폭포수형에서 반복적 개발로 — 프로젝트 관리자의 전이 과제

Philippe Kruchten

Rational Software 백서

TP 173, 5/00

Rational®
the software development company

목차

소개

Rational Unified Process(RUP)는 소프트웨어 개발 라이프사이클에 대한 반복 또는 나선형 접근 방식을 지원합니다. 이 접근 방식은 많은 측면에서 폭포수형 접근 방식에 비해 우수한 방식임이 여러 차례 입증되었습니다. 그러나 반복 라이프사이클이 제공하는 많은 이점에는 그에 따른 대가가 있음에 또한 유의해야 합니다. 반복적 개발은 소프트웨어 개발에 있어 가능한 모든 문제점 또는 어려움을 해결할 수 있는 마법의 지팡이가 아닙니다. 프로젝트가 반복적인 특성을 갖고 있다고 해서 설정, 계획 또는 제어하기가 더 쉬운 것은 아닙니다. 프로젝트 관리자는 특히 첫 번째 반복 프로젝트에서 실제로 어려운 타스크에 직면하게 됩니다. 또한 위험성이 높고 조기 실패 가능성이 있는 초기 프로젝트 반복에서도 많은 어려움에 처하게 됩니다. 이 문서에서는 프로젝트 관리자의 관점에서 반복적 개발의 도전 과제에 대해 설명합니다. 또한 Rational 동료의 보고서 또는 중요 사례나 Rational 의 컨설팅 경험을 바탕으로, 프로젝트 관리자가 일반적으로 쉽게 빠질 수 있는 몇 가지 "함정"에 대해 설명합니다.

반복적 개발

기존의 소프트웨어 개발 프로세스는 다음 그림과 같은 폭포수형 라이프사이클을 따릅니다. 이 접근 방식(그림 1 참조)에서는 개발 프로세스가 요구사항 분석에서 디자인, 코드 및 유닛 테스트, 서브시스템 테스트, 시스템 테스트로 진행되며 각 단계별로 이전 단계의 결과에 대한 제한적인 피드백이 제공됩니다.

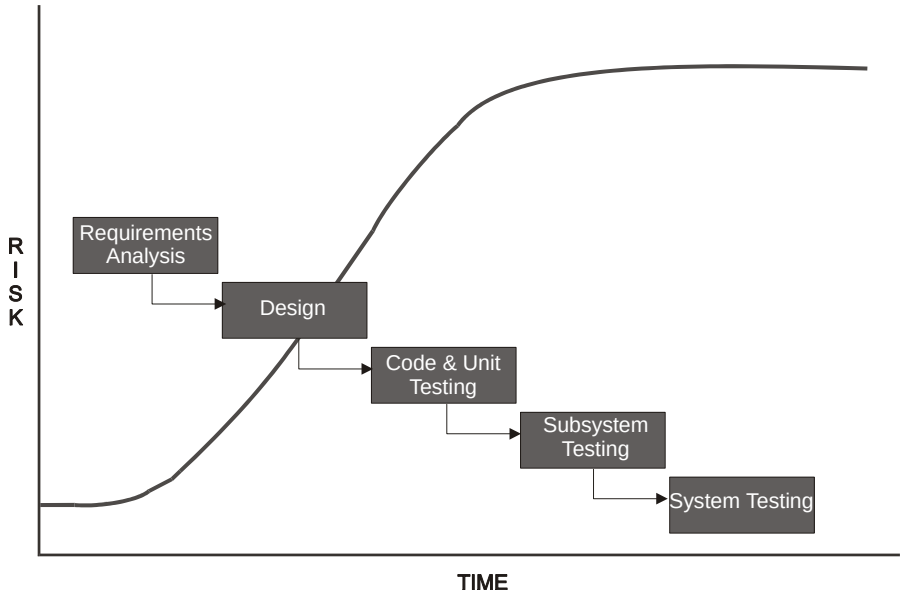

그림 1: 폭포수형 개발 프로세스

이 접근 방식의 가장 큰 문제점은 시간이 지남에 따라 위험성이 누적되며 이로 인해 이전 단계에서의 실수를 회복하는 데 많은 비용이 소요된다는 것입니다. 초기 디자인은 핵심 요구사항과 관련하여 일반적으로 결점이 있으며 이러한 디자인 결함을 나중에 발견함으로써 많은 비용이 소요되거나 프로젝트가 취소될 수도 있습니다. 폭포수형 접근 방식의 경우 프로젝트의 실제 위험성이 외부로 나타나지 않아 필요한 조치를 취할 수 있는 시점을 놓칠 수 있습니다.

이러한 폭포수형 접근 방식의 대안은 반복 및 점진적 프로세스입니다(그림 2 참조).

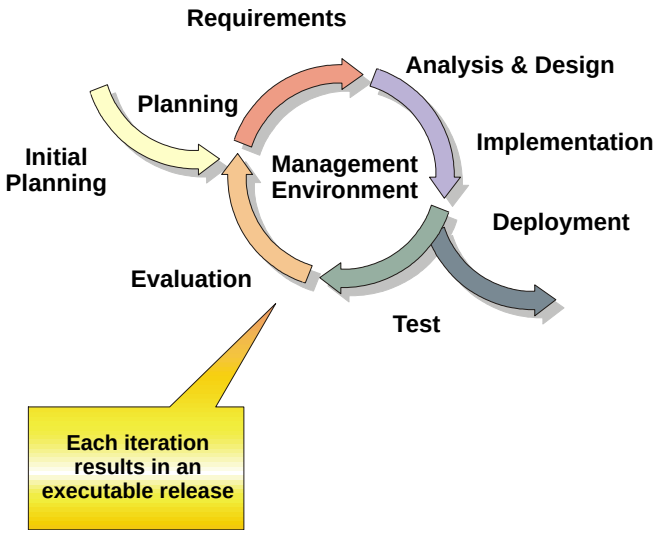

그림 2: 반복적 개발 접근 방식

이 접근 방식은 Barry Boehm 이 개발한 나선형 모델("추가 참조" 참고)을 기반으로 하며, 프로젝트의 위험성을 라이프사이클 초기에 식별함으로써 필요한 시점에 효율적으로 위험성을 처리하고 대응할 수 있습니다. 이 접근 방식에서는 발견, 고안 및 구현이 지속적으로 수행되며 각 반복마다 개발 팀에서 예측 가능하고 반복 가능한 방식으로 프로젝트 아티팩트를 완성해야 합니다.

