, sir name, telephone number and e-mail address. ID and password are created in system. Client enter ID and password.	
Input	Name	Value
	id	setapong
	passwd	<blank>
	name	andryi
	sir_name	shevchenko
	telephone	1234567
	address	Italy
Expected Output	System submit id and password form client System check id and password System show an error message "Please enter Password".	
Post-condition	System show an error message "Please enter Password".	

ตารางที่ 4.10 กรณีทดสอบหมายเลข 1.4 ของยูสเคส Add Contract

Test case id	1.4
Test case name	Add contact
Description	Alternative Scenario: (len(name)<=0)
Pre-condition	Client enter the contact information: name, sir name, telephone number and e-mail address. ID and password are created in system. Client enter ID and password.

ตารางที่ 4.10 กรณีทดสอบหมายเลข 1.4 ของยูสเคส Add Contract (ต่อ)

Input	Name	Value
	id	setapong
	passwd	note
	name	<blank>
	sir_name	shevchenko
	telephone	1234567
	address	Italy
Expected Output	System submit id and password form client System check id and password Client log into the system System submit the contact information form client System show an error message “Please enter Name”.	
Post-condition	System show an error message “Please enter Name”.	

ตารางที่ 4.11 กรณีทดสอบหมายเลข 1.5 ของยูสเคส Add Contract

Test case id	1.5	
Test case name	Add contact	
Description	Alternative Scenario: (len(sir_name)<=0)	
Pre-condition	Client enter the contact information: name, sir name, telephone number and e-mail address. ID and password are created in system. Client enter ID and password.	
Input	Name	Value
	id	setapong
	passwd	note
	name	andryi
	sir_name	<blank>
	telephone	1234567
	address	Italy

ตารางที่ 4.11 กรณีทดสอบหมายเลข 1.5 ของยูสเคส Add Contract (ต่อ)

Expected Output	System submit id and password form client System check id and password Client log into the system System submit the contact information form client System show an error message "Please enter Sir name".
Post-condition	System show an error message "Please enter Sir name".

ตารางที่ 4.12 กรณีทดสอบหมายเลข 1.6 ของยูสเคส Add Contract

Test case id	1.6	
Test case name	Add contact	
Description	Alternative Scenario: (telephone < "0000000") OR (telephone > "9999999")	
Pre-condition	Client enter the contact information: name, sir name, telephone number and e-mail address. ID and password are created in system. Client enter ID and password.	
Input	Name	Value
	id	setapong
	passwd	note
	name	andryi
	sir_name	shevchenko
	telephone	98765
	address	Italy
Expected Output	System submit id and password form client System check id and password Client log into the system System submit the contact information form client System show error message "telephone number length must be 7"	
Post-condition	System show error message "telephone number length must be 7"	

บทที่ 5
สรุปผลการวิจัย

งานวิจัยชิ้นนี้ได้นำเสนอแนวคิดในการสร้างมาตรวัดสำหรับการทดสอบซอฟต์แวร์เชิงวัตถุ ซึ่งสามารถนำมาใช้ในการวัดความก้าวหน้าของการทดสอบ และประเมินประสิทธิภาพของการทดสอบได้ พร้อมกันนี้ งานวิจัยนี้ได้นำเสนอวิธีการทดสอบ และการสร้างกรณีทดสอบที่เหมาะสมสำหรับลักษณะเฉพาะของซอฟต์แวร์เชิงวัตถุด้วย

ในงานวิจัยชิ้นนี้ ได้พิจารณาการทดสอบซอฟต์แวร์ใน 2 ระดับ คือ การทดสอบในระดับระบบ และการทดสอบในระดับบูรณาการ ในการทดสอบระดับระบบ ในงานวิจัยนี้ ได้เสนอวิธีการทดสอบซอฟต์แวร์โดยการสร้างกรณีทดสอบโดยอ้างอิงจากแผนภาพและเอกสารยูสเคส ซึ่งกรณีทดสอบนี้ จะบอกถึงขั้นตอนที่ต้องปฏิบัติเพื่อทดสอบ และ ผลที่ควรจะได้จากการทดสอบ จากวิธีที่ได้นี้ สามารถนำมาสร้างมาตรวัดทางการทดสอบที่สามารถช่วยบอกความก้าวหน้าของการทดสอบได้ ทั้งในเชิงความพยายามที่ได้ลงแรงไปในการทดสอบ และจำนวนกรณีทดสอบที่ทดสอบได้เสร็จสิ้นสมบูรณ์

ในการทดสอบในระดับบูรณาการ ผู้วิจัยได้เสนอวิธีการทดสอบและการสร้างกรณีทดสอบโดยอ้างอิงจากแผนภาพซีเควนซ์ซึ่งเสดงการทำงานร่วมกันของกลุ่มออบเจกต์ ซึ่งวิธีการดังกล่าวได้มีการพิจารณาถึงคุณสมบัติโพลีมอร์ฟิซึมซึ่งเป็นลักษณะเฉพาะของซอฟต์แวร์เชิงวัตถุด้วย นอกจากนั้น ในงานวิจัยนี้ยังได้เสนอ มาตรวัดความครอบคลุมของการทดสอบโดยอ้างอิงจากแผนภาพซีเควนซ์ ซึ่งนอกจากจะใช้เป็นหลักในการสร้างกรณีทดสอบเพื่อให้ได้ความครอบคลุมตามที่ต้องการแล้ว ยังสามารถใช้ในการวัดความก้าวหน้าของการทดสอบได้ ว่าทำได้ครอบคลุมมากน้อยเพียงใด โดยมีเป้าหมายที่การครอบคลุมที่ครบสมบูรณ์

จากวิธีการที่ได้นำเสนอในงานวิจัยนี้ ทำให้สามารถสร้างกรณีทดสอบได้จากข้อกำหนดและแผนภาพยูเอ็มแอล สำหรับซอฟต์แวร์เชิงวัตถุ ทั้งในระดับการทดสอบระดับระบบ และในระดับบูรณาการ โดยที่กรณีทดสอบที่ได้นี้ นอกจากจะช่วยลดระยะเวลาและข้อผิดพลาดที่อาจจะเกิดขึ้นในระหว่างการออกแบบการทดสอบและการลงมือทดสอบแล้ว ยังนำมาใช้สร้างมาตรวัดสำหรับวัดความก้าวหน้าในการทดสอบ ซึ่งเป็นกระบวนการที่สำคัญกระบวนการหนึ่งในการพัฒนาซอฟต์แวร์ ค่าที่ได้จากมาตรวัดที่ได้นำเสนอนี้ นอกจากจะช่วยนำไปช่วยติดตามและประเมินความก้าวหน้าโดยรวมของการพัฒนาซอฟต์แวร์แล้ว ยังสามารถนำไปใช้เป็นส่วนหนึ่งของการประเมินประสิทธิภาพและปรับปรุงกระบวนการทดสอบได้อีกด้วย

ในขั้นตอนสุดท้ายของงานวิจัยนี้ ได้มีการพัฒนาเครื่องมือ ซึ่งเกิดจากการนำหลักการในงานวิจัยมาประยุกต์ใช้ โดยที่เครื่องมือนี้จะช่วยให้การทำงานตามขั้นตอนการสร้างกรณีทดสอบตามที่ได้กล่าวมา สามารถทำได้อย่างอัตโนมัติ เครื่องมือกลุ่มนี้ จะรับแผนภาพและข้อกำหนดของซอฟต์แวร์มาเป็นอินพุต พร้อมทั้งเงื่อนไขอื่นๆ ที่จะกำหนดสำหรับการสร้างกรณีทดสอบ และจะสร้างกรณีทดสอบตามแผนภาพ ข้อกำหนด และเงื่อนไขนั้นๆ

หมายเหตุ ในอนาคตหลังจากได้ปรับปรุงเครื่องมือที่ได้จากงานวิจัยนี้ จะทำการเผยแพร่ในเวบไซต์ http://www.se.cp.eng.chula.ac.th

เอกสารอ้างอิง

[1] T. Suwannasart, "Towards Development of a Testing Maturity Model," Ph. D. Dissertation, Dept. of Computer Science, Illinois Institute of Technology, May 1996.

[2] Software Technology Support Center (STSC), *Software Testing Technologies Report*, August, 1994.

[3] Object Management Group Inc., Unified Modeling Language, August, 2004. Available from: http://www.uml.org

[4] Robert V. Binder, *Testing Object-Oriented System: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[5] D. E. Perry and G. E. Kaiser, "Adequacy Testing and Object-Oriented Programming", Journal of Object Oriented Programming, January/February 1990.

[6] M. D. Smith and D. J. Robson, "Object-Oriented Programming – the Problems of Validation", Proceedings of Conference on Software Maintenance, San Diego, CA USA, 26-29 November 1990, pp. 272-281.

[7] S. Barbey and A. Strohmeier, "The Problematics of Testing Object-Oriented Software", Proceedings of SQM '94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, Volume 2, 1994, pp. 411-426.

[8] R. V. Binder, "Testing Object-Oriented Systems: A Status Report", American Programmer, April 1994.

[9] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "A Test Strategy for Object-Oriented Programs", Proceedings of the 19th International Computer Software and Applications Conference (COMPSAC'95), Dallas, Texas, August 09 - 11, 1995.

[10] K. C. Tai and F. J. Daniels, "Test Order for Inter-Class Integration Testing of Object-Oriented Software", Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC '97), Washington, DC, USA, August 13-15, 1997, pp. 602-607.

[11] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand, "Testing Levels for Object-Oriented Software", Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000, pp. 136-145.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, 1st Edition, Addison-Wesley, 1995.

[13] Cockburn, A *Writing Effective Use cases*. United States of America: Addison-Wesley. 2000.

ภาคผนวก ก

ผลงานวิจัย

บทความเรื่อง

An Instrumentation Model for Supporting Software Testing
Based on UML Sequence Diagrams

นำเสนอใน

The 4th Information and Computer Engineering Postgraduate Workshop
(ICEP 2004)

ระหว่างวันที่ 22-23 มกราคม 2547

จังหวัดภูเก็ต ประเทศไทย

An Instrumentation Model for Supporting Software Testing Based on UML Sequence Diagrams

Siros Supavita and Taratip Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Thailand
Siros.S@student.chula.ac.th, Taratip.S@chula.ac.th

Abstract

As UML Sequence Diagram usually represents an interaction of objects in object-oriented software in a form of a sequence of messages sending between the objects, it is a useful source as the design specification for the implementation. Likewise, testing, especially integration and system testing, potentially benefits from using it as a source of test specification. Whereas the main purpose of the testing is to verify whether the implementation conforms to the design, message sending sequence of the implementation is required to be compared to the one from the design. While the expected message sending sequence is extracted from UML Sequence Diagram, the actual message sending sequence is derived through the instrumentation of test execution. This paper presents a model for representing both message sending sequences, along with the basic guidance of how to apply the model in software testing, as test oracle and test coverage.

4 Introduction

While object-oriented paradigm provides features, like inheritance and polymorphism, that help developers to easily solve problems, it poses difficulties in testing. As the features are specific to the paradigm, testing techniques for structural programming paradigm are not adequately appropriate. Although some techniques, particularly functional testing techniques, may be applicable [1], most are usually not. An empirical study [2] shows that programs designed with object-oriented paradigm usually result in many operations with only simple intraprocedural control flow. This is significantly different from structural programs; therefore, control flow and data flow based testing techniques, which are originally designed for the structural design principle, is not suitable [3]. Moreover, unit testing in object-oriented program is inseparable from integration testing [4].

As UML (Unified Modeling Language), as a modeling language for object-oriented paradigm, is gaining more popularity, it becomes the essential part of many object-oriented software development projects. With its standard notations and semantic, designers can use UML to model their design in an expressive way, and also communicate to others effortlessly. Beside design and implementation, testing based on the model is very desirable. The goal is to verify conformance of the implementation against the design model. Nevertheless, there is no standard way of using UML notations in models and diagrams. Testability requirement for the model must be established so that the notations are uniformly used to produce test-ready model. Several researches have defined usages for their specific test approaches. While some focus on system-level specification [5,6], some focus on internal interaction or implementation [7,8,9,10].

