창발의 아키텍처: 조합 가능하고 직교적이며 최소한의 기능을 가진 소프트웨어 설계

서론: 원시 기능의 힘

가장 강력하고 회복력 있는 소프트웨어 시스템은 거대한 단일체로 한 번에 구축되지 않는다. 오히려, 그것들은 단순하고, 독립적이며, 재사용 가능한 구성 요소들의 창의적인 조합으로부터 창발(emerge)한다. 이는 우아한 소프트웨어 설계를 관통하는 핵심 원리이며, 조합성(Composability), 직교성(Orthogonality), 그리고 최소주의(Minimalism)라는 세 가지 기본 개념에 의해 뒷받침된다. 이 보고서는 이 '설계의 삼위일체'가 어떻게 개별적으로 작동하고 상호작용하여, 예측할 수 없는 복잡하고 가치 있는 사용자 행동을 이끌어내는 시스템을 만들어내는지 심층적으로 탐구한다.
이 원칙들을 이해하는 것은 단순히 기술적 지식을 쌓는 것을 넘어, 소프트웨어를 바라보는 근본적인 관점을 바꾸는 일이다. 시스템의 힘이 "프로그램 자체보다 프로그램들 간의 관계에서 더 많이 나온다"는 유닉스 철학의 오래된 지혜를 현대적으로 재해석하는 것이기도 하다.

핵심 개념 정의

이 여정을 시작하기에 앞서, 우리의 논의를 지탱할 세 가지 기둥을 명확히 정의해야 한다.

- 직교성 **(Orthogonality):** 기하학에서 유래한 이 개념은 소프트웨어 설계에서 '독립성'을 의미한다. 직교적으로 설계된 시스템에서 한 구성요소를 변경해도 다른 구성요소에 예기치 않은 부작용(side effect)을 일으키지 않는다. 예를 들어, 사용자 인터페이스(UI)를 수정하는 것이 데이터베이스 스키마의 변경을 요구하지 않는다면, UI와 데이터베이스는 직교적이라고 할 수 있다. 이 원칙은 본질적으로 높은 응집도(High Cohesion)와 낮은 결합도(Loose Coupling)라는 두 가지 고전적인 소프트웨어 공학 원칙이 성공적으로 결합된 결과물이다. 각 컴포넌트는 하나의 잘 정의된 책임(높은 응집도)을 가지며, 다른 컴포넌트의 내부 구현에 대해서는 최소한으로만 알아야 한다(낮은 결합도).
- 조합성 **(Composability):** 이는 마치 레고 블록으로 복잡한 구조물을 만드는 것과 같이, 작고 자율적인 구성 요소들을 조립하여 시스템을 구축하는 설계 원칙이다. 단순한 모듈화(modularity)가 시스템을 분리된 부분으로 나누는 데 초점을 맞춘다면, 조합성은 이 부분들이 어떻게 서로 원활하게 상호 운용되고 다른 맥락에서 재사용될 수 있는지에 더 중점을 둔다. 개발자는 이미 존재하는 구성 요소들을 '플러그 앤 플레이' 방식으로 결합하여 더 크고 복잡한 시스템을 신속하게 구축할 수 있다.
- 최소주의 **(Minimalism):** 이는 "시스템의 핵심적이고 광범위한 품질을 달성하기 위해 절대적으로 필요한 것만 아키텍처로 수행한다"는 원칙이다. 이 접근법은 불필요한 기능을 미리 구현하지 말라는 'YAGNI(You Aren't Gonna Need It)' 원칙과 맞닿아 있으며, 과도한 엔지니어링을 방지한다. 최소주의는 변경의 마찰을 줄이고, 개발팀의 자율성을 높이며, 아키텍트가 시스템의 가장 중요한 본질에 집중하도록 강제하는 강력한 도구다.