장점: 반복적 개발의 이점

전통적인 폭포수형 프로세스와 비교할 때 반복 프로세스는 다음과 같은 많은 이점을 갖고 있습니다.

1. 라이프사이클 초기에 심각한 문제가 밝혀짐으로써 대응할 수 있는 시간적 여력이 있습니다.
2. 적극적인 사용자 피드백을 통해 시스템의 실제 요구사항을 도출할 수 있습니다.
3. 개발 팀은 프로젝트에 있어 가장 중요한 문제에 역량을 집중하고, 팀 구성원은 다른 문제에 방해를 받지 않고 프로젝트의 실제 위험성에 전념할 수 있습니다.
4. 지속적이고 반복적인 테스트를 통해 프로젝트 상태를 객관적으로 평가할 수 있습니다.
5. 요구사항, 디자인 및 구현 간의 불일치를 조기에 발견할 수 있습니다.
6. 팀, 특히 테스트 팀의 워크로드가 전체 라이프사이클에 균등하게 배분됩니다.
7. 이 접근 방식을 사용하는 경우 팀에서 얻은 교훈을 활용함으로써 프로세스를 지속적으로 개선할 수 있습니다.
8. 프로젝트의 이해 당사자(stakeholder)가 라이프사이클 전체에서 프로젝트 상태를 구체적으로 확인할 수 있습니다.

위험성 완화

반복 프로세스에서는 통합을 통해서만 위험성을 발견하거나 처리할 수 있어 위험성을 조기에 완화시킬 수 있습니다. 초기 반복을 전개할 때 도구, 기존 소프트웨어 제품 및 인력 스킬과 같은 다양한 프로젝트 측면을 연습함으로써 모든 프로세스 컴포넌트를 검토할 수 있습니다. 인지된 위험성이 위험하지 않은 것으로 판명되거나 예상치 못한 새로운 위험성을 발견할 수 있습니다.

특정 이유로 프로젝트를 중단해야 하는 경우, 많은 시간, 노력 및 비용이 소요되기 전에 가능한 빨리 중단하는 것이 좋습니다. 지나치게 신중한 태도보다는 위험성에 맞서는 것이 올바른 방법입니다. 제품을 잘못 빌드하는 등 기타 위험성 중에서도 반복적 개발 프로세스를 통해 조기에 완화시킬 수 있는 위험성은 다음과 같은 두 가지 카테고리로 요약됩니다.

- 통합 위험성
- 아키텍처 위험성

반복 프로세스에서는 여러 반복을 통해 오류를 정정하므로 보다 견고한 아키텍처가 생성됩니다. 제품이 도입/인식(Inception) 단계를 지나는 초기 반복에서 결함이 발견됩니다. 성능 병목 현상이 발생하는 경우 제품을 전달하기 직전이 아니라 문제를 해결할 수 있는 시점에서 발견할 수 있습니다.

통합이 라이프사이클 종료 시점에서 발생하는 "큰 사건"이 아니며 점진적인 요소 통합이 수행됩니다. 실제로, 권장되는 반복 접근 방식에는 거의 지속적인 통합이 포함됩니다. 장시간 불확실하고 어려웠던 문제(프로젝트 종료 시점에는 총 노력의 40% 소요)가 6 – 9 개의 소규모 통합으로 분할됩니다. 이러한 통합은 훨씬 적은 요소로도 시작할 수 있습니다.

변경사항 수용

다음과 같은 여러 변경사항 카테고리를 예견할 수 있습니다.

- 요구사항 변경사항

  반복 프로세스를 통해 변경되는 요구사항을 고려할 수 있습니다. 실제로 요구사항은 일반적으로 변경됩니다. 요구사항 변경 및 "요구사항 지체"는 항상 프로젝트 문제의 주요 원인으로서, 전달 지연, 스케줄 지연, 고객 불만 및 개발자를 어렵게 하는 결과를 가져왔습니다. 그러나 사용자 또는 사용자 대표에게 제품의 초기 버전을 공개함으로써 타스크에 대한 제품의 적합성 여부를 확인할 수 있습니다.

- 전술적 변경사항

  반복 프로세스는 예를 들어, 기존 제품과의 경쟁을 위해 경영진에게 제품을 전술적으로 변경할 수 있는 방법을 제공합니다. 예를 들어, 제한된 기능의 제품을 조기에 릴리스함으로써 경쟁자를 먼저 공격하거나 특정 기술에 대해 다른 벤더를 채택할 수 있습니다. 또한 반복 컨텐츠를 재구성하여 공급자가 수정해야 하는 통합 문제점을 완화시킬 수 있습니다.

- 기술적 변경사항

  반복 접근 방식에서는 적게나마 기술 변경사항을 수용할 수 있습니다. 이 기능은 정제(Elaboration) 단계에서 사용할 수 있으며, 구현/구축(Construction) 및 전이(Transition) 단계에서는 본질적으로 위험하므로 이러한 유형의 변경을 피해야 합니다.