Keeping the issues about object-oriented software testing and testing based on UML in mind, we are working on defining a test approach based on message sending sequence, modeled in UML Sequence Diagram. We focus on testing on system or integration level, as UML Sequence Diagram usually represents interaction of a system or a subsystem. This paper presents a model for representing message sending sequence which is an essential part of the test approach. Section 2 gives an overview of the test approach. Section 3 shows related work. Concept of the model is discussed in section 4, while section 5 explains the model elements. Section 6 gives an example of the model and the application of the model is given in section 7. Finally, section 8 concludes the paper.

2. Testing of Message Sending Sequence

2.1 Message Sending Sequence

Objects or components in object-oriented software interact by sending messages to each other. It is said that a sender object sends a message to a receiver object which is similar to the sender object calls an operation on the receiver object. A message sending sequence is an ordered sequence of message flying between objects in the interaction.

"Message" is a design term and is usually in higher level and conceptual; hence, it can be interpreted into several different meanings in implementation. "An object sends a message to another object" can be interpreted as "an object calls an operation on another object", "an object posts an event which causes another object, as an event handler, to be executed", or even "an object asynchronously calls an

operation on another object". In our current work, we consider only message which is a synchronous operation call.

2.2 Test Approach

From an interaction diagram like UML Sequence Diagram, message sending sequence is defined as the design specification. Programmers use the message sending sequence as the guidance of their implementation of the interaction. For test purpose, it is treated as the expected message sending sequence using as a part of test oracle, in addition to the output. The implementation under test must perform message sending equivalent to the expected message sending sequence.
According to the test approach, instrumentation of test execution is required to capture the actual message sending occurs under the execution. As stated earlier, it is compared to the expected message sending sequence to verify conformance of the implementation against the design model. An overview of how the approach works is shown in Figure 1.

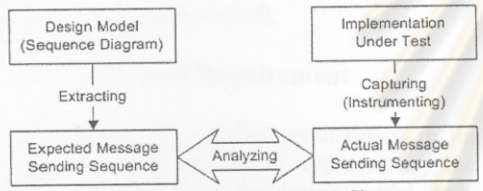


Figure 1. Testing of message sending sequence

We are currently working on defining the systematic procedure to verify the execution result (the actual message sending sequence) against the expected result (the expected message sending sequence). There are issues that need to be considered, regarding message sending sequence analysis. Ideally, the expected and the actual message sending sequence must be identical. Nonetheless, the interaction diagram, as the source of the expected message sending sequence does not usually contain interaction in a great detail like the implementation. Designers ordinarily omit the interaction with objects of common classes, like String in Java, or their own utility or framework classes. As a result, the actual message sending sequence is evidently nonidentical to the expected message sending sequence. Furthermore, polymorphism, which is a specific feature to object-oriented paradigm, is another important issue for sequence comparison.

2.3 Scope

Although our approach aims generally to UML Sequence Diagram, it apparently has limited range of usage. As stated earlier, our focus is on message which is a synchronous operation call, even though UML Sequence Diagram also supports other types of message, like an asynchronous call.
Another limitation lies in the way the interaction diagram is written. The previous section shows that the interaction diagram usually omits some classes, resulting in a gap between the interaction diagram and the implementation. As the gap grows wider, more messages in the actual message sending sequence are missing from the expected message sending sequence. Hence, the accuracy of message sending sequence verification is lessened. This becomes even worse when the diagram is written for high-level design where many parts of the interaction have been omitted from the diagram. Thus we currently limit the source of the expected message sending sequence to only the detail-level design diagram which reflects the main portion of the implementation.

3. Related Work

Fraikin et al. presented a tool, named SeDiTeC [9], to generate test for Java program from UML Sequence Diagram. UML Sequence Diagrams are taken as the test specification and test drivers are generated according to the interaction specified in the diagram. Several diagrams can be combined to form a test scenario; as a result, it allows preparation step, test execution, and result verification step to be written in separated diagrams. Test data is supplied as parameters of operation calls which are identified in the diagrams. Moreover, test stub class can be created for the incomplete implementation.
The test execution is determined as pass or fail by comparing the execution with the Sequence Diagram. The order of call, the object identity, input parameters, and return values specified by the testers, are used for the comparison. When the design model does not fully represent the implementation, the testers are required to identify whether the test is passed or failed. However, it does not provide support for polymorphic interaction.

4. Concept of the Model

4.1 Basic Requirement

Our goal is to have the expected message sending sequence and the actual message sending sequence represented on the same model to ease the sequence verification. The proposed model must have all common features of both message sending sequences, while must also avoid contradiction or confusion from both message sending sequence.

4.2 Instrumentation Model Requirement

Basic requirement of the instrumentation model is that the model must represent message sending sequence between

objects which occurs in an execution of object-oriented software. The model is significantly different from the instrumentation models which capture only function entry/exit in structural program, since object-oriented software makes uses of objects. An object is a cohesive piece of data and operations that manipulate the data. It is different from a structure type in structural programming which contains only data; the manipulation operations live apart. Moreover, each object has an identity of its own which distinguishes itself from other objects. Two objects, although contain identical values of data members, have different identities; hence, they are not identical. This difference is the major requirement of the model.

As polymorphism is a specific feature of object-oriented paradigm, discovering faults related to polymorphism is the main purpose of an object-oriented testing technique. As a consequence, the model must be capable of representing polymorphic server situation.

Beside the issues addressed above, the requirement is similar to usual function entry/exit instrumentation models. The model must be able to address call hierarchy, called function name and arguments (to identify the specific function), and context of the calling (to identify where the function is called).

4.3 Design Model Requirement

The model, which satisfies requirement in the previous section, is also capable of representing the message sending sequence from the design model, as the general information about the message sending sequence (message sender, message receiver, message action, and information about the operation and class where the execution occurs) are similar. Although for now we design the model to support message sending sequence comparison between the execution and the design model, the model should be extensible to support further analysis, for example return condition (usual return or exception thrown), and returned value verification. Since we do not yet complete the requirement for the analysis, this part of the model must be open for possible future extension.

4.4 Assumption and Restriction

First assumption is about the interpretation of message sending as described earlier; only the message sending which is an operation invocation is supported. Another assumption is about thread of execution. It is allowed to present concurrent execution in a single UML Sequence Diagram, by using notations like asynchronous call, return call, and object lifeline. However, testing for concurrent execution is complicated. Specific test techniques and instrumentation techniques are required to support concurrency. As a result, our current model supports only a single thread of execution for both the actual execution and the design model.

As UML Sequence Diagram allows guard conditions to be attached to messages in a diagram, a single Sequence diagram can represent a main scenario with several alternatives. Our model is a representative of one scenario; therefore, guard conditions are not currently considered. Further extension of this model, however, might include guard conditions for analysis purpose.

5. Design of the Model

Although UML provides semantic that covers wide array of object-oriented modeling including structural model and behavioral model, the structure of message sequence in UML, where all messages are grouped into an ordered list, is not appropriate to represent execution sequence in our case. Our model has a structure similar to dynamic call graph [11], which does not share a node when there is more than one call to an operation.

The model is presented in class diagram as shown in Figure 2. In the subsequent subsections, each element in the instrumentation model is discussed and possibly compared to UML Sequence Diagram element to give an example of how the design model might be extracted and compared to the execution.

5.1 ExecutionContext

An ExecutionContext element holds information about an execution of an operation under a specific circumstance. An execution may send messages to instances and results in executions in other contexts. Therefore, a context can be a message sender, a message receiver, or both.

As a message sender, a context has a reference to an ordered list of messages it sends. The messages are sorted in chronological order. Each message represents call and implicit return of the call. Using depth first traversal on the context hierarchy yields the sequence of message sending for entire interaction in timely order.

For the execution model, there are 2 aspects of operation attached to execution context, a direct associated operation in the execution context as an operation actually executed, and an operation associated to the stimulus message as an operation the sender intends to invoke. This is for representation of polymorphism in the execution. For the design model where polymorphic interaction is usually not explicitly presented, both operations are identical.

Unless the execution is on class operation, an execution context must be associated to an instance. The instance must be an instance of the class where the operation of the execution context and the operation of the stimulus message are defined or inherited. This is not an issue in some object-oriented programming languages (i.e. Java, C++), for they perform static type checking which already prevents undeclared operation calls on an object [12].

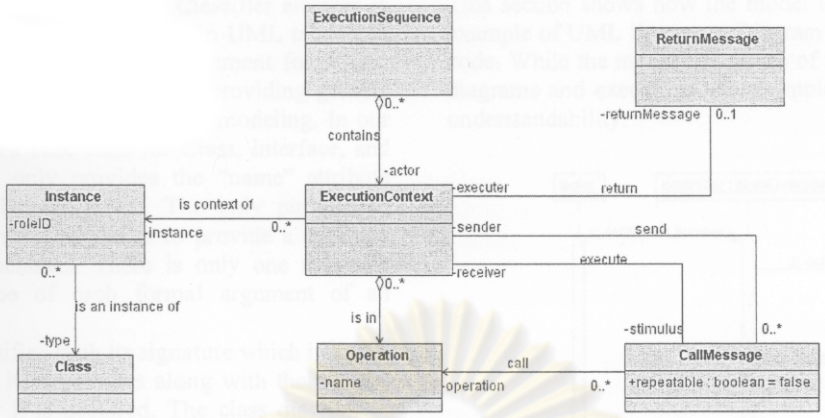


Figure 2. Message sending sequence model

5.2 Instance

An Instance element is a representative of an object in object-oriented program. It is similar to ClassifierRole in UML. The major difference is the association with type information. ClassfierRole can have more than one base Classifier, because an object in the collaboration can play roles of several Classifiers. However, an object in the implementation must have a type as a concrete class.

In an interaction, several instances of the same class possibly play different roles. It is important for the model to identify instance in the execution context, as the verification needs to compare the instances of the sender and the receiver of each message as well as the message itself. In the design model, instances are identified by assigning a role to each instance. The instrumentation model must also support instance identification by, for example, using generated object runtime ID. Although mapping between roles in the design and the execution is not possible, improper instance usage is recognizable.

5.3 CallMessage

A CallMessage element is a representative of a message sent from a sender object to a receiver object. As stated earlier, our focus is on an operation call message, which is similar to CallAction in UML semantic. A message has references to its sender and its receiver, which are both ExecutionContext elements. The sender execution context is the context where the message is issued, while the receiver execution context is the context that the receiver instance executes as the effect of the message. As an execution context element has a reference to an operation it is associated to, we can trace from a message to the operation which is the origin of the message and the operation which is the target of the execution.

In addition to the context of the sender and the receiver, a CallMessage element also has a reference, named "call", to an operation as the target of the call. The purpose of the reference is to accommodate polymorphic interaction. Along with the operation associated to the execution context of the receiver, the call association on the call message is the key to reveal defects in polymorphic implementation.

5.4 ReturnMessage

A ReturnMessage element contains information about return value or return condition. As stated earlier, the return information from both the design model and the instrumentation model are important if we want to perform analysis on the execution result or condition, in addition to message sending sequence. For design model, this element may contain information specified in the model as return message of the execution, if any. For the execution model, it may represent return value or condition captured from the execution, probably the thrown exception. However, we do not have a particular requirement on this issue yet, so the element is currently empty and optional.

5.5 ExecutionSequence

An ExecutionSequence element is a container element which wraps all other elements. It has an association to an execution context, which is the context of the actor of the interaction. The actor, as the stimulus, initiates the message sequence by sending messages to other objects. This element may contain other information which is applied to all elements in the interaction as global attributes of the interaction; for example, an identifier which is assigned to a particular execution scenario for further reference or the condition of the scenario collected from guard conditions of the diagram.