핵심 논제와 보고서의 여정

이 보고서의 핵심 논제는 이 세 가지 원칙이 함께 적용될 때, '창발(emergence)' 현상을 위한 비옥한 토양이 만들어진다는 것이다. 창발이란 시스템의 핵심 원시 기능(primitive)들의 단순한 상호작용으로부터 복잡하고 가치 있으며, 종종 설계자가 처음에는 예상하지 못했던 사용자

행동이 자연스럽게 나타나는 현상을 의미한다.
본 보고서는 이러한 설계 철학의 지적 계보를 따라가는 여정을 안내할 것이다. 먼저 유닉스(Unix) 커맨드 라인에서 그 철학적 기원을 탐색하고, 워드 커닝햄(Ward Cunningham)의 오리지널 위키(Wiki)를 원형적 사례로 심층 분석한다. 그 후 비지칼크(VisiCalc), 노션(Notion), 롬 리서치(Roam Research), 피그마(Figma), 그리고 언리얼 엔진의 블루프린트(Unreal Engine's Blueprints)와 같은 현대적 및 역사적 애플리케이션들을 비교 연구하며 이 원칙들이 어떻게 다양한 형태로 구현되었는지 살펴볼 것이다. 마지막으로, 이 모든 통찰을 종합하여 야심 찬 아키텍트가 자신의 설계 역량을 키울 수 있는 실용적인 지침을 제공하며 마무리할 것이다.

제**1**장**:** 기원 **–** 유닉스 철학과 파이프라인의 힘

조합 가능한 설계의 역사적, 개념적 기반을 탐색하기 위해서는 1970년대 벨 연구소(Bell Labs)로 거슬러 올라가야 한다. 그곳에서 탄생한 유닉스 운영체제와 그 핵심 철학은 오늘날 우리가 논하는 모든 원칙의 씨앗을 품고 있었다. 특히, '파이프와 필터(pipes and filters)' 아키텍처는 몇 가지 단순한 규칙을 엄격하게 적용함으로써 어떻게 강력하고 재사용 가능한 도구들의 생태계 전체가 창발할 수 있는지를 보여주는 최초의 증거였다.

더그 맥클로이의 유닉스 철학 해부

유닉스 파이프의 발명가이자 유닉스 전통의 창시자 중 한 명인 더그 맥클로이(Doug McIlroy)는 1978년 벨 시스템 기술 저널(Bell System Technical Journal)을 통해 유닉스 철학을 다음과 같이 간결하게 문서화했다. 이 세 가지 원칙은 각각 직교성, 조합성, 그리고 이를 가능하게 하는 구체적인 구현 전략을 명확하게 제시한다.

1. **"**각 프로그램은 한 가지 일을 잘해야 한다 **(Write programs that do one thing and do it well)."** 이는 프로그램 수준에서 직교성을 가장 잘 표현하는 원칙이다. grep(텍스트 검색), sort(정렬), wc(단어 수 계산)와 같은 유닉스 명령어들은 각각 하나의 명확하게 정의된 책임만을 수행한다. 이 원칙은 프로그램의 복잡성을 억제하고 명확성을 강제하며, 각 도구를 예측 가능하고 신뢰할 수 있게 만든다. 새로운 기능이 필요할 때 기존 프로그램을 복잡하게 만드는 대신, 새로운 도구를 만드는 것을 장려한다.
2. **"**모든 프로그램이 함께 작동하도록 작성하라 **(Write programs to work together)."** 이는 조합성의 핵심 원칙이다. 모든 프로그램은 자신의 출력이 아직 알려지지 않은 다른 프로그램의 입력이 될 수 있다는 기대를 가지고 설계되어야 한다. 이는 각 프로그램이 독립적인 동시에, 더 큰 워크플로우의 일부가 될 수 있는 잠재력을 지녀야 함을 의미한다.
3. **"**텍스트 스트림을 처리하도록 프로그램을 작성하라**.** 그것이 보편적인 인터페이스이기 때문이다 **(Write programs to handle text streams, because that is a universal interface)."** 앞선 두 철학을 현실로 만드는 결정적인 구현 세부사항이다. 텍스트 스트림은 각 직교적 도구들을 연결하는 표준화되고 단순한 '파이프' 또는 'API' 역할을 한다. 이 보편적인 인터페이스 덕분에, 서로의 내부 구현에 대해 전혀 알지 못하는 프로그램들이 거의 무한한 조합으로 연결될 수 있다.