지속적인 학습

반복 프로세스의 이점 중 하나는 개발자가 지속적인 학습을 통해 전체 라이프사이클에서 다양한 역량과 전문성을 효과적으로 활용할 수 있다는 것입니다. 예를 들어, 테스터는 테스트를 조기에 시작하고 전문 기술 작성자 역시 일찍 작업을 시작할 수 있는 반면 비반복적 개발에서는 테스터 및 전문 기술 작성자 모두 계획별 진행에 따라 작업을 시작해야 합니다. 훈련 요구 또는 추가 지원(일반적으로 외부 지원)에 대한 요구 또한 평가 검토 시 조기에 발견됩니다.

프로세스 자체 또한 지속적으로 개선되고 정제될 수 있습니다. 반복 종료 시 수행되는 평가를 통해 제품/스케줄 관점에서 프로젝트 상태를 검토하고 조직 및 프로세스에서 변경되어야 할 사항을 분석함으로써 다음 반복에서 보다 나은 기능을 수행할 수 있도록 합니다.

재사용 기회의 증가

반복 프로세스에서는 처음부터 모든 공통성을 식별하는 대신 부분적으로 디자인 또는 구현된 공통 파트를 쉽게 식별할 수 있어 프로젝트 요소를 쉽게 재사용할 수 있습니다. 재사용가능한 부분을 식별하고 개발하는 것은 어렵습니다. 설계자가 초기 반복에서 디자인을 검토함으로써 예상치 못한 잠재적 재사용 기회를 식별할 수 있으며 후속 반복에서 공통 코드를 개발하고 완성시킬 수 있습니다. 공통 문제점에 대한 공통 솔루션이 발견되고 시스템 전체에 적용되는 아키텍처 메커니즘 및 패턴이 식별되는 것은 정제(Elaboration) 단계의 반복에서입니다.

전반적인 품질 향상

반복 프로세스의 결과 제품은 기존의 순차적 프로세스의 결과 제품보다 전반적으로 품질이 우수합니다. 시스템 테스트를 여러 번 수행함으로써 테스트 품질도 향상됩니다. 요구사항이 정제되어 사용자의 실제 요구와 보다 밀접한 관련을 갖게 됩니다. 또한 전달 시점에서 시스템 수명 또한 늘어납니다.

단점: 예상치 못한 취약점 및 일반적인 함정

반복적 개발이 반드시 적은 작업량과 스케줄 단축을 의미하는 것은 아닙니다. 가장 큰 장점은 결과물과 스케줄을 보다 정확하게 예측할 수 있다는 것입니다. 이러한 경우 요구사항은 물론 디자인 및 구현을 개선할 수 있는 시간을 확보할 수 있어 고품질의 제품을 만들 수 있으며 결과적으로 일반 사용자의 실제 요구를 만족시킬 수 있습니다.

반복적 개발에는 실제로 많은 계획이 포함됩니다. 이러한 경우 전체 계획을 개발하는 것 이외에도 각 반복마다 세부 계획을 개발해야 하므로 프로젝트 관리자에게 많은 부담을 줄 수 있습니다. 또한 문제점, 솔루션 및 계획을 서로 절충하기 위해 지속적인 협상이 필요하며 초기에 보다 구체적인 아키텍처 계획이 수립됩니다. 각 개정마다 반복적으로 아티팩트(계획, 문서, 모델 및 코드)를 수정, 검토 및 승인해야 합니다. 경우에 따라 전술적 변경사항 또는 범위 변경사항으로 인해 계획을 지속적으로 수정해야 하며 이러한 경우 각 반복에서 팀 구조를 약간 수정해야 합니다.

함정: 지나치게 완벽하고 세부적인 계획 수립

일반적으로 스케줄 및 자원의 전체적인 범위를 평가하기 위한 연습의 경우를 제외하고 세부 계획을 완전히 구성하는 것은 비경제적입니다. 이 계획은 첫 번째 반복이 종료되기 전에 이미 쓸모 없는 계획이 되어버리기 때문입니다. 올바른 아키텍처를 구성하고 요구사항을 정확히 파악하기 전에는(LCA 이정표에서 수행) 현실적인 계획을 수립할 수 없습니다.

따라서 계획 대상 활동, 아티팩트 또는 반복에 대해 알고 있는 지식 한도내에서 정확하게 계획을 수립해야 합니다. 즉, 단기 계획은 자세하고 세부적으로 수립합니다. 반면에 장기 계획은 개략적인 형태를 유지합니다.

관련 지식이 없거나 잘못된 정보를 갖고 있는 관리자가 "종합적인 전체 계획"을 작성해서는 안됩니다. 관리자를 교육시키고 반복 계획의 개념을 설명하며 먼 미래의 세부사항을 미리 예측하려는 노력의 비효율성을 이해시킵니다. 예를 들어, 뉴욕에서 LA 까지 자동차 여행을 하는 경우 전반적인 루트에 대한 계획은 수립하되 도시를 떠나 실제 여행을 시작할 때는 세부 운전 지침만 필요합니다. 캘리포니아 도착은 말할 것도 없이 캔사스 지방에서의 운전을 위한 정확한 세부사항은 아직 필요하지 않습니다. 실제로 캔사스 지방을 지날 때 도로 보수 공사를 하는 경우 그 때 대체 루트를 찾으면 됩니다.

빠른 재작업 필요성 인식

폭포수형 접근 방식에서는 최종 테스트 및 통합 과정에서 예상치 못한 까다로운 버그가 발견되어 프로젝트 종료 시점에 많은 재작업을 수행해야 합니다. 더구나 대부분의 "문제" 원인은 디자인 오류에서 비롯된 것입니다. 이러한 경우 일시적인 해결 방법을 통해 구현 시 완화시키려고 하면 결과적으로 보다 심각한 문제로 발전합니다.