5.6 Classifier and Operation

Though sharing the same name, the Classifier element in this model is different from Classifier in UML model. In UML, a Classifier element is a base element for others, like Class, Interface, Components etc., providing general attributes to support different aspects of modeling. In our model, Classifier, as a base class for Class, Interface, and DataType elements, only provides the "name" attribute common to all of its subclasses. The only purpose of Classifier element in our model is to provide a common structure for its subclasses. There is only one usage of Classifier, as a type of each formal argument of an operation.

An operation is identified with its signature which includes its name, the list of its arguments along with their types, and the class where it is declared. The class diagram of Classifier and Operation is shown in Figure 3.

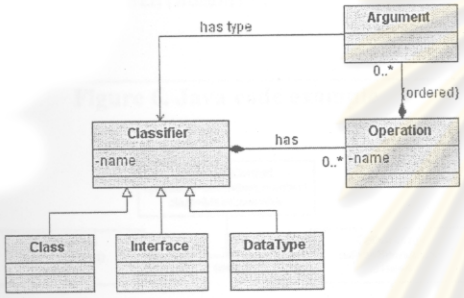


Figure 3. Model for structural part

5.7 Class, Interface, and DataType

As described in previous section, Class, Interface, and DataType elements are subclasses of Classifier element. They inherit an attribute, name, from Classifier element. The attribute represents name of class, interface, and data type respectively.

We designed these elements based on well-known object-oriented programming languages, Java and C++. Class element is apparently required as these languages are class-based. These languages, while provide support for most of object-oriented features like inheritance and dynamic binding, are classified as hybrid [12]. They do not require everything to be an object, so there are pieces of data that are defined as primitive types, which are represented by DataType elements in our model. In addition, Java has an additional construct, interface, which is considerably different from class in many senses; as a consequence, Interface element is designed to support this construct.

6. Example

This section shows how the model looks like for a given example of UML Sequence Diagram and a snippet of Java code. While the model is capable of representing complex diagrams and execution, the example is rather simple for understandability.

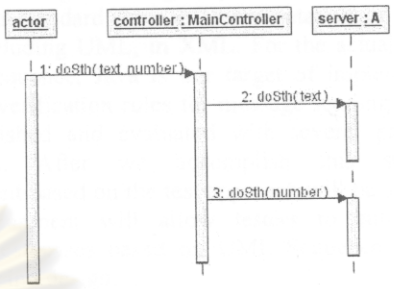


Figure 4. An example of UML Sequence Diagram

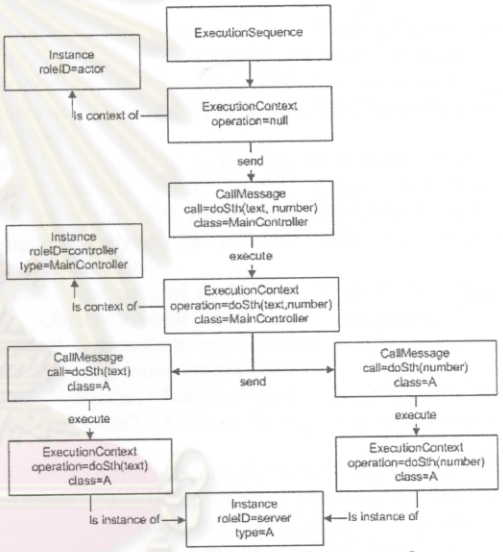


Figure 5. Design model example

Figure 4 shows a simple example of Sequence Diagram. An actor sends a message to controller, an instance of class MainController. The instance, in turn, sends 2 messages to server, an instance of class A; as a result, 2 operations on server, doSth(text) and doSth(number), are executed respectively. The message sending sequence model extracted from the diagram is shown in Figure 5. The example model shows that in the design model, as previously discussed, the operation associated to an execution context is identical to the operation associated to its stimulus message. Moreover, how overloading methods are modeled is also shown in the example model.

Figure 6 shows a snippet of Java code which is an example of implementation of the diagram in Figure 4. When the controller instance is instantiated with an instance of class A, the message sending sequence captured from the

execution is identical to the one in Figure 5. However, it is slightly different, when an instance of any subclass of class A is supplied for the instantiation. Figure 7 shows the part that is different from Figure 5 with the different elements highlighted, for the case where an instance of class B, as a subclass of class A, is supplied rather than an instance of class A.

```
class MainController {
    protected A server;

    public MainController(A server) {
        this.server = server;
    }

    public void doSth(String text, int number) {
        ...
        server.doSth(text);
        ...
        server.doSth(number);
        ...
    }
}
```

Figure 6. Java code example

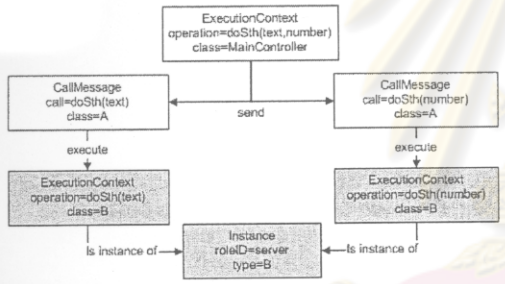


Figure 7. Partial instrumentation model example

7. Application

The model presented in this paper can be applied to several tasks in testing. Application of the model as test oracle, as emphasized in this paper, is the most important one. This model provides fundamental for defining test generation approach, where execution of generated test case yields the message sending sequence equivalent to the model.

Besides, the model is also applicable as test coverage model. When the Sequence Diagram contains branch conditions, basic coverage criteria similar to control flow based testing can be applied. As the model is capable of representing polymorphic interaction, it is also possible to define coverage criteria based on polymorphism for the interaction.

8. Future Work and Conclusion

In this paper, we propose a model for representing message sending sequence of object-oriented software. The major purpose of the model is to support testing based on message sending sequence which we are currently working on. We plan to implement extraction of the expected message sending sequence from XMI format which is a standard form of representation of metadata model, including UML, in XML. For the actual message sending sequence, Java is our target of implementation. Next, the verification rules for message sending sequence are established and evaluated with several patterns of interaction. After we accomplish this step, test environment based on the test approach will be developed. The environment will allow testers to automatically generate test cases based on UML Sequence Diagrams written during design.

9. References

[1] A. Jefferson Offutt, Alisa Irvine, "Testing Object-Oriented Software Using the Category-Partition Method", 17th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95), Santa Barbara, CA, August 1995.

[2] Amie L. Souter, Lori L. Pollock, Dixie Hisley, "Inter-class Def-Use Analysis with Partial Class Representation", Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Toulouse, France, 1999.

[3] Amie L. Souter, Lori L. Pollock, "OMEN: A Strategy for Testing Object-Oriented Software", Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States, 2000.

[4] Robert V. Binder, "Testing Object-Oriented Systems: models, patterns, and tools", Addison Wesley, ISBN 0-201-80938-9, 1999

[5] Jeff Offutt, Aynur Abdurazik, "Generating Tests from UML Specifications", 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, October 1999.

[6] Lionel Briand, Yvan Labiche, "A UML-based Approach to System Testing", Journal of Software and Systems Modeling (Springer) Volume 1 Issue 1, 2002.

[7] Aynur Abdurazik, Jeff Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", The 3rd International Conference on the Unified Modeling Language (UML'00), York, UK, October 2000

[8] Ye Wu, Mei-Hwa Chen, Jeff Offutt, "UML-based Integration Testing for Component-based Software", The 2nd International Conference on COTS-Based Software Systems (ICCBSS), Ottawa, Canada, February 2003.

[9] Falk Fraikin, Thomas Leonhardt, "SeDiTeC – Testing Based on Sequence Diagrams", Proceedings ASE 2002 7th IEEE International Conference, 2002

[10] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger, "UML-Based Integration Testing", Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States, 2000.

[11] S. L. Graham, P. B. Kessler, M. K. McKusick, "gprof: a Call Graph Execution Profiler", Proceedings of the 1982

SIGPLAN symposium on Compiler construction, Boston, Massachusetts, United States, 1982.
[12] J. Voegele, "Programming Language Comparison", http://www.jvoegele.com/software/langcomp.html, November 2003.
[13] The Object Management Group, "OMG Unified Modeling Language Version 1.4", http://www.omg.org, September 2001.

ภาคผนวก ข

ผลงานวิจัย

บทความเรื่อง

Adequacy Criteria for Testing Polymorphism in the Context of Interactions

นำเสนอใน

The International Conference on Software Engineering Research and Practice (SERP'04),

ระหว่างวันที่ 21-24 มิถุนายน 2547

Las Vegas, Nevada, U.S.A

Adequacy Criteria for Testing Polymorphism in the Context of Interactions

Siros Supavita and Taratip Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok, Thailand 10330
Siros.S@student.chula.ac.th, Taratip.S@chula.ac.th

Abstract

Testing is very important for polymorphic interactions, as the exact interactions or operation calls are unpredictable during design. Therefore, they are vulnerable to defects. Adequacy criteria are required to address thoroughness of testing in this context. Although there are other proposed criteria that concern about inheritance and polymorphism, none of them aims at the context of interactions. This paper proposes 5 adequacy criteria in the view of inheritance and polymorphism. Combined with other criteria from interaction view, they can be applied as criteria to generate and select test cases for object-oriented interaction testing.

Keywords: Software Testing, Test Adequacy Criteria, Object-Oriented, Interactions, Polymorphism

1. Introduction

As object-oriented software testing requires significantly different concerns from procedure-oriented software testing, specific test approaches for object-oriented paradigm are necessary [1,2,3,4,5]. As the unique features of the paradigm, including encapsulation, inheritance, and polymorphism, cause faults that do not exist in procedure-oriented software, the test approaches must aim to uncover these kinds of faults. Besides, it is addressed clearly in [2] that these features pose some difficult issues for testing and result in test adequacy criteria that are different from procedure-oriented software.

Adequacy criteria are defined specifically for each test approach, as each of them concerns with faults on different angles. In this paper, we explore an idea of adequacy of testing object-oriented software, focusing on inheritance and polymorphism features in the context of interactions, as we are working on defining a test approach for testing interactions defined in UML Sequence Diagrams. Given an interaction or, strictly speaking, a polymorphic interaction, the effects of inheritance and polymorphism under the interaction execution and testing are important to design an appropriate method for test generation and selection. Even also focused on inheritance and polymorphism, other researches about adequacy criteria focus on the problem in different ways.

The paper is organized as followed. Section 2 gives necessary background in adequacy criteria, inheritance, and polymorphism. Related works are reviewed in section 3. Our criteria are presented in section 4, while section 5 compares and discusses about their efficiency and subsumption. A case study is illustrated in section 6, and the paper is concluded in section 7.

2. Background

2.1 Adequacy Criteria

Adequacy criteria are always an essential part of software testing approaches. Without adequacy criteria, thoroughness of testing cannot be evaluated, and neither do correctness of the software under test. An adequacy criterion expresses whether a given test set is appropriate for uncovering faults in software under test according to a particular testing technique. It keeps a test set from being an exhaustive test set, which requires a very long time and a lot of effort to execute. This is not cost-effective and, sometimes, impossible in practice. Testing software with an adequate test set guarantees that the software is tested to some acceptable degree of thoroughness; as a consequence, quality or correctness of the software is trusted to the level corresponding to the criterion.

For program-based testing approaches, an adequacy criterion is defined based on basic building blocks of the artifacts under test [6]. For example, adequacy criteria of path testing (i.e. branch coverage criterion etc.) [7] are defined based on elements in the control flow graph of the program under test.

2.2 Inheritance & Polymorphism

As very convenient features of object-oriented paradigm, inheritance and polymorphism can help to solve many problems with less complicated design. One of their important characteristics is instance substitution under a polymorphic interaction, where an instance of a class may be substituted, under the actual execution, by an instance of any subclasses of the class. With a little help from creational patterns like Abstract Factory [8], new classes can be introduced to inheritance hierarchies without any modification to the interaction code. However, it does not mean that testing for the additional classes is unnecessary [2,5].

There are variants of inheritance from different aspects. The effects of strict inheritance, subtyping, and subclassing under software testing are discussed in [5], and they all are also considered in this paper.