아키텍처 혁명으로서의 파이프**(|)**

맥클로이가 발명한 파이프(|)는 단순한 셸(shell) 문법 그 이상이었다. 그것은 소프트웨어 아키텍처에 대한 사고방식을 근본적으로 바꾼 혁명이었다. 코루틴(coroutine) 개념에 매료되었던 맥클로이는 "프로세스들을 정원 호스처럼 나사로 조이는(screwing processes together like garden hose)" 아이디어를 구상했고, 이것이 파이프의 개념으로 발전했다. 기술적으로 파이프는 한 프로세스의 표준 출력(stdout)을 다른 프로세스의 표준 입력(stdin)으로

직접 연결하는 커널 수준의 메커니즘이다. 파이프가 도입되기 전에는 한 프로그램의 결과를 다른 프로그램에서 사용하려면 중간 파일을 생성하고, 디스크에 썼다가 다시 읽어야 했다. 이 방식은 느리고 번거로우며, 임시 파일들로 시스템을 어지럽혔다. 파이프는 이 모든 비효율을 제거했다. 더 중요한 것은, 파이프라인의 모든 단계가 동시에 실행된다는 점이다. 각 프로그램은 이전 단계의 출력을 기다리면서 점진적으로 데이터를 처리하므로, 전체 데이터 처리가 끝날 때까지 기다릴 필요 없이 낮은 지연 시간으로 결과를 얻을 수 있다.

사례 연구: 커맨드 라인에서의 창발적 문제 해결

이 철학의 실제적인 힘을 이해하기 위해, 구체적인 문제를 유닉스 도구들로 해결하는 과정을 단계별로 살펴보자. 해결할 문제는 "웹 서버 접근 로그 파일에서 가장 빈번하게 접속한 상위 10개의 IP 주소를 찾는 것"이다.

1. cat access.log 가장 먼저, cat 명령어를 사용하여 로그 파일의 내용을 표준 출력으로 보낸다. cat은 파일을 읽어 그 내용을 출력하는 단 하나의 직교적인 기능을 수행한다. 이것이 파이프라인의 데이터 소스가 된다.
2. | awk '{print $1}' cat의 출력(텍스트 스트림)은 파이프를 통해 awk 명령어의 입력으로 전달된다. awk는 텍스트를 한 줄씩 처리하는 강력한 도구다. 여기서는 각 줄의 첫 번째 필드(IP 주소)만을 추출하여 출력하는 직교적인 역할을 수행한다.
3. | sort 추출된 IP 주소 목록은 다시 파이프를 통해 sort 명령어의 입력으로 들어간다. sort는 입력된 텍스트 줄들을 알파벳 순서로 정렬한다. 이 단계는 동일한 IP 주소들을 서로 인접하게 만들어 다음 단계를 준비시키는 역할을 한다.
4. | uniq -c 정렬된 IP 목록은 uniq -c로 전달된다. uniq는 중복된 연속적인 줄을 제거하는 도구이며, -c 옵션은 각 줄이 몇 번 반복되었는지 세어서 출력하는 직교적인 기능을 추가한다. 이 단계의 결과는 "반복 횟수 IP주소" 형태의 텍스트가 된다.
5. | sort -rn 이제 우리는 sort 명령어를 다시 사용하지만, 다른 직교적인 목적으로 활용한다. -r 옵션은 역순(내림차순)으로, -n 옵션은 숫자 기준으로 정렬하라는 의미다. 따라서 이 단계는 반복 횟수가 가장 많은 IP 주소가 맨 위에 오도록 목록을 재정렬한다.
6. | head -n 10 마지막으로, 재정렬된 목록은 head -n 10으로 전달된다. head는 입력의 첫 부분만 출력하는 도구이며, -n 10 옵션은 처음 10줄만 출력하도록 지정한다. 이로써 우리는 원래의 복잡한 문제를 해결한 최종 결과를 얻게 된다.

이 과정에서 주목해야 할 점은, 여기에 사용된 cat, awk, sort, uniq, head 중 어느 것도 "로그 분석"이라는 특정 목적을 위해 설계되지 않았다는 것이다. 각 도구는 지극히 일반적이고 직교적인 기능을 수행할 뿐이다. 복잡한 문제 해결 능력은 이 단순한 도구들이 '텍스트 스트림'이라는 보편적인 인터페이스와 '파이프'라는 저비용 연결 메커니즘을 통해 조합될 때 창발적으로 나타난다.