반복 접근 방식에서는 재작업의 필요성을 미리 파악할 수 있습니다. 이러한 재작업은 초기에 집중되며 조기 아키텍처 프로토타입에서 발견한 문제점을 수정해야 합니다. 또한 실행 가능한 프로토타입을 빌드하기 위해

스텁 및 테스트 발판을 빌드해야 합니다. 이러한 스텁 및 테스트 발판은 구현의 완성도가 높아지고 견고해짐에 따라 나중에 바뀝니다. 올바른 반복 프로젝트에서는 폐기 또는 재작업의 백분율이 급속히 감소해야 합니다. 즉, 아키텍처가 안정화되고 어려운 문제가 해결됨에 따라 프로젝트 전체에 영향을 주는 변경사항이 감소해야 합니다.

함정: 프로젝트의 비수렴성

반복적 개발은 각 반복에서 모든 내용을 폐기하는 것이 아닙니다. 폐기 및 재작업은 특히 아키텍처의 기준선이 LCA 이정표인 경우 반복 과정에서 계속 감소해야 합니다. 개발자는 종종 반복적 개발을 통해 보다 나은 기법을 도입하거나 재작업을 수행하려고 합니다. 프로젝트 관리자는 문제가 없는 정상 요소 또는 올바른 요소에 대한 재작업이 이루어지지 않도록 주의해야 합니다. 또한 개발 팀의 규모가 커지고 일부 인력의 이동에 따라 신입 인력이 투입됩니다. 이들 신입 인력은 프로젝트 수행에 대한 자신만의 아이디어를 갖고 있습니다. 고객(또는 프로젝트의 고객 대표: 마케팅, 제품 관리) 또한 반복적 개발의 특성인 자유를 남용함으로써 변경사항을 수용하거나 요구사항을 계속 변경 또는 추가할 수 있습니다. 이로 인해 종종 "요구사항 변형"이라는 결과가 발생합니다. 프로젝트 관리자는 절충과 우선순위 협상에 있어 냉정한 자세를 갖추어야 합니다. LCA 이정표 부근에서 요구사항이 기준선으로 작성되며, 스케줄 및 예산을 재협상하지 않는 한 모든 변경사항의 비용은 한정적입니다. 따라서 무엇인가를 얻는다는 것은 곧 다른 것을 버려야 함을 의미합니다.

또한 여기서 "완벽은 올바름의 적"이라는 사실을 명심해야 합니다 (불어로는 "Le mieux est l’ennemi du bien"이라고 합니다).

함정: 성급한 시작과 세부사항 결정 회피

반복적 개발은 계속 불분명한 상태의 개발을 의미하지 않습니다. 단지 팀을 쉬지 못하도록 하기 위해 또는 명확한 목적이 갑자기 나타날 것이라는 희망으로 무작정 디자인 및 코딩을 시작해서는 안됩니다. 즉, 반복적 개발에서도 먼저 명확한 목적을 정의하고 서면으로 작성해야 하며 모든 관계자의 동의를 얻은 다음 정제, 확장하고 다시 동의를 얻어야 합니다. 반복적 개발에 있어 장점은 요구사항을 디자인, 코딩, 통합, 테스트하고 유효성을 검증하기 전에 모든 요구사항을 명시하지 않아도 된다는 것입니다.

함정: 자신이 이룬 성공의 희생자가 됨

프로젝트 종료 시점에 이르면 또 다른 위험성이 존재합니다. 바로 "고객의 입장" 변화입니다. 즉, 사용자는 결과물을 기대하지 않던 처음 입장에서 팀이 실제로 어려운 일을 해낼 수 있을 것이라는 신뢰를 갖게 됩니다. 물론 프로젝트에 대한 외부 인식의 변화는 반가운 뉴스지만, 오늘 받은 제품에 만족해하던 사용자가 다음 날에는 기능이 완벽하지 않다는 사실에 실망하게 됩니다. 이는 프로젝트에 있어 나쁜 뉴스입니다. 첫 번째 및 두 번째 베타를 작성하는 과정에서 많은 기능에 대해 첫 번째 릴리스 포함 여부를 결정해야 하며 특정 시점에 이르면 이 문제가 가장 큰 문제가 됩니다. 프로젝트 관리자는 처음에는 가능한 최소 기능만 제공하려고 했지만 결국 모든 최종 요구사항을 첫 번째 인도물의 "핵심 기능"으로 결정하게 됩니다. 이러한 경우 모든 미해결 항목이 "최"우선순위 상태로 승격됩니다. 그러나 실제로는 해야할 일도 필요한 시간도 모두 그대로입니다. 외부의 인식은 변화했지만 우선순위 결정 역시 매우 중요합니다.

이처럼 중요한 시점에서 프로젝트 관리자가 겁을 내고 모든 요청에 굴복하기 시작하면 프로젝트 스케줄에 다시 심각한 문제가 발생하게 됩니다. 이러한 경우 새로운 요청에 굴복하지 않고 계속 냉정하게 대처해야 합니다. 이 시점에서는 기존의 것을 새로운 것으로 절충하는 것만으로도 위험성이 증가할 수 있습니다. 특별한 주의 없이는 성공이 실패로 전락할 수 있습니다.