3. Related Works

3.1 Criteria for Testing Polymorphic Relationships

Alexander and Offutt present four test adequacy criteria for testing polymorphic relationships in object-oriented software in integration testing level [9]. These criteria are designed to support their test approach [10], which is an altered version of coupling-based testing technique [11]. Basically the criteria are defined around "defs" and "uses" like criteria for data flow testing [7]. Concerns about inheritance and polymorphism are applied, as two of the criteria require all subclasses to be tested in the interaction context of their respective superclasses.

3.2 Test Adequacy Criteria for UML Design Models

Andrews et al. propose adequacy criteria for testing UML design models, which include class diagrams and interaction diagrams [12]. Testing is performed on the design models using symbolic execution techniques rather than on the implementation of the models. The criteria are defined based on their respective diagram elements in several different aspects. For example, association-end multiplicity (AEM) criterion, which is defined based on association relationships in class diagrams, requires an adequate test set to cover scenarios where various numbers of instances are put on the opposite end of an association.

They also conducted an experiment to see whether test cases created based on the criteria are effective in detecting faults in design models. The result shows that some selected criteria are effective in revealing faults, including incorrect sequence numbering, missing flows, and data flow gaps.

4. Our Criteria

There are two important concerns in defining the criteria. The first concern is about how thorough an inheritance hierarchy is tested under a polymorphic interaction. Apparently there are two choices; whether to include all subclasses to be tested in the context of their superclass, or to just include only the subclasses which override the method under test. Although there is no practical evidence about necessity of testing all subclasses in the context of the superclass, several research studies address this issue [2,13]. From designers' point of view, inherited methods seem to be ready for use right away, as inheritance is supposed to promote reusability of the inherited method. However, the results from the studies, though counter-intuitive, give strong reasons to test all subclasses in the context of the superclass. Other researches, which focus on testing inheritance and polymorphism features, adopt this idea into their adequacy criteria [9,12]. In our research, we also embrace this idea as an adequacy criterion, as well as other more relaxed criteria as options.

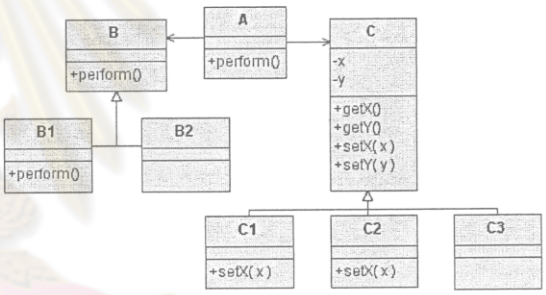


Figure 1. Class Diagram for the example

Another concern is in the situation where an interaction comprises more than one class with inheritance hierarchy. Say there are classes as shown in Figure 1, and their interaction is shown in Figure 2. From the diagrams, both class B and class C, which both have their own inheritance hierarchies, involve in the interaction. Supposed that the test adequacy criterion requires all subclasses from each inheritance hierarchy to be tested, there will be 3 classes from class B hierarchy (B, B1, and B2) and 4 classes from class C hierarchy (C, C1, C2, and C3) to be tested. A decision is required to be made, as either the classes are tested in combination (like all paths coverage in path testing), or just cover each class at least once. It is clear to see that the first option results in a test set that grows multiplicatively, while the latter yields a test set that is limited to the maximum number of classes in each inheritance hierarchy. Indeed, the first option is more desirable in testers' perspective, as it covers all combinations of subclass substitution and results in a more thorough test set. However, it possibly results in a

massive test set for situations like the example, where 12 test cases (the product of 3 and 4) are required for testing such a simple interaction without any condition or loop.

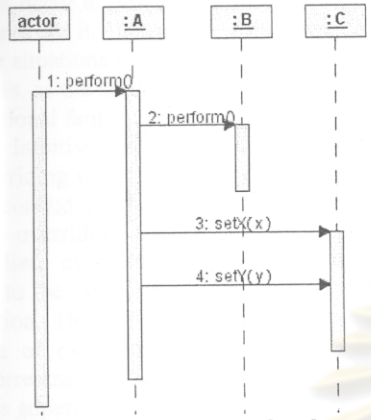


Figure 2. Sequence Diagram for the example

From these concerns, five criteria are defined as shown below. Table 1 shows an example of class selection set of test cases for the interaction in Figure 2 corresponding to each criterion, while the subsumption hierarchy of the criteria is shown in Figure 3. Note that although, inheritance coverage and strong overriding method coverage criteria are aligned on the same level, they are not comparable.

4.1 Base Class Coverage

This criterion requires no superclass-subclass substitution. Classes that must be covered under test are only those defined in the interaction.

4.2 Overriding Method Coverage

This criterion requires each class in an interaction to be substituted by its subclasses which override the method involved in the interaction.

4.3 Inheritance Coverage

Subsuming the previous criterion, this criterion requires each class in an interaction to be substituted by all of its subclasses.

4.4 Strong Overriding Method Coverage

Based on overriding method coverage criterion, this criterion is an extension to support the case where an interaction comprises more than one class with inheritance hierarchy. In addition to cover all overriding methods, this criterion requires testing for all combinations of all choices of substitution.

4.5 Strong Inheritance Coverage

Similar to strong overriding method coverage, this criterion is an extension of inheritance coverage criterion to support several subclass substitutions in an interaction. All substitutable classes are required to be covered in combinations.

Table 1. Example of class selection sets of test cases

Criteria	Class Selection Sets*
Base Class Coverage	{B,C}
Overriding Method Coverage	{B,C} {B1,C1} {B,C2}
Inheritance Coverage	{B,C} {B1,C1} {B2,C2} {B,C3}
Strong Overriding Method Coverage	{B,C} {B,C1} {B,C2} {B1,C} {B1,C1} {B1,C2}
Strong Inheritance Coverage	{B,C} {B,C1} {B,C2} {B,C3} {B1,C} {B1,C1} {B1,C2} {B1,C3} {B2,C} {B2,C1} {B2,C2} {B2,C3}

* As class A is always required for all test cases, it is left from the class selection sets.

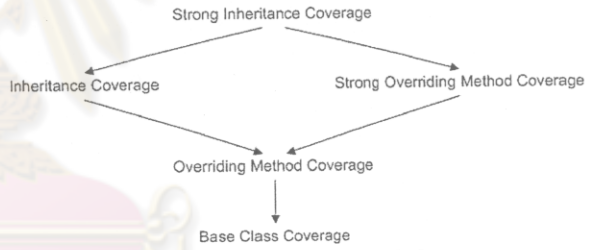


Figure 3. Subsumption hierarchy of the criteria

5. Discussion

As the weakest criterion, base class coverage criterion is apparently weak, inappropriate, and only useful in some exceptional cases. Such a case is where a subclass is the pure extension of its superclass (no overriding), and analysis result shows that the class does not involve in multiple context scenario. This situation is free from ITU (Inconsistent Type Use) fault presented by Offutt et al. [13], as the extension methods, which can cause failures, are not accessible in the context of the superclass, and the absence of multiple context scenario also keeps them from being accessed in the context of the subclass as well.

Another situation that this criterion is appropriate is where the inheritance is subclass inheritance rather

than subtype inheritance. Subclass inheritance, as a relaxed version of subtype inheritance, does not preserve substitutable property of inheritance, since it rather aims at reusability of attributes and methods of the superclass. Superclass-subclass substitution does not occur at any time; consequently, no testing is required for it. Using base class coverage criterion in these situations cuts down the number of required test cases, as the additional test cases tend to uncover no additional fault.

It is intuitive that when a method is overridden, the overriding version of the method should be tested in the context of the overridden method to ensure that the overriding method is correct. According to this belief, overriding method coverage criterion seems to be appropriate for testing polymorphic interaction. However, showing in the studies, the absence of overriding method does not guarantee total correctness. While a subclass inherits a method from its superclass, correctness of the method is not inherited to the context of the subclass. There may be side effects from situations outside the scenario of the interaction under test like, for example, usage of multiple context (casting) and difference in initialization (or constructors) of superclass and subclass. Testing is usually required for the entire inheritance hierarchies to ensure complete correctness, whether overriding or not; therefore, inheritance coverage criteria is considerably better in terms of reliability and thoroughness. However, overriding method coverage criterion can provide a reasonable trade off in the situation where base class coverage criterion is inappropriate, and inheritance coverage criterion requires an unacceptable amount of effort.

Advancing the regular criteria by exercising all combinations of substitutable classes, strong overriding method coverage and strong inheritance coverage criteria are designed to cope with an interaction, which includes more than one class with inheritance hierarchy. While they are very effective to reveal faults which occur in the situation of a particular combination of class substitution, they may require a tremendous test set. In the situation where, in an interaction, two objects with inheritance hierarchies do not interact with each other directly, there is a very little chance that faults could occur from the combinations of class substitution. In such a case, either one of the regular criteria is possibly more cost-effective than its strong version criterion, as the excessive test cases are eliminated while comparable test effectiveness is achieved.

6. A Case Study

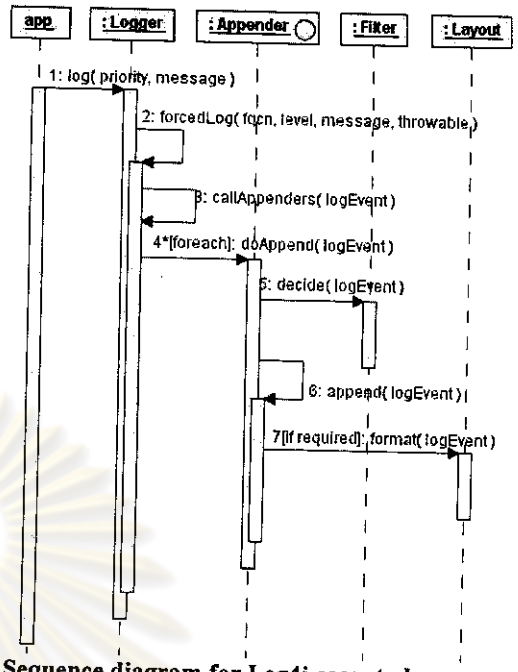


Figure 4. Sequence diagram for Log4j case study

We select Jakarta Log4j, an Apache open source project [14], to be analyzed as our case study. Log4j is a framework providing logging mechanism to any application developed under Java platform. Underneath its simple API, the logging mechanism is configurable, as the logging interaction is designed to be polymorphic. Basically, the logging interaction comprises several main classes as shown in UML Sequence Diagram in Figure 4. The diagram represents an overview of the logging interaction. (Although there are a lot more classes involved in the actual interaction, only the classes that we focus are shown to simplify the example). Appender, Filter, and Layout have their own inheritance hierarchies shown in Figure 5 and 6, as any instance of their concrete subclasses can be substituted in the interaction to form any desirable logging capability. For example, substitution of an instance of ConsoleAppender class in the place of Appender interface causes log messages to go to the console of the application. Each implementation of Appender, Filter, and Layout must override append, decide, and format methods respectively to implement their specific functionality.