이러한 설계 방식은 그 자체로 중요한 교훈을 담고 있다. 첫째, 진정으로 창의적인 조합을 이끌어내기 위해서는 구성 요소들을 연결하는 비용이 거의 '0'에 가까워야 한다. 유닉스에서는 파이프와 평문 텍스트가 그 역할을 했다. 이는 현대의 마이크로서비스 아키텍처가 성공하기 위해서는 API와 같은 연결 계층의 품질, 단순성, 보편성이 얼마나 중요한지를 시사한다. 복잡하거나 독점적인 API는 창발적 조합의 적이다.

둘째, 최소주의는 우아한 설계를 위한 강력한 강제 기능(forcing function)이 될 수 있다. 초기 유닉스 개발자들은 8KB에 불과한 메모리와 같은 극심한 하드웨어 제약 속에서 작업해야 했다. 이러한 제약은 미학적 선택이 아니라 생존 전략이었다. 이 제약은 그들로 하여금 작고 단일 목적을 가진 프로그램을 만들도록 '강제'했고, 이는 다시 더 큰 문제를 해결하기 위해 그 프로그램들을 결합할 방법을 발명하도록 '강제'했으며, 그 결과가 바로 파이프였다. 이는 의도적으로 제약을 가하는 것이 거대하고 비대한 단일체 대신 더 우아하고 조합 가능한 해결책을 발견하도록 아키텍트를 이끌 수 있음을 보여준다. 맥클로이가 말했듯, 이는 "고통을

통한 구원(salvation through suffering)"이었다.

제2장: 원형 – 워드 커닝햄의 위키위키웹(WikiWikiWeb) 해부

유닉스 철학이 기계와 기계(프로세스와 프로세스)의 상호작용을 위한 조합성의 원형을 제시했다면, 워드 커닝햄(Ward Cunningham)이 창조한 최초의 위키, 위키위키웹(WikiWikiWeb)은 인간과 인간의 협업을 위해 그 철학을 가장 순수하게 구현한 소프트웨어의 원형이라 할 수 있다. 위키는 극단적으로 최소화된 기능 집합이 어떻게 복잡하고 자기 조직적인 지식 창조 시스템을 가능하게 하는지를 보여주는 가장 강력한 사례다.

비전: 개발자 커뮤니케이션 문제 해결

워드 커닝햄의 원래 목표는 오늘날 우리가 아는 위키피디아와 같은 백과사전을 만드는 것이 아니었다. 그의 동기는 훨씬 더 구체적이고 실용적이었다. 그는 소프트웨어 개발 커뮤니티, 특히 '소프트웨어 디자인 패턴'을 논의하는 그룹 내에서 아이디어 교환을 더 쉽게 만들고 싶었다. 당시 지식은 이메일 리스트와 같이 선형적이고 휘발성이 강한 매체 속에서 쉽게 사라지곤 했다. 그는 "사람들의 경험을 서로 연결하여 새로운 문헌을 만들고", "이야기하고 싶은 사람들의 자연스러운 욕구를 활용"할 수 있는 기술을 원했다. 그리고 그 도구는 '저술(authoring)'에 익숙하지 않은 사람들도 편안하게 느낄 수 있어야 했다.

철학: "작동할 수 있는 가장 간단한 것 (The Simplest Thing That Could Possibly Work)"

위키의 설계는 커닝햄이 익스트림 프로그래밍(Extreme Programming) 실천 과정에서 얻은 "작동할 수 있는 가장 간단한 것"이라는 철학의 직접적인 산물이다. 이 철학은 프로그래밍 중에 "1분 이상 막혔을 때(stuck more than a minute)" 완벽하고 포괄적인 해결책을 설계하려다 마비되는 대신, 진전을 이루기 위해 필요한 절대적인 최소 기능에 집중하는 기법이다. 핵심은 일단 화면에 무언가를 띄워놓고 그것에 반응하며 점진적으로 개선해 나가는 것이다. 커닝햄 스스로 위키를 "작동할 수 있는 가장 간단한 온라인 데이터베이스"라고 묘사한 것은 이러한 철학을 명확히 보여준다.