소프트웨어 우선

폭포수형 접근 방식에서는 "스펙"(즉, 문제점 영역 설명)과, 해당 스펙의 올바른 작성, 완성, 마무리 및 사인오프를 특히 강조합니다. 반면에 반복 프로세스에서는 개발하는 소프트웨어가 가장 중요합니다. 소프트웨어 아키텍처(즉, 솔루션 영역 설명)는 초기 라이프사이클 결정을 이끌어내야 합니다. 고객은 스펙을 구입하는 것이 아니므로 소프트웨어 제품 자체에 주의를 기울여야 하며 스펙과 소프트웨어를 함께 발전시켜야 합니다. 이러한 "소프트웨어 우선" 주의는 여러 팀에 영향을 줍니다. 예를 들어, 테스터는 테스트를 시작하기 전에 많은 사전

통지를 통해 완전하고 안정적인 스펙을 받는 데 익숙해져 있습니다. 반면에 반복적 개발에서는 테스터가 지속적으로 발전하는 스펙과 요구사항을 바탕으로 즉시 작업을 시작합니다.

함정: 관리 아티팩트에 대한 지나친 강조

일부 관리자는 자신이 프로젝트 관리자이므로 가능한 가장 우수한 관리 아티팩트 세트를 만드는 데 주력해야 하며 관리 아티팩트가 가장 중요한 핵심 요소라고 주장합니다. 그러나 이는 사실과 다릅니다. 올바른 관리도 중요하지만 프로젝트 관리자는 결국 가장 우수한 제품을 최종 제품으로 만들어야 합니다. 프로젝트 관리는 최선의 관리 노력을 입증함으로써 프로젝트 실패에 따른 프로젝트 관리자의 책임을 회피하는 수단이 되어서는 안됩니다. 또한 과거 잘못된 요구사항 관리로 인한 실패 경험이 있으므로 가장 올바른 스펙을 개발하는 데 주력할 수 있습니다. 그러나 해당 제품이 버그가 있고 속도가 느리고 불안정하며 견고하지 못한 경우 이러한 노력도 아무런 의미가 없습니다.

어려운 문제의 조기 처리

폭포수형 접근 방식에서는 여러 가지 어려운 문제점, 위험성 요인 및 실제로 알려져 있지 않은 사항이 계획 프로세스에서 배제되어 까다로운 시스템 통합 활동에서 해결되어야 합니다. 이러한 경우 문제를 서면으로 처리하고 테스터 등과 같은 이해 당사자(stakeholder), 하드웨어 플랫폼, 실제 사용자 또는 실제 환경이 많이 반영되지 않는 프로젝트 전반부는 상대적으로 원만하게 진행됩니다. 그러나 프로젝트가 통합 단계에 접어들면서 모든 상황은 역전됩니다. 반복적 개발에서는 일반적으로 위험성 및 알려져 있지 않은 사항을 기반으로 계획이 수립되므로 처음 시작할 때부터 까다로운 작업을 수행해야 합니다. 어렵고 중요하며 때때로 하위 레벨의 기술 문제 또한 나중으로 미루지 않고 즉시 처리해야 합니다. 즉, 반복적 개발에서는 자기 자신과 다른 사람 모두에게 거짓말을 오래 할 수 없습니다. 따라서 반복적 접근 방식에서는 실패할 소프트웨어 프로젝트라면 초기에 실패하게 됩니다.

비슷한 예로, 한 대학교의 수업을 들 수 있습니다. 교수는 학기 전반부를 상대적으로 기본적인 개념 설명에 할애함으로써 학생들에게 최소한의 노력으로 중간 고사에서 좋은 점수를 얻을 수 있는 쉬운 과목이라는 인상을 줍니다. 그러나 학기말이 가까워오자 수업 속도가 빨라지기 시작하며 모든 어려운 주제가 기말 고사 직전에 다루어집니다. 이러한 경우 가장 일반적인 시나리오는 대부분의 학생이 실패하여 기말 시험 성적을 낮게 받는 것입니다. 교수는 이러한 시나리오가 매년, 수업마다 반복된다는 사실에 놀라게 됩니다. 보다 현명한 접근 방식은 수업 전반부에 집중하는 방법으로서, 중간 고사 이전에 어려운 과제를 포함하여 수업 내용의 60%를 다루는 것입니다. 반복 프로젝트 관리에 대한 상관은 심각하지 않은 문제점을 해결하고 사소한 타스크를 수행하는 데 초기에 중요한 시간을 낭비하지 않는 것입니다. 시작 시 기술적 실패의 가장 일반적인 원인은 모든 시간을 쉬운 일에 할애했기 때문입니다."

함정: 회피

이는 민감한 문제이므로 문제점 파악을 위해 심사숙고해야 한다고 말하는 경우가 종종 있습니다. 이러한 문제에 대한 해결책은 나중으로 연기함으로써 생각해볼 시간을 얻을 수 있다고 말합니다. 이러한 경우 프로젝트에서 쉬운 타스크를 모두 수행하며 어려운 문제점에 대해서는 그다지 신경을 쓰지 않게 됩니다. 그러나 솔루션이 필요한 시점에 이르면 솔루션 개발과 의사 결정이 성급하게 수행되거나 프로젝트 자체가 실패합니다. 올바른 방법은 이와 반대로 어려운 일을 먼저 처리하는 것입니다. 일부 이유 때문에 프로젝트를 중지해야 할 경우, 시간과 비용을 낭비하기 전에 가능한 빨리 중지시키는 것이 바람직합니다."

함정: 새로운 위험성 간과

도입/인식(Inception) 과정에서 위험성 분석을 수행하고 해당 결과를 계획에 반영했지만 나중에 프로젝트에서 전개될 위험성에 대해서는 잊고 있었습니다. 그러나 해당 위험성이 나중에 갑자기 나타날 수 있습니다. 따라서 위험성은 월 단위 또는 주 단위로 지속적으로 재평가해야 합니다. 처음에 작성한 위험성 목록은 임시 목록임에 유의해야 합니다. 소프트웨어를 우선하여 팀이 구체적인 개발을 수행하기 시작하면 많은 다른 위험성이 발견됩니다.