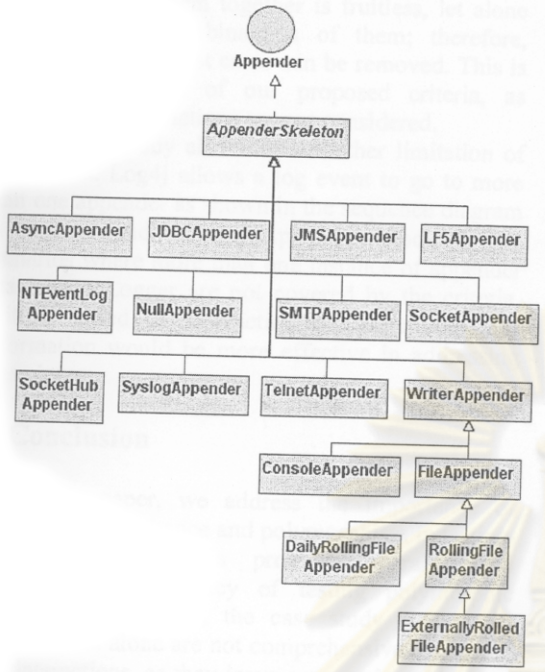


Figure 5. Class diagram of appenders

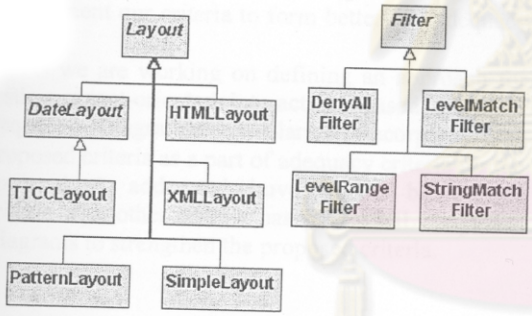


Figure 6. Class diagram for layouts and filters

6.1 Base Class Coverage

Testing this interaction using base class coverage criterion requires only one test case. Since Appender is an interface, and Layout and Filter are abstract classes, it is impossible to have direct instances of these types. Therefore, the test case must rather use instances of their concrete subclasses/implementations instead. As the criterion does not consider polymorphic interaction, any single selection of the classes satisfies the criterion. This criterion is obviously inappropriate for testing the interaction, as subclass substitution is fundamental under actual execution.

6.2 Overriding Method Coverage and Inheritance Coverage

All concrete subclasses of Appender, Filter, and Layout have overriding methods; consequently, applying overriding method coverage and inheritance coverage criteria yield similar test sets. The numbers of concrete implementations of Appender, Filter, and Layout are 17, 4, and 5 respectively; hence, the total number of required test cases is 17.

6.3 Strong Overriding Method Coverage and Strong Inheritance Coverage

Strong overriding method coverage and strong inheritance coverage criteria require significantly more test cases than overriding method coverage and inheritance coverage criteria. Considering only the concrete implementations, 340 test cases (17 x 4 x 5) are required to satisfy either one of the strong criteria. Although either overriding method coverage or inheritance coverage criterion requires a reasonably small amount of test cases, they seem unable to uncover faults from various combination of class substitution. Either strong overriding method coverage or strong inheritance coverage criterion seems to be more effective in detection of these faults, but the massive amount of required test cases does not seem to be cost-effective in practice.

6.4 Improvement

Selectively applying the criteria to different parts using some domain knowledge could yield a more effective test set, in both terms of cost and possibility of defect detection. From analysis of the interaction and the implementation, it is clear that Appender implementations deal with any Filter implementations in only doAppend method in AppenderSkeleton abstract class, the superclass of all Appender implementations. If the interaction between AppenderSkeleton and Filter is tested with one of their implementation selection, it is less likely that other selections would introduce a new defect. Testing combinations of these classes seems to uncover no additional defect; consequently, some test cases can be removed without significant effect.

However, it is different for the interaction between Appender and Layout. Each implementation of Appender interface differently interacts with Layout; for example, they call the format method on Layout in different sequence or, in some cases, the method is not called at all (as some implementation of Appender does not require Layout, as marked in guard condition in Figure 4). As a result, it is still necessary to test combinations of these classes. However, another improvement comes from this fact that some Appender implementations do not require

Layout. Testing them together is fruitless, let alone testing various combinations of them; therefore, some unnecessary test cases can be removed. This is one of limitations of our proposed criteria, as conditions in interactions are not considered.

The case study also shows another limitation of the criteria. Log4j allows a log event to go to more than one appender as shown in the sequence diagram as iteration condition of doAppend method call. The scenarios where more than one instance of appender attached to Logger are not covered by the criteria. Criteria based on interaction diagrams with this information would be more effective in addressing these issues.

7. Conclusion

In this paper, we address the importance of concerns of inheritance and polymorphism in testing. Our proposed criteria provide several useful guidelines for adequacy of testing polymorphic interactions. However, the case study shows that these criteria alone are not comprehensive enough to test interactions, as they leave some vulnerable parts of the interaction untested. Adequacy criteria from other aspects; for example interaction condition and multiplicity of association ends, are required to complement our criteria to form better test adequacy criteria.

As we are working on defining an approach for testing object-oriented interactions based on UML Sequence Diagrams, we plan to incorporate the proposed criteria as a part of adequacy criteria for the approach. As addressed above, we will blend these criteria with other criteria based on UML Sequence Diagrams to strengthen the proposed criteria.

8. Acknowledgement

This research is supported by TJTTP-OECF (Thai-Japan Technology Transfer Project - Japanese Overseas Economic Cooperation Fund) and Department of Computer Engineering – Industry Linkage Research Project, year 2004, Chulalongkorn University. The authors would like to thank Prof. Koichiro Ochimizu of JAIST (Japan Advance Institute of Science and Technology) for his useful comments and suggestions.

9. References

[1] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[2] D. E. Perry and G. E. Kaiser, "Object-Oriented Programs and Testing", *Journal of Object Oriented Programming*, January/February 1990

[3] J. H. Hayes, "Object-Oriented Programming Systems (OOPS): A Fault-Based Approach", *Proceedings of International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, Palermo, Italy, September 1994, pp. 205-220.

[4] E. V. Berard, "Issues in the Testing of Object-Oriented Software", *Proceedings of Electro'94 International*, IEEE Computer Society Press, 1994, pp. 211–219.

[5] S. Barbey and A. Strohmeier, "The Problematics of Testing Object-Oriented Software", *Proceedings of SQM '94, Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, volume 2, 1994, pp. 411-426.

[6] E. J. Weyuker, "The Evaluation of Program-Based Software Test Data Adequacy Criteria", *Communications of the ACM*, Volume 31, Issue 6, June 1988.

[7] B. Beizer, *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold Co., 1990.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, 1st Edition, Addison-Wesley, 1995.

[9] R. T. Alexander and A. J. Offutt, "Criteria for Testing Polymorphic Relationships", *Proceedings of the International Symposium on Software Reliability and Engineering (ISSRE00)*, IEEE Computer Society, San Jose CA, 2000, pp.15-23.

[10] R. T. Alexander and A. J. Offutt, "Analysis Techniques for Testing Polymorphic Relationships", *Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, Tokyo, Japan, 2000, pp. 172-178.

[11] Z. Jin and A. J. Offutt, "Coupling-Based Integration Testing", *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '97)*, Montreal, Canada, October 1996, pp. 10-17.

[12] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test Adequacy Criteria for UML Design Models", *Software Testing, Verification, and Reliability Journal*, Volume 13, Number 2, June 2003.

[13] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A Fault Model for Subtype Inheritance and Polymorphism", *Proceedings of Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, Hong Kong, PRC, November 2001, pp. 84-95.

[14] Apache Software Foundation, Jakarta Log4j, http://logging.apache.org/log4j/docs/, 2004.

ภาคผนวก ค

ผลงานวิจัย

บทความเรื่อง

Testing Polymorphic Interactions in UML Sequence Diagrams

นำเสนอใน

International Conference on Information Technology Coding and Computing

(ITCC 2005),

ระหว่างวันที่ 4-6 เมษายน 2548

Las Vegas, Nevada, U.S.A

Testing Polymorphic Interactions in UML Sequence Diagrams

Siros Supavita and Taratip Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok, Thailand 10330
Siros.S@student.chula.ac.th, Taratip.S@chula.ac.th

Abstract

Nowadays UML based testing becomes more and more noteworthy, as UML is broadly accepted as de facto standard for object-oriented modeling language. A lot of researches propose test approaches for several types of UML diagrams, focusing on various viewpoints. An approach for testing polymorphic interactions, which are defined in UML Sequence Diagrams, is presented in this paper. From our observation, there are several forms of polymorphic interactions, which have their polymorphic behavior controlled by distinct factors. A method to test each form is determined from these factors. If an interaction is in one of the polymorphic forms defined in this paper, the factors that affect polymorphic behavior of the interaction can be addresses. Finally test cases for the interaction can be created for testing its polymorphic behavior.

Keywords: Software Testing, Object-Oriented, Polymorphism, UML, Sequence Diagram

1. Introduction

UML [1] is getting more popular as a standard modeling language for object-oriented analysis and design. With its well-understood notations, it provides a wide array of diagrams, which help designers to effectively communicate with developers in many different aspects. As an interaction diagram, UML Sequence Diagram shows a design of an interaction among a group of objects under a particular scenario. Basically, an interaction is represented as a sequence of messages sending between objects. Conformance-directed testing [2] aims at uncovering any discrepancy between specification and implementation. Based on UML Sequence Diagrams, this kind of testing is to determine whether the implementation produces the equivalent message sending sequences as defined in the UML Sequence Diagrams as the specification.

In addition to control flow, testing object-oriented interactions also needs to take into account the effect of polymorphism. In our previous work [3], we present an instrumentation model for message sending sequences. The model supports comparison of the expected message sending sequence against the actual message sending sequence, which is not necessarily exactly identical to each other, due to the effect of polymorphism. However, it is required that a polymorphic interaction is tested with particular scenarios to ensure that particular subclasses are tested in the interaction as instances of their superclass, which is defined in the interaction.

For a polymorphic interaction, there are factors that influence its polymorphic behavior. Test cases must be designed around these factors in order to execute required classes under particular scenarios. This paper discusses several forms of polymorphic interactions and their factors. For a polymorphic interaction under test, it is necessary that the factors are identified. This paper shows the basic pattern and factors of each form along with guideline for test design.

This paper is organized as follow. Related background knowledge about object-oriented software testing is explained in Section 2, while Section 3 reviews our test approach. In Section 4, the forms of polymorphic interactions are presented along with some examples of the forms. A discussion about variations of the forms is given in Section 5. The paper is concluded in Section 6 with potential future work.

2. Object-Oriented Software Testing

2.1 Polymorphism and Testing

Polymorphism means the ability to take several forms [4]. In other words, an instance of a class can be handled as an instance of one of its superclass. In cooperation with dynamic binding, it is a very useful feature, as it allows a program to have dynamic behavior according to the actual type of object under execution. A number of design patterns [5] are formalized based on this concept.

Despite its usefulness, a number of researches address that polymorphism does not prevent the software from defects [2,6,7,8]. Perry and Kaiser state that both inherited and overriding operations require new test sets, which are different from the test set for their superclass [6]. Furthermore, an object reference in runtime can be substituted by a reference to the class, which is not the one defined in

the code. This could make behavior of the program unpredictable, which is not good in testing point of view [7].

There are several terms defined by Meyer [4] which are used in this paper. "Polymorphic assignment" is the situation when a reference is assigned with an object which is not its exact type, but one of its subclasses. "Polymorphic entity" is a reference which appears in a polymorphic assignment.

2.2 Testability

Testability is always one of the most important issues in testing. In [9], Binder states that testability consists of 2 important factors: controllability and observability.

Controllability is the capability of controlling inputs of the program under test. Lack of controllability results in low testability, as it is difficult to test the program under test as intended. The term "input" also includes data that are not explicitly passed to the program. As an object can have its own internal state, an operation on the object can have execution logics according to the state. Usually encapsulation and information hiding prevent object states from being accessed directly. This could become a big obstacle in testing, since object states as ones of inputs cannot be controlled by test drivers. A common workaround for this problem is to temporarily break encapsulation by adding public methods to access and mutate the states during the test.

Observability is the ability to examine intermediate outputs of the program under test. It is important to have access to these outputs in order to determine whether the test passes or fails. Similar to controllability, encapsulation is also the major obstacle for observability.

Controllability is critical in the discussion of this paper, since it is required that a polymorphic interaction under test is controlled to be executed with a particular set of classes. From now on, it is assumed that whatever input is discussed, it is always accessible to test drivers. Therefore, there is no controllability issue.

3. Message Sending Sequence Testing

3.1 Concept

Message sending is a way to express object communication in an interaction. A sender object sends a message to request a receiver object to fulfill a particular service [4]. A message sending sequence is a sequence of messages flying between objects in an interaction under a specific scenario.