위키 설계 원칙 해부

커닝햄은 위키위키웹을 이끌었던 핵심 설계 원칙들을 직접 명시했다. 이 원칙들은 각각이 어떻게 최소한의 기능으로 구현되었으며, 이 기능들이 조합되었을 때 어떻게 복잡한 협업이라는 창발적 행동을 유도했는지 보여주는 완벽한 로드맵이다.

원칙 (Principle)	정의 및 의도	직교적 기능(들) (Orthogonal Feature(s))	창발적 행동 (Emergent Behavior)
개방 **(Open)**	"페이지가 불완전하거나 제대로 구성되지 않았다고 생각되면, 어떤 독자든 자신이 적합하다고 생각하는 대로 편집할 수 있다."	모든 페이지에 존재하는 단 하나의 보편적인 '편집(Edit)' 버튼. 로그인이나 별도의 권한 없이 누구나 접근 가능하다.	콘텐츠에 대한 집단적 소유권(collective ownership), 지속적인 개선, 그리고 오류의 자가 수정. 이는 "인터넷에서 올바른 답을 얻는 가장 좋은

원칙 (Principle)	정의 및 의도	직교적 기능(들) (Orthogonal Feature(s))	창발적 행동 (Emergent Behavior)
			방법은 질문을 하는 것이 아니라, 틀린 답을 게시하는 것"이라는 '커닝햄의 법칙(Cunningham's Law)'을 직접적으로 가능하게 한다.
점진적 **(Incremental)**	"아직 작성되지 않은 페이지를 포함하여 다른 페이지를 인용하는 것이 가능하고 유용해야 한다."	CamelCase 링크 컨벤션. 복잡한 UI가 아닌 단순한 텍스트 패턴(단어두개를붙여쓰고각단어의첫글자를대문자로)으로 링크를 생성한다. 대상 페이지의 존재 여부와 상관없이 자동으로 링크가 만들어진다.	지식 그래프의 유기적이고 상향식(bottom-up) 성장. 중앙 계획자 없이 사용자들이 자발적으로 지식의 그물을 엮어 나간다. '존재하지 않는' 링크를 채우기 위한 새로운 콘텐츠 작성을 장려하고, 관련된 아이디어를 하나의 페이지로 통합하도록 유도하여 정보의 분산을 막는다.
유기적 **(Organic)**	"사이트의 구조와 텍스트 내용은 편집과 진화에 열려 있다."	'개방' 원칙의 편집 기능과 '점진적' 원칙의 링크 기능의 조합. 사용자는 언제든지 콘텐츠뿐만 아니라 페이지 간의 연결 구조 자체를 변경할 수 있다.	사이트 구조가 고정되지 않고 커뮤니티의 필요에 따라 끊임없이 변화하고 발전한다. 지식은 살아있는 유기체처럼 성장, 통합, 재구성된다. 이는 예상치 못한 변화를 우아하게 수용하는 애자일(Agile) 철학의 구현이다.
보편적 **(Universal)**	"편집과 구성의 메커니즘은 글쓰기의 메커니즘과 동일하므로, 모든 작성자는 자동적으로 편집자이자 구성자가 된다."	별도의 '관리자'나 '구성' 인터페이스가 존재하지 않는다. 사이트를 구조화하는 행위는 콘텐츠를 작성하는 행위(텍스트를 쓰고 CamelCase로 링크하는 것)와 완전히 동일하다.	모든 사용자가 작가, 편집자, 아키텍트의 역할을 동시에 수행할 수 있게 된다. 이는 정보를 기여하고 구조화하는 데 필요한 인지적 장벽을 극적으로 낮추어, 공동 책임 의식을 고취시킨다.
수렴적 **(Convergent)**	"유사하거나 관련된 콘텐츠를 찾아	CamelCase 링크와 모든 페이지의 제목이	정보의 파편화를 방지하고 지식의