다양한 라이프사이클 모델에 따른 충돌

반복 프로젝트의 관리자는 경영진, 고객 및 계약자와 같이 반복적 개발을 채택하지 않은, 심지어 그 특성도 이해하지 못하는 다른 그룹과의 충돌을 종종 경험하게 됩니다. 이러한 그룹은 주 이정표에서 완전하고 안정적인 아티팩트를 예상하며, 요구사항을 부분적으로 검토하려고 하지 않습니다. 또한 재작업에 대한 두려움을 갖고 있으며 정형화되지 않은 아키텍처 프로토타입의 목적 또는 가치를 이해하지 못합니다. 이들은 반복을 무의미한 실수, 기술의 남용, 스펙 확정 이전의 코드 개발 및 임시 코드 테스트 정도로 인식합니다.

최소한 의도와 계획은 명확하게 표시되어야 합니다. 반복 접근 방식을 자신만 이해하고 팀 구성원들은 완전히 이해하지 못하는 경우 나중에 문제가 발생합니다.

프로젝트 관리자는 외부에서 팀을 방해하지 못하게 하기 위해 외부 공격 및 정책으로부터 팀을 보호해야 합니다. 즉, 프로젝트 관리자는 자동차의 완충 장치와 같은 역할을 수행해야 합니다. 또한 "프로젝트를 안정적으로 지휘"하기 위해 프로젝트 관리자는 외부 커뮤니티와 신뢰 및 신의 관계를 구축해야 합니다. 따라서 특히 많은 사람이 "계획"의 불안정한 측면을 염려하므로 가시성 및 "계획 추적"이 중요합니다. 실제로 이는 매우 중요한 사항입니다.

함정: 다양한 그룹과 그룹별 상이한 스케줄

모든 그룹(또는 팀 또는 하청업체)은 동일한 단계 및 반복 계획에 따라 쉽고 효과적으로 운영할 수 있습니다. 프로젝트 관리자는 종종 각 개별 팀의 스케줄을 세부적으로 조정함으로써 시간을 최적화하며 그 결과 각 팀은 고유한 반복 스케줄을 갖게 됩니다. 그러나 이러한 경우 기존의 이점은 모두 없어지고 모든 팀은 스케줄이 느린 그룹에 맞춰 작업을 수행할 수 밖에 없습니다. 가능한 모든 작업자는 동일한 속도를 유지해야 합니다.

함정: 도입/인식(Inception) 단계의 고정 가격 입찰

많은 프로젝트가 도입/인식 단계에서 지나치게 빨리 계약 개발을 위한 입찰에 참여하게 됩니다. 반복적 개발에 있어 모든 관계자에게 가장 적합한 입찰 시점은 LCA 이정표(정제(Elaboration) 종료)입니다. 입찰과 관련한 비법은 없으며 이해 당사자(stakeholder)에게 반복적 개발의 이점을 보여주는 협상과 교육, 또한 궁극적으로 2 단계 입찰 프로세스가 수행됩니다.

상이한 진행상태 표시

아티팩트가 완전하거나 안정적이지 않으며 여러 차례 증분을 통해 재작업이 수행되므로 진행상태를 설명하는 전통적인 획득 가치(EV) 시스템은 다릅니다. 아티팩트가 획득 가치(EV) 시스템의 특정 값을 가지며 작성한 첫 번째 반복에서 인정을 받는 경우 진행상태에 대한 평가는 전반적으로 낙관적입니다. 둘 또는 세 번의 반복 후에 안정될 때만 인정을 받는 경우 진행상태에 대한 측정 결과는 매우 비관적입니다. 따라서 이러한 접근 방식으로 진행상태를 모니터하는 경우, 아티팩트를 여러 부분으로 분해해야 합니다(예를 들어, 초기 문서(40%), 첫 번째 개정(25%), 두 번째 개정(20%), 최종 문서(15%)). 각 부분에는 값이 할당되어야 합니다. 이러한 경우 각 요소를 완성하지 않고도 획득 가치(EV) 시스템을 사용할 수 있습니다.

또 다른 방법은 반복을 기준으로 획득 가치(EV)를 구성하고 평가 기준에 따라 값을 평가하는 것입니다. 이러한 경우 상태 평가에 보고된 중간 추적 점수(일반적으로 월 단위)는 반복 계획에 따라 빌드됩니다. 이를 위해서는 전통적인 요구사항 스펙, 디자인 스펙 등보다 세부적으로 아티팩트를 추적해야 합니다. 다양한 유스 케이스, 테스트 케이스 등의 완료를 추적하기 때문입니다.

Walker Royce 는 "프로젝트 관리자는 변경사항(텍스트 페이지 수 및 코드 행 수 계산보다 요구사항, 디자인 및 코드의 변경사항) 측정 및 모니터링에 보다 집중해야 합니다"라고 주장합니다. (아래 "추가 참조" 참고) 여기에 Joe Marasco 는 변경사항뿐만 아니라 변동(churn)에도 관심을 기울여야 한다고 주장합니다. 여러 번 변경을 거친 후 원래 시작점으로 돌아오게 되면 보다 심각한 문제점이 발생했음을 의미합니다."

긍정적인 측면에서 보면, 일찍 실행되는 구체적인 소프트웨어를 갖게 되며 현명한 관리자가 초기에 신뢰성 점수를 얻기 위해 이 소프트웨어를 사용할 수 있습니다. 이 소프트웨어는 수백 개의 선택란을 선택하여 완성하고

검토한 저작물보다 의미있는 방식으로 진행상태를 나타낼 수 있습니다. 또한 엔지니어는 "작업 방식의 문서화"보다 "작업 방식의 시연"을 선호합니다. 따라서 문서화보다 시연 작업을 먼저 수행합니다.