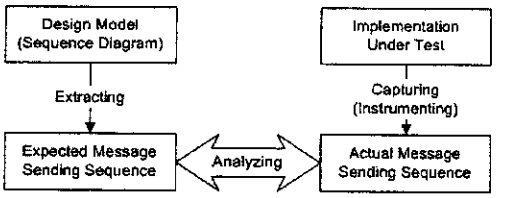


Figure 1. Overview of the comparison of message sending sequences

In addition to checking the actual test result with test oracle, the comparison of message sending sequences is the key of this test approach. The approach aims to detect any discrepancy between interactions in the design model and the implementation. This test approach is conformance-directed testing rather than fault-directed testing [2], as there is no specific fault model. Any difference between the message sending sequences is considered as a defect. As shown in Figure 1, the expected message sending sequence, which is extracted from the design model, is analyzed against the actual message sending sequence, which is instrumented from the execution of the implementation. A model which represents both types of message sending sequences is presented in our previous work [3]. The model captures all information necessary for comparison of the message sending sequences, particularly polymorphism related information.

3.2 Refinement Issue

In design model, all detailed information is not usually presented. Refinement is an action of implementing the software slightly differently from the design model usually by adding implementation level detail, which is intentionally omitted from the design model. With refinement, the message sending sequence from the implementation is not identical to the one from the design model, i.e. UML Sequence Diagram, although the implementation is correctly implemented based on the diagram.

For example, common classes, which provide elementary services like String and Integer in Java, are omitted from the design model for simplicity in the design model, although they are presented and required in the implementation. Comparison of message sending sequence must take care of identifying refinement messages; therefore, the actual message sending sequence of an implementation with refinement can be compared against its design model.

4. Forms of Polymorphic Interactions and Testing

This section discusses forms of polymorphic interaction and how to test them. To thoroughly test a polymorphic interaction requires that all subclasses are substituted to the polymorphic entity; as a consequence, each test case must be designed to execute a particular subclass.

Polymorphism could only happen when there is polymorphic assignment in the interaction. Forms discussed in this section differ from each other in the patterns of their polymorphic assignments. The form of non-polymorphic interaction is introduced before the 3 forms of polymorphic assignments to identify and exclude non-polymorphic interaction from our further discussion.

4.1 Non-Polymorphic Interaction

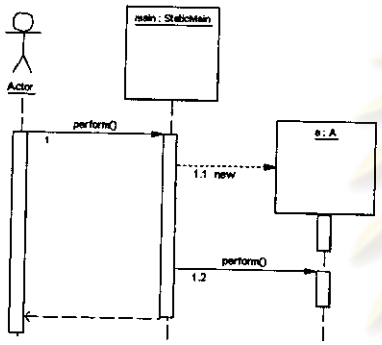


Figure 2. Example of non-polymorphic interaction

Basically non-polymorphic interaction is an interaction which does not have any polymorphic assignment, although a reference in the interaction is defined as a class with one or more subclasses. In other words, it is the situation where the classes of the instances in an interaction are unconditionally fixed to the same set of classes for every execution. Figure 2 shows an example of this form of interaction. Although class A may have subclasses, it is not possible for polymorphism to occur.

4.2 Simple Polymorphic Assignment

The first form of polymorphic assignment is the simplest one. An instance which is assigned to the polymorphic entity is passed as a parameter to the interaction; therefore, polymorphic behavior is directly controlled by the parameter. Figure 3 shows the UML Class Diagram of classes in the class hierarchy used in the examples in this section. An example of an interaction with simple polymorphic assignment is shown in Figure 4.

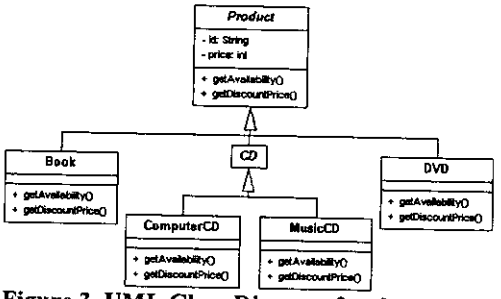


Figure 3. UML Class Diagram for the example

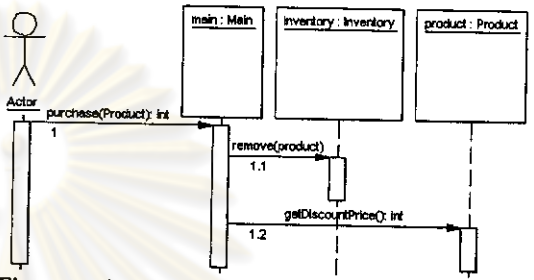


Figure 4. Sequence Diagram of simple polymorphic assignment

Testing an interaction with this form of polymorphic assignment requires each test case to pass in an instance of the class intended to be tested as test input. This form of polymorphic assignment is easy to test, as the parameter to the interaction is directly assigned to the polymorphic entity.

From Figure 3, there are 4 concrete subclasses of abstract class Product (classes with italic names are abstract): Book, ComputerCD, MusicCD, and DVD. In order to test this interaction with all 4 subclasses, 4 test cases must be created; one for each class. The skeletons of test cases for this example are shown in table 1.

Table 1. Test cases for interaction in Figure 4[1]

No.	Test Input
1	An instance of Book
2	An instance of ComputerCD
3	An instance of MusicCD
4	An instance of DVD

4.3 Parameter-Influenced Polymorphic Assignment

Parameter-influenced polymorphic assignment, as its name suggests, is a polymorphic assignment which has its behavior dependent on one or more parameters. The common form of this type of assignment is one of the variations of "factory method" design pattern, known as "parameterized

[1] Note that the table does not show complete test cases. Some information, e.g. exact test data and expected result, is omitted, as it is not relevant to the discussion. This is also true for the subsequent subsections.

factory method" [5]. As a creational pattern, a "parameterized factory method" is responsible for creating instances of various classes from the same inheritance hierarchy. One or more parameters passed to the method are used to determine from which class an instance is created.

An example of an interaction with parameter-influenced polymorphic assignment, still based on inheritance hierarchy in Figure 3, is shown in Figure 5. The polymorphic assignment is in step 1.1, where the reference "product" is assigned to the return result from method "getProduct" on finder object as an instance of class ProductFinder. The method "getProduct" is a parameterized factory method, as it creates an instance of one of concrete subclasses of class Product according to the parameter "id" (it may determine the type of product by looking up in data storage, e.g. database, using the given id.)

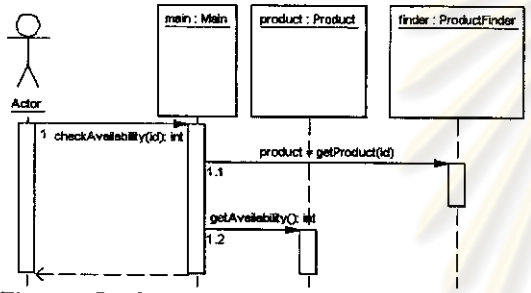


Figure 5. Sequence Diagram of parameter-influenced polymorphic assignment

This form of assignment is similar to simple polymorphic assignment discussed in the previous subsection in that they both are dependent on parameters. Testing this form of assignment, however, is not as straightforward as the other. Each test case must be designed with specific test input which causes the polymorphic assignment to result in the particular class.

Table 2 shows skeletons of test cases for this example. Notice that these test cases are more abstract than the ones from table 1. It is not possible to determine exactly what values of the test inputs should be just from the interaction diagram. Additional information and, sometimes, human decision are required to identify the exact appropriate test input values. As a result, testing this form of polymorphic assignment is less likely to be automated.

Table 2. Test cases for interaction in Figure 5

No.	Test Input (id)
1	Id of a book
2	Id of a computer CD
3	Id of a music CD
4	Id of a DVD

From Figure 5, the parameter "id" is identified as the parameter which affects the polymorphic assignment. It is required that this parameter is also one of parameters of the interaction or is directly dependent on parameters of the interaction, in order to fulfill testability requirement. As this example falls into the first case, the test cases are designed based on the parameter that influences the polymorphic assignment (product id). For the latter case, test case design is more complicated, if not impossible.

4.4 Configuration-Influenced Polymorphic Assignment

The last form is configuration influenced polymorphic assignment. For this type of assignment, one or more types of configuration are the factor that influences polymorphic assignment. Configuration in this context includes all types of set-up and environment, which is external to the interaction. Different from parameter-influenced polymorphic assignment, this form of assignment is not dependent on any parameter passed to the interaction; thus, test cases for this form of assignment are not designed solely around test inputs.

Also based on inheritance hierarchy in Figure 3, Figure 6 shows an example of an interaction with configuration-influenced polymorphic assignment. The polymorphic assignment is in step 1.1, which is the result of the execution of method "getTopSelling" on finder object as an instance of class ProductFinder. Notice that it is different from the interaction in Figure 5 in that the method "getTopSelling" does not take any argument. The method "getTopSelling" returns an instance of the best selling product, which could be one of concrete subclasses of class Product. The behavior of this polymorphic assignment is based on external set-up, not any of parameters of the interaction. Testing this interaction with different subclasses of class Product requires set-up to have different types of product as the best selling product. The skeletons of test cases for this example are shown in table 3.

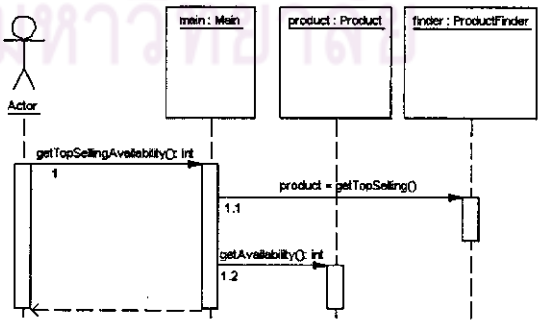


Figure 6. Sequence Diagram of configuration-influenced polymorphic assignment

Table 3. Test cases for interaction in Figure 6

No.	Test Set-up
1.	Set the top selling product to be a book
2.	Set the top selling product to be a computer CD
3.	Set the top selling product to be a music CD
4.	Set the top selling product to be a DVD

The interaction in Figure 6 does not have any argument, and the polymorphic assignment is not dependent on any parameter either. Test cases for this interaction must be designed around configuration, as it is the factor that affects the polymorphic assignment. Testing this form of assignment is more complicated than testing parameter-influenced polymorphic assignment, as test cases with configuration set-up or, best known as, test set-up are difficult to be designed systematically. Moreover, automated test for this form is difficult due to the variety of configuration implementation.

There are a number of possible implementations which falls into configuration-influenced polymorphic assignment. From the example in Figure 6, some content, which is persisted in a file or a database, affects the polymorphic assignment; therefore, changing sales record may result in change of the best selling product. For each test case execution on this interaction, set-up steps must be performed to achieve the required configuration state.

Another example of this form is inversion of control or dependency injection pattern presented by Fowler [10]. There are several Java implementations of this concept, e.g. Spring [11], PicoContainer [12] etc. With dependency injection, associations between objects are not statically fixed in the compiled code, but injected into the code in runtime. Dependency injection could result in polymorphic assignment. Another alternative is "Service Locator" patterns presented by Sun Microsystems [13].

5. Discussion

5.1 Variations

In the previous section, forms of polymorphic assignments are discussed in basic patterns. However, it is not uncommon to see different patterns of polymorphic assignments in the real world. Figure 7 shows an example of parameter-influenced polymorphic assignment, which does not strictly follow the basic pattern. The polymorphic entity is "conn" as a reference to class URLConnection, which has subclasses for specific URL protocols (i.e. HttpURLConnection for HTTP connection and FtpURLConnection for FTP connection etc.). However, the polymorphic assignment in step 1.2 does not take any parameter. The parameter that affects the polymorphic assignment is "urlString", supplied in step 1.1 when an instance of URL is created. Then test case design for this interaction must focus on the parameter "urlString".

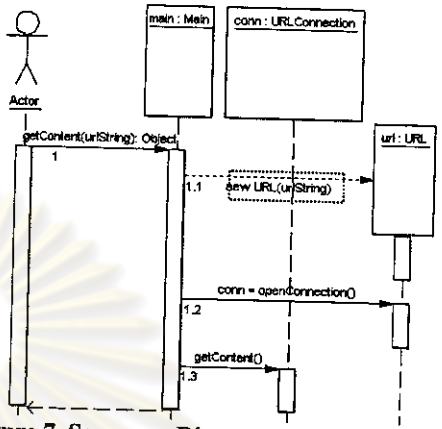


Figure 7. Sequence Diagram of URL example

This example shows that, although does not follow the pattern exactly, the concept of testing can also be applied. The problem is how to identify the form of these interactions. Stereotypes in the design model may be required in order to automate test case creation.

5.2 Combination

Another possible situation is when there is more than one polymorphic entity in an interaction. In addition, it is possible that different forms of polymorphic assignments happen in the same interaction. The concept in this paper is still applicable in this case, although it does not deal with the issue of combination. In our previous work [14], we present adequacy criteria for testing polymorphic interaction which address the situation where more than one polymorphic entity is involved.

6. Conclusion

In this paper, 3 forms of polymorphic assignments are presented along with their test case design approach. Given a polymorphic interaction with an inheritance hierarchy, it is possible to apply the relevant approach to design test cases for the interaction. Although test data generation is not considered in this paper, automated test case generation is possible with some help from additional information in the design model.

The finding in this paper complements our previous works [3,14]; a tool can be implemented to aid testing of OO software based on its sequence

diagrams. From UML Sequence Diagrams with additional information for inheritance hierarchy from, for example, UML Class Diagram, adequacy criteria presented in [14] can be applied to decide which classes (or, strictly speaking, subclasses) are to be included under test and how many test cases are required to cover these classes. Combined with conventional control flow testing, the concept in this paper can be applied to identify test input and/or test set-up for each test case, and finally an instrumentation model in [3] is applied to check whether test execution results in the same message sending sequence, including the order of messages and classes of instances under test, as designed in the design model and the test case.

7. Acknowledgement

This research is supported by TJTTP-OECF (Thailand-Japan Technology Transfer Project - Japanese Overseas Economic Cooperation Fund) and Department of Computer Engineering – Industry Linkage Research Project, year 2004, Chulalongkorn University. The authors would like to thank Prof. Koichiro Ochimizu of JAIST (Japan Advance Institute of Science and Technology) for his useful comments and suggestions.

8. References

[1] The Object Management Group, "OMG Unified Modeling Language Version 1.4", http://www.omg.org, September 2001.

[2] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[3] S. Supavita and T. Suwannasart, "An Instrumentation Model for Supporting Software Testing Based on UML Sequence Diagrams", *Proceedings of the 4th Information and Computer Engineering Postgraduate Workshop (ICEP 2004)*, Phuket, Thailand, January 2004.

[4] B. Meyer, *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, 1st Edition, Addison-Wesley, 1995.

[6] D. E. Perry and G. E. Kaiser, "Object-Oriented Programs and Testing", *Journal of Object Oriented Programming*, January/February 1990.

[7] M. D. Smith, D. J. Robson, "Object-Oriented Programming - the Problems of Validation", *Proceedings of Conference on Software Maintenance*, San Diego, CA, USA, 26-29 November 1990, pp. 272-281.

[8] S. Barbey and A. Strohmeier, "The Problematics of Testing Object-Oriented Software", *Proceedings of SQM '94 Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, Volume 2, 1994, pp. 411-426.

[9] R. V. Binder, "Design for Testability in Object-Oriented Systems", *Communications of the ACM*, Volume 37, Issue 9, September 1994.

[10] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern", http://www.martinfowler.com/articles/injection.html, January 2004.

[11] "Spring – Java/J2EE Application Framework", http://www.springframework.org/, November 2004.

[12] "Pico Container", http://www.picocontainer.org/, November 2004.

[13] Sun Microsystems, "Core J2EE Patterns - Service Locator", http://java.sun.com/blueprints/corej2eepatterns /Patterns/ServiceLocator.html, November 2004.

[14] S. Supavita and T. Suwannasart, "Adequacy Criteria for Testing Polymorphism in the Context of Interactions", *Proceedings of The International Conference on Sofware Engineering Research and Practice (SERP'04)*, July 2004.

ภาคผนวก ง

ผลงานวิจัย

บทความเรื่อง

An Approach for Generating Test Cases from Use Cases

นำเสนอใน

National Conference on Computing and Information Technology

(NCCIT05),

ระหว่างวันที่ 24-25 พฤษภาคม 2548

สถาบันเทคโนโลยีพระจอมเกล้าพระนครเหนือ

An Approach for Generating Test Cases from Use Cases

S. Leeraharattanarak and T. Suwannasart
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University
Bangkok, 10330, Thailand
E-mail: Lsetapong@yahoo.com, Taratip.S@chula.ac.th

Abstract

Testing is a very important activity in software development process. During testing, testers have to generate test cases, which consist of test input values and expected results, before test execution. In functional testing, test designers can generate test cases from a requirements document. The test cases generating process takes a lot of time in large systems. In this paper, we propose an approach for generating test cases from use cases, which represent software requirements. These test cases can launch the test process early in the software development life cycle.

Keywords: Software Testing, Use cases, Test Cases

1. Introduction

Testing is an important activity in software development process, since it is expensive and takes a lot of time. If testing can start at an early stage such as requirement analysis, money spent in developing a software product would be less.

To start testing process early in software development process, we should focus on how to design test based on functions of a software. To date, UML [1] has emerged and widely used in many software development organizations. Use case diagram, which is a UML 's diagram, shows relationship among use cases within a system or other semantic entity and their actors. Use case diagrams and documents are written during analysis phase to identify and partition system functionalities. Our research is focused on how to derive specification-based test cases from use cases. In this paper, we propose an approach that generates test cases from use cases.

The organization of this paper is as follows. In section 2 of this paper, we discuss related works. Section 3 describes the UML use case diagram and document. Use cases are explained in section 4. Our approach is illustrated with a case study in section 5. In section 6, we discuss the future works. Finally, we conclude the paper in section 7.

2. Related Works

There are numbers of researchers have been focused on test case generation based on UML diagrams. However, few researches propose the techniques that automatically generate test cases from a use case diagram or a requirement document. Jim Heumann [2] proposed how test cases are generated from use cases by extracting use cases' description in order to get their scenarios, and then, building the test case matrix that includes test case id, inputs and expected output. With his approach, test designers must analyze the use case description and create the scenarios for each use case.

Imp Software, Inc. [3] developed the tool, named "SpecStudio", that can build and analyze behavior specification. This tool can generate test cases from the behavior specification. System analyst can generate test cases by choosing actions from the behavior specification. The generated test cases have no test data therefore they cannot be applied for testing immediately. Thus, testers have to identify test data to each test cases.

3. Use case diagram

Use case diagram [1] is an UML diagram which shows relationship between actors and use cases. Use cases represent function of a system, or a subsystem. Actors represent users or systems that connect to use cases. A use case diagram is a graph of actors, a set of use cases, possibly some interfaces, and the relationships between these elements. An example use case diagram is shown in figure 1.

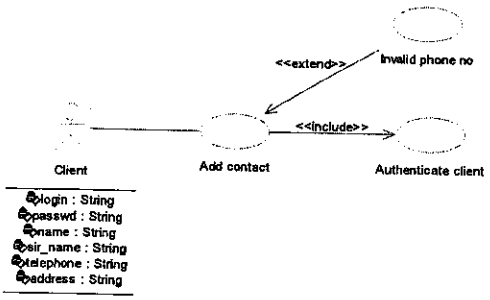


Figure 1. A use Case Diagram

A use case is shown as an ellipse containing the name of the use case. The behavior of a use case can be described in plain text. An actor is shown as a stick figure with the actor name.

There are several standard relationships among use cases or between actors and use cases.

- *Extend* – An extend relationship is happened when use case A cannot apply the actions as a normal behavior and call use case B for handling the failure or exception.
- *Include* – An include relationship is happened when an use case A call use case B. Use case A includes the behavior of use case B. It's similar to main program call sub program.
- *Association* – An association relationship is a relationship between actors and use cases that is happened when the actors use the use cases.

4. Use Cases

Use cases are a powerful tool to capture functional requirements. They provide a mean to specify the interaction between a certain software system and its environment and allow for structuring requirements according to user goals. An effective and widely used technique for specifying use cases was presented by Cockburn in [4]. In this research, we choose some parts of Cockburn's use cases (including use case name, actor, pre-condition, success scenario, alternative scenario and post-condition) to deal with software requirements specification for generating test cases. Use case description format, which is used in this approach, is depicted in table 1 and described as follows.

Table 1. Example of use case: Add contact

Use case no.:		1		
Use case name:		Add contact		
Description:		Adding new contact to list		
Actor:		Client		
Pre-condition:		Client enters the contact information: name, surname, telephone number and address.		
Required-item:		Name	Type	Size
		name	String	15
		surname	String	20
		telephone	String	7
		address	String	20
Is abstract:		1		
Success scenario:				
Condition no:	0	(name.length > 0) && (surname.length > 0) && (telephone>="0000000") && (telephone<="9999999")		
Step		Action		
1		{UC2}		
2		System submits the contact information from client.		
3		System saves the contact information into database.		
4		System shows a message "Save new contact complete".		
Alternative scenario:				
Condition no:	2.1	(name.length <= 0)		
Step		Action		

2.1.1		System shows an error message "Please enter Name".
Condition no:	2.2	(sir_name.length <= 0)
Step		Action
2.2.1		System shows an error message "Please enter surname".
Condition no:	2.3	-
Step		Action
2.3.1		{UC3}
Post-condition		0 System saves new contact and shows a message "Save new contact complete". 2.1 System shows an error message "Please enter name" 2.2 System shows an error message "Please enter surname"

- *Use case no* – a number as unique identifier of a use case which starts at 1.
- *Use case name* – the name is the use case goal as a short active verb phase.
- *Description* – a detailed explanation of a use case.
- *Actor* – a name of user or system that calls on the system to deliver one of its services.
- *Pre-condition* – a condition that the system has to satisfy before starting the use case.
- *Required items* – Inputs, which actor has to give the system, including item names, item types, and item sizes.
- *Is abstract* – the property of use case which shows that the use case is the abstract use case. If "Is abstract" is represented by "0", it means that the use case is an abstract use case, otherwise it is not.
- *Success scenario* – a story line that shows the system deliver the goal. A success scenario consists of a condition as an expression, and steps of actions. When the condition is true, the system will apply the steps of actions as a normal behavior. The number of condition in success scenario is always set to "0". The step number of actions starts at 1.
- *Alternative scenario* – a story line that shows the conditions and actions which handle system failure or exception. An alternative scenario includes numbers of conditions, the conditions corresponding to a condition numbers, and steps of actions that are the results of the conditions. The condition number is represented by the step number of action, which happens as an exception, in success scenario followed by "." and a running number. The step number of action in an alternative scenario is represented by the condition number in the alternative scenario followed by "." and a running number. As in table 1, in case that client enters "name" as blank or represented by "name.length == 0", it is not applied the steps of actions number 2 of the success scenario. Therefore, the use case has to follow alternative scenario. In this case if condition number 2.1 is true, its action must be followed.
- *Post-condition* – the state to be happened after the use case completely executes. The state will be classified by number of conditions. As in table 1, post-condition has 3 states.

An use case has to have one success scenario and have possible none or more alternative scenarios. The example in table 1, the use case "Add contact" has a success scenario and 3 alternative scenarios.

5. Our Approach

In this paper, we propose an approach that generates test cases from use cases. The conceptual framework is shown in figure 2.