반복 횟수, 지속 기간 및 컨텐츠 결정

먼저, 반복적 개발 방식을 처음 접하는 관리자는 일반적으로 반복 컨텐츠를 결정하는 데 어려움이 있습니다. 초기에 이 계획은 위험성, 기술 및 프로그램 측면과, 구성 중인 시스템의 기능 심각도에 따라 수행됩니다 (RUP 는 반복 횟수 및 지속 기간 결정을 위한 가이드라인을 제공합니다). 이 기준은 또한 전체 라이프사이클에서 발전됩니다. 구현/구축(Construction) 단계에서의 계획의 목적은 특정 기능 또는 특정 서브시스템을 완성하기 위한 것이며 전이(Transition) 단계에서는 문제점을 수정하고 시스템의 견고성 및 성능을 향상시키기 위한 것입니다.

함정: 첫 번째 반복에 대한 지나친 부담

앞에서 어려운 문제점을 먼저 처리해야 함을 설명했습니다. 반면에 지나치게 부정적인 입장을 취하는 것 또한 실패로 연결될 수 있습니다. 즉, 너무 많은 문제를 제기하고 첫 번째 또는 초기 몇몇 반복에서 지나치게 많은 문제를 처리하려고 하는 경우 다른 요소를 인식하지 못하게 됩니다. 예를 들어, 팀을 구성하고 훈련시켜야 하며 새 기법을 학습하고 새 도구를 구입해야 합니다. 또한 문제 도메인이 다수의 개발자에게 새로운 영역인 경우가 종종 있으며 이러한 경우 첫 번째 반복에 너무 많은 문제가 집중되어 전체 반복 접근 방식에 대한 신뢰도가 하락하거나 반복 자체가 중단됩니다. 즉, 실제로 실행되는 내용이 없는 상태에서 반복이 완료된 것으로 선언됩니다. 예를 들어, 아무런 교훈을 얻지 못하거나 반복적 개발에 따른 대부분의 이점이 누락되어 있는 경우입니다.

불확실하거나 위기에 직면한 경우에는 해당 문제점, 솔루션, 팀을 축소하는 것이 효과적인 방법입니다. 완전성은 라이프사이클 후반부의 관심사항입니다. 라이프사이클 초반부의 관리자 관심사항은 "적절한 불완전성"이어야 합니다. 첫 번째 반복에 너무 많은 목적이 포함되는 경우, 두 반복에 나누어 분할한 후 먼저 달성해야 하는 목표에 따라 과감하게 우선순위를 부여해야 합니다.

프로젝트 초기에는 보다 단순하고 간결한 목적을 설정하는 것이 좋습니다. 그렇다고 쉬운 목적을 말하는 것은 절대 아닙니다. 프로세스 초기에 정확하고 실질적인 결과를 얻음으로써 팀의 사기를 높일 수 있습니다. 첫 번째 이정표가 없는 프로젝트는 일반적으로 복구가 불가능하며 이후 대담한 노력으로도 회복될 수 없습니다. 따라서 초기 이정표가 많이 누락되지 않도록 주의해야 합니다.

함정: 반복의 남용

프로젝트에서 매일 또는 매주 수행되는 빌드와 반복을 혼동해서는 안됩니다. 반복의 계획, 모니터링 및 평가에는 일정 오버헤드가 발생하므로 이 접근 방식에 익숙하지 않은 조직에서는 첫 번째 프로젝트에서 지나치게 빠른 반복을 시도해서는 안됩니다. 반복의 지속 기간은 또한 조직의 규모, 지리적 분산 정도 및 관련 개별 조직의 수를 고려해야 합니다. 또한 "6 을 기준으로 한 오차 범위 3"의 법칙을 재검토해야 합니다.

함정: 반복의 겹침

또 다른 함정은 반복이 많이 겹치는 경우입니다. GANTT 차트에서 시작할 때 많은 활동(예: 세부 분석 시작, 현재 반복을 완료하기 전에 다음 반복 디자인 및 코딩, 현재 반복에서의 학습)을 중복시키면서 현재 반복의 마지막 다섯 번째로 다음 반복을 계획하는 것이 매력적으로 보일 수도 있지만 이러한 경우 문제점이 발생합니다. 즉, 작업자에 따라 현재 반복에 대한 자신의 컨트리뷰션을 이행 또는 완료하지 않거나 수정 작업에 적극적이지 않거나 일부 또는 모든 피드백을 다음 반복에만 적용하려고 할 수 있습니다. 또한 특정 소프트웨어 파트가 다음 작업을 지원하지 못할 수도 있습니다. 현재 반복과 관련이 없는 작업을 수행하기 위해 인력을 재배치할 수도 있지만 이러한 조치는 특별한 경우에만 또한 최소 수준으로 수행되어야 합니다. 이 문제점은 일반적으로 특정 조직 구성원의 스킬이 부족하거나 경직된 조직에서 발생합니다. 예를 들어, Joe 의 업무는 분석가로서 분석 업무만 수행할 수 있으며 본인도 다른 업무를 원하지 않습니다. 즉, 그는 디자인, 구현 또는 테스트에는 참여하지 않으려고 합니다. 또 다른 부정적인 예로 대규모 명령 및 제어 프로젝트를 들 수 있습니다. 이 프로젝트에는 많은

반복이 중복되어 동시에 여러 곳에서 함께 실행되므로 관리자가 반복에 전체 인력을 배분해야 합니다. 또한 이전 반복에서 다음 반복으로의 교훈 피드백도 기대할 수 없는 상황입니다.

일반적인 비생산적인 반복 패턴 예는 그림 3 을 참조하십시오.

Teams not synchronized -> chaos

First iteration too loaded -> panic

Too much overlap -> no feedback loop

그림 3: 위험한 반복 패턴

올바른 프로젝트 관리자와 올바른 설계자

소프트웨어 프로젝트의 성공을 위해서는 올바른 프로젝트 관리자와 설계자가 필요합니다. 최상의 관리 및 반복적 개발의 경우라도 올바른 아키텍처 없이는 성공적인 제품을 만들어낼 수 없습니다. 반대로 프로젝트를 올바르게 관리하지 않으면 이상적인 아키텍처라도 실패할 수 밖에 없습니다. 따라서 두 역할의 균형이 중요하며 프로젝트 관리에만 초점을 맞추면 프로젝트가 성공할 수 없습니다. 프로젝트 관리자는 아키텍처를 무시할 수 없으며 아키텍처 전문가와 도메인 전문가가 함께 초기 반복에서 수행해야 할 작업의 20%를 결정해야 합니다.

함정: 프로젝트 관리자와 설계자 역할의 동시 수행

프로젝트 관리자와 설계자 역할을 *한 사람*이 수행하는 경우는 소규모 프로젝트(5 – 10 명의 팀 구성원)에만 해당됩니다. 대규모 프로젝트의 경우, 프로젝트 관리자와 설계자 역할을 한 사람이 수행하면 일반적으로 프로젝트가 올바르게 관리되지 않거나 올바른 아키텍처를 구성할 수 없습니다. 우선 역할마다 다른 스킬이 필요하기 때문입니다. 또한 각 역할마다 많은 책임과 업무 시간이 필요합니다. 따라서 프로젝트 관리자와 설계자는 지속적인 커뮤니케이션 및 협력과 타협 관계를 이루어야 합니다. 이 두 역할은 영화 감독과 영화 제작자의 관계와도 같아서 공통 목적을 지향하면서도 전혀 다른 업무 측면을 담당합니다. 두 역할을 한 사람이 수행하는 경우 프로젝트 성공을 바라기 어렵습니다.

결론

앞에서 설명한 문제점과 함정을 볼 때 반복적 개발에 대한 회의가 들 수도 있습니다. 실제로, 반복적 개발의 계획 및 실행이 쉽지는 않지만 이러한 모든 문제를 시스템적으로 해결할 수 있는 여러 방법과 기법이 있습니다. 또한

신뢰할 수 있는 고품질의 소프트웨어 제품 개발 측면에서 볼 때 불편함보다 그에 따른 이점이 더 많은 방법임을 알 수 있습니다. 반복적 개발의 핵심 주제는 다음과 같이 요약할 수 있습니다. 1) 위험성을 먼저 처리하지 않으면 결국 위험에 빠지게 됩니다 (*참조 자료 및 추가 참조*에 있는 Tom Gilb 의 서적 참고). 2) 소프트웨어가 가장 우선시되어야 합니다. 3) 폐기 및 재작업을 인정합니다. 4) 협력할 프로젝트 관리자와 설계자를 선택합니다. 5) 반복적 개발의 이점을 활용합니다.

폭포수형 모델은 관리자에게 유리하며 엔지니어링 팀에는 어려운 모델입니다. 반복적 개발의 경우 소프트웨어 엔지니어의 작업 방식에 맞춰져 있기 때문에 관리가 복잡할 수 있습니다. 대부분의 팀에서 엔지니어 대 관리자의 비율이 5 대 1 이상임을 감안할 때 이는 상당한 절충안입니다.

반복적 개발은 처음 수행하는 경우 일반적인 접근 방식보다 어렵지만 장기적으로 실질적인 효과를 얻을 수 있습니다. 이 방식을 이해하게 되면 보다 유능한 관리자가 될 수 있으며 복잡한 대규모 프로젝트도 쉽게 관리할 수 있습니다. 또한 팀 전체가 반복적으로 이해하고 사고하게 되면 기존의 접근 방식보다 이 방법이 더 낫습니다.

참고: John Smith, Dean Leffingwell, Joe Marasco 및 Walker Royce 는 반복 프로젝트 관리 분야의 경험을 제공함으로써 이 문서 작성에 많은 도움을 주었습니다. 이 문서의 일부는 동료 Gerhard Versteegen 이 소프트웨어 개발과 관련하여 새로 집필한 서적의 제 6 장에 포함되어 있습니다(아래 참조 자료 및 추가 참조 참고).

작성자 소개

Philippe Kruchten 은 1987 년 Rational Software 에 입사했으며 현재 Vancouver, B.C.에 거주하면서 Rational Fellow 직책을 맡고 있습니다. 프로세스 사업 본부 본부장을 역임하면서 Rational Unified Process 의 개발을 지휘했습니다. 그는 소프트웨어 아키텍처 및 디자인뿐만 아니라 소프트웨어 엔지니어링 사례 및 개발 프로세스에도 많은 관심을 갖고 있습니다. 대학에서 기계 공학을 전공했으며 프랑스에서 전산학 분야 박사 학위를 취득했습니다.

참조 자료 및 추가 참조

1. *Rational Unified Process 2000*, Rational Software, Cupertino, Ca., 2000.
2. Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, IEEE, pp.61-72.
3. Tom Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.
4. Philippe Kruchten, *The Rational Unified Process—An Introduction*, Addison Wesley Longman, 1999.
5. Walker Royce, *Software Project Management—A Unified Approach*, Addison Wesley Longman, 1999.
6. Gerhard Versteegen, *Projektmanagement mit dem Rational Unified Process*, Springer-Velag, Berlin, 2000.

Rational
the software development company

본사 안내:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
전화번호: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
전화번호: (781) 676-2400

수신자 부담 전화번호: (800) 728-1212
전자 우편: info@rational.com
웹: www.rational.com
전세계 지사 안내: www.rational.com/worldwide