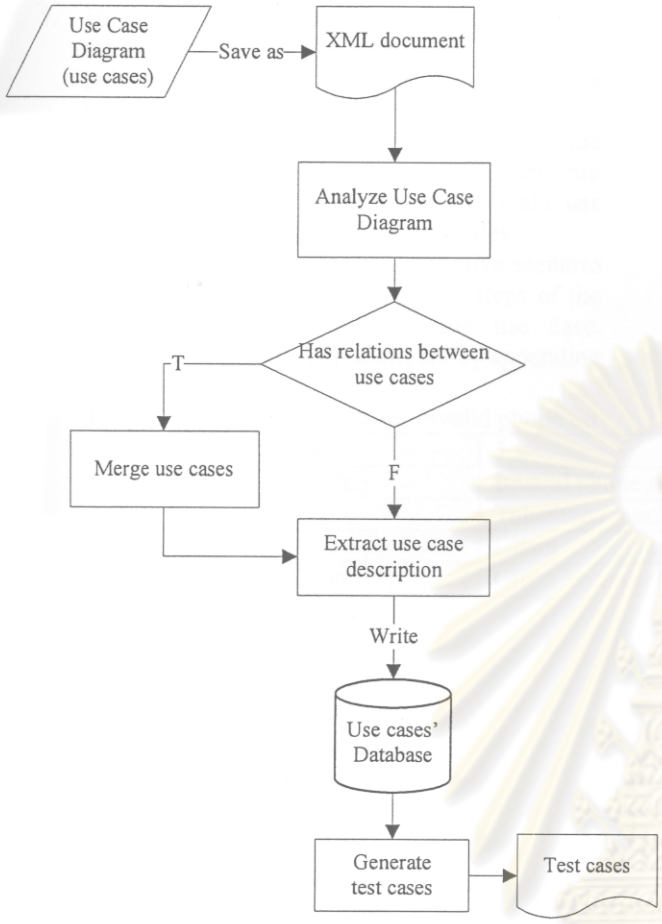


Figure 2. The test case generator process

The test generation process starts with the preparation of use case documents. Use case documents must be written in the format described in section 4. Beside general use case information, the format includes some additional information that could aid the test case generation process. In the previous section, the example of a use case document shows condition numbers and step numbers for basic flow and alternative flows. There is another additional information: linked points. When there is a relationship between use cases, either include or extend, a linked point is specified in a step of action where the use cases connected. If a step in use case number 1 calls use case number 2 as include relationship, a linked point "{UC2}" is represented in the step. As shown in the example in table 1, the first step in the success scenario of the use case "Add contact" is a linked point to the use case number 2, "Authenticate client". When a relationship between use cases is an extend relationship, a linked point is specified in a step of action in the alternative scenario of the use case. As shown in the example in the figure 1 , the use case "Add contact" is extended by the use case "Invalid phone no" and the example in table 1, the step number 2.3.1 of the alternative scenario calls use case number 3, "Invalid phone no".

Use case diagrams and use case documents are physically stored in XML format. When there are relationships between use cases, the use case documents of the related use cases must be merged into one use case document representing those use cases. As shown in Figure 1,the use case "Add contact" includes the use case "Authenticate client" and is extended by the use case "Invalid phone no", therefore, the use case "Add contact" (shown in table 1) has to merge use case "Authenticate client" (shown in table 2) and use case "Invalid phone no" (shown in table 3) to become the new use case "Add contact". Table 4 shows a logical view of merged use case document representing use case "Add contact", "Authenticate client", and "Invalid phone no", which are shown in table 1, 2, and 3 respectively.

We introduce a merging use cases procedure as follows:

1. Define the caller use case as the *main use case* and the callee use case as the *sub use case*.
2. Combine the pre-condition of the sub use case into pre-condition of the main use case.
3. Add the new required items from the sub use case to the required items of the main use case.
4. If the use case relationship is an include relationship, or, in the other word, the linked point appears in a step in the success scenario. The merge procedure proceed as follows:
   - 4.1 Combine the conditional expression of the success scenario of the sub use case to the conditional expression of the success scenario of the main use case with an AND operator ("&&").
   - 4.2 Replace the action of the success scenario of the main use case at the linked point with the actions of the success scenario of the sub use case. Adjust step number of the new actions. The new step numbers begin with the previous old step number followed by a "-" and running number (started from 1). For instance, in table 1, the first step in the success scenario is replaced with the actions in success scenario from use case number 2, and the step number of the actions are labeled as 1-1, 1-2, 1-3, and so on.
   - 4.3 Add the steps from the alternative scenario of the sub use case to the alternative scenario of the main use case and adjust the condition numbers and the step numbers of the new steps as described in the previous merging step. An example of merged include relationship use case is shown in table 4.

5. If the use case relationship is an extend relationship, or, in the other word, the linked point appears in the alternative scenario, the merge procedure proceeds as follows:
   5.1 Insert the conditional expression of the success scenario of the sub use case into the alternative scenario of the main use case where the linked point locates.
   5.2 Replace the step in the alternative scenario of the main use case with the steps of the success scenario of the sub use case. Adjust the new step numbers by appending the previous step number with a dot and a running number. For example, the merged use case in table 4 shows the new step number as 2.3.1.
   5.3 Add the post-condition of the alternative scenario of the sub use case to the post-condition of the main use case.

After use case documents are merged, we have to extract all information (including use case number, use case name, pre-condition, required-item, success scenario, alternative scenario, and post-condition) of each use case and insert them into the database.

Table 2. Example of use case: Invalid phone no

Use case no.:		3		
Use case name:		Invalid phone no		
Description:		Telephone number is invalid		
Actor:		Client		
Pre-condition:		Client enters telephone number.		
Required-item:		Name	Type	Size
		Telephone	String	7
Is abstract		1		
Success scenario:				
Condition no:	0	(telephone < "0000000") || (telephone > "9999999")		
Step		Action		
1		System shows error message "telephone number length must be 7"		
Post-condition:		0 System shows error message "telephone number length must be 7"		

Table 3. Example of use case: Authenticate client

Use case no.:		2		
Use case name:		Authenticate client		
Description:		Authenticate client before using the system.		
Actor:		Client		
Pre-condition:		I Login name and password are created in system. Client enters login name and password.		
Required-item:		Name	Type	Size
		login passwd	String String	8 5
Is abstract		1		
Success scenario:				
Condition no:	0	(login <> "") && (passwd <> "")		
Step		Action		
1 2 3		System submits login name and password form client. System checks login name and password. Client logs in to the system.		
Alternative scenario:				
Condition no:	2.1	(login == "")		
Step		Action		
2.1.1		System shows an error message "Please enter login".		
Condition no:	2.2	(passwd == "")		

Step	Action
2.2.1	System shows an error message “Please enter password”.
Post-condition:	0 Client logs into the system. 2.1 System shows an error message “Please enter login”. 2.2 System shows an error message “Please enter password”.

Table 4. Example of merged use case: Add contact

Use case no.:		1		
Use case name:		Add contact		
Description:		Adding new contact to list		
Actor:		Client		
Pre-condition:		Client enters the contact information: name, surname, telephone number and address. Login name and password are created in system. Client enters login name and password.		
Required-item:		Name	Type	Size
		login	String	8
		passwd	String	5
		name	String	15
		surname	String	20
		telephone	String	7
		address	String	20
Is abstract		1		
Success scenario:				
Condition no:	0	(name.length>0) && (surname.length>0) && (telephone>=”0000000”) && (telephone<=”9999999”) && (login <> “”) && (passwd <> “”)		
Step		Action		
1-1		System submits login name and password from client.		
1-2		System checks login name and password.		
1-3		Client logs in to the system.		
2		System submits the contact information from client.		
3		System saves the contact information into database.		
4		System shows a message “Save new contact complete”.		
Alternative scenario:				
Condition no:	1-2.1	(login == “”)		
Step		Action		
1-2.1.1		System shows an error message “Please enter login”.		
Condition no:	1-2.2	(passwd == “”)		
Step		Action		
1-2.2.1		System shows an error message “Please enter password”.		
Condition no:	2.1	(name.length<=0)		
Step		Action		
2.1.1		System shows an error message “Please enter name”.		
Condition no:	2.2	(sir_name.length<=0)		
Step		Action		
2.2.1		System shows an error message “Please enter surname”.		

Condition no:	2.3	(telephone < "0000000") || (telephone > "9999999")
Step		Action
2.3.1		System show error message "telephone number length must be 7"
Post-condition:		0 Client log into the system. System save new contact and show message "Save new contact complete" 1-2.1 System shows an error message "Please enter login". 1-2.2 System shows an error message "Please enter password". 2.1 System shows an error message "Please enter name". 2.2 System shows an error message "Please enter surname". 2.3 System shows error message "telephone number length must be 7".

The format of our generated test cases consists of parts as follows:

- *Test case id* – the unique running number of test case.
- *Test case name* – the name of test case which is from the name of the source use case.
- *Description* – the detail shows whether the test case is generated form the alternative or the success scenario along with the conditional expression of the scenario.
- *Pre-condition* – the pre-condition of the source use case.
- *Input* – the test data of the test case which is generated randomly corresponding to the data types of required items and the conditional expression of scenario.
- *Expected output* – the result that will be happened after testers use these test data to test the system. The expected output is displayed as a sequence of actions.
- *Post-condition* – the post-condition of the scenario of the source use case.

We use information of use cases, which are extracted and stored in the database, to generate test cases. One conditional expression of use cases representing a single scenario can be generated into a test case; thus the number of generated test cases for a use case is equal to the number of all conditional expressions of the use case. We propose steps to generate test cases as follows:

1. Define test case id as the use case id and followed by a dot and a running number started at 1.
2. Assign the use case name to the test case name.
3. If the test case is generated for a success scenario, write description of test case as "Success scenario"; otherwise write it as "Alternative scenario". And follow by conditional expression of scenario of use case.
4. Assign the pre-condition of the use case to the pre-condition of the test case.
5. Randomly generate test data considering the required item types, sizes and the conditional expression of the scenario of the use case.
6. Set the expected output by using the order of actions which happen according to the conditional expression.
7. Assign the post-condition of the use case to the post-condition of test case.

An example of the test case, which is generated from use case "Add contact", is shown in table 5. Since use case "Add contact" has 6 conditional expressions, the number of the generated test cases for the use case is 6.

Table 5. Example of test case that is generated from use case: Add contact

Test case id	1.2
Test case name	Add contact
Description	Alternative Scenario: (login == "")
Pre-condition	Client enters the contact information: name, surname, telephone number and address. Login name and password are created in system. Client enters login name and password.

Input	Name	Value
	login	<blank>
	passwd	vr
	name	c2cs
	surname	ru845uw8n1ku
	telephone	636053
	address	zc5rks
Expected Output	1. System submits login name and password from client. 2. System checks login name and password. 3. System shows an error message "Please enter login".	
Post-condition	System shows an error message "Please enter login".	

6. Future Works

We have developed a tool to support automation of the proposed approach. The tool can generate test cases from use cases by extracting use case description from use case diagrams that are drawn with Rational Rose and exported to an XML file. This tool supports four data types: integer, float, boolean and string. Four arithmetic operators can appear in conditional expression: +, -, * and /. The outputs of this tool are test cases in HTML documents. All test data of test cases is generated randomly according to data types of input; therefore, some test data is maybe not realistic. So, one of our future research directions is to overcome this limitation.

7. Conclusion

This paper presents an approach to generate test cases from use cases. We define the use case format that is suitable for test case generation. We discuss the process of generating test case step by step including use case merging procedure and test generation procedure. The generated test cases cover all possible scenarios of the source use cases. Based on this approach, we have developed a tool that can automatically generate test cases from use cases. These test cases can be applied for system testing in software development process.

8. Acknowledgement

This research is support by TJTTP-OECF (Thai-Japan Technology Transfer Project – Japanese Overseas Economic Cooperation Fund) and Department of Computer Engineering – Industry Linkage Research Project, Year 2004, Chulalongkorn University.

9. References

[1] Object Management Group, "OMG Unified Modeling Language Specification", March 2003, Available from: http://www.omg.org

[2] Heumann, J., "Generating Test Cases From Use Cases", June 2001, Available from: http://www.therationaledge.com/content/jun_01/m_cases_jh.html

[3] Impl Software, Inc., "Object-Oriented Behavior Modeling and Software Testing", May 2003, Available from: http://www.specstudio.com

[4] Cockburn, A., "Writing Effective Use cases", United States of America: Addison-Wesley, 2000.