원칙 (Principle)	정의 및 의도	직교적 기능(들) (Orthogonal Feature(s))	창발적 행동 (Emergent Behavior)
	인용함으로써 모호함과 중복을 억제하거나 제거할 수 있다."	고유한 이름 공간(flat space)에 있다는 '통합(Unified)' 원칙의 결합. 모든 페이지는 고유한 이름을 가지며, 쉽게 참조될 수 있다.	밀도를 높인다. 여러 곳에 흩어져 있던 비슷한 논의들이 자연스럽게 하나의 페이지로 모여들고, 중복된 내용은 삭제되며, 지식은 점차 정제되고 수렴된다.
관찰 가능 **(Observable)**	"사이트 내의 활동은 다른 모든 방문자가 지켜보고 검토할 수 있다."	'최근 변경(RecentChanges)' 페이지. 사이트 내의 모든 변경 사항이 시간 순서대로 기록되는 단일 페이지다.	투명성을 통해 암묵적인 사회적 통제 메커니즘을 제공한다. 반달리즘이나 부적절한 편집은 즉시 다른 사용자들에게 발견되고 수정될 수 있다. 이는 복잡한 권한 시스템 없이도 시스템의 무결성을 유지하는 데 기여한다.
단순함 **(Simple) &** 평범함 **(Mundane)**	"남용하기보다 사용하기가 더 쉽다." "소수의 (불규칙한) 텍스트 관례가 가장 유용한 페이지 마크업에 대한 접근을 제공한다."	HTML을 재발명하는 대신, 몇 가지 직관적인 텍스트 기호(예: 들여쓰기를 위한 콜론, 목록을 위한 별표)만을 사용한다.	학습 곡선이 거의 없다. 사용자는 복잡한 마크업 구문을 배우는 데 에너지를 쏟는 대신, 콘텐츠 자체에 집중할 수 있다. 이는 참여의 장벽을 낮추는 데 결정적인 역할을 한다.

이 원칙들의 상호작용을 분석하면, 위키의 설계가 단순한 기능의 나열이 아니라, 하나의 거대한 시스템 철학임을 알 수 있다. 예를 들어, '개방' 원칙은 '관찰 가능' 원칙이 없다면 무분별한 파괴 행위에 취약해질 수 있다. '점진적' 원칙은 '수렴적' 원칙을 통해 정보의 폭발을 질서 있는 성장으로 이끈다. 이처럼 각 원칙은 서로를 보완하며 전체 시스템의 회복력과 창발성을 강화한다.

이러한 설계 방식에서 두 가지 중요한 통찰을 얻을 수 있다. 첫째, 신뢰는 아키텍처적 결정이다**.** 전통적인 시스템은 방어적으로 설계되어 권한, 역할, 검증과 같은 기능을 처음부터 복잡하게 구현한다. 커닝햄은 정반대의 접근을 택했다. 그는 99%의 선의를 가진 사용자를 위해 설계하고, 1%의 악의적인 사용자에 대한 대응은 필요성이 명백해질 때 점진적으로 추가하는 방식을 선택했다. 이는 신뢰가 단순한 문화적 가치를 넘어, 시스템을 근본적으로 단순화하고 강력하게 만드는 아키텍처 전략이 될 수 있음을 보여준다.

둘째, 사용자 인터페이스가 곧 시스템이다**.** 위키에는 '콘텐츠'와 '콘텐츠 관리 도구' 사이에 구분이 없다. CamelCase를 입력하는 행위가 곧 링크를 거는 행위이며, 페이지를 편집하는 행위가 사이트의 구조를 재구성하는 행위다. 이는 '보편적' 원칙의 정수다. 이는 소프트웨어를 '사용'하는 것과 '형성'하는 것 사이의 인지적 거리가 사라질 때, 시스템이 사용자의 사고 과정에 대한 직관적이고 강력한 확장이 된다는 심오한 설계 교훈을 암시한다. 이는 복잡한 '관리자 패널'이나 '설정 화면'을 가진 시스템들과 극명한 대조를 이룬다.

제3장: 직교적 및 조합적 설계의 분류

직교성, 조합성, 최소주의의 원칙은 유닉스나 위키와 같은 과거의 유산에만 머물러 있지 않다. 이 원칙들은 시대와 기술의 변화에 따라 끊임없이 재해석되며 현대의 복잡한 문제들을 해결하는 데 활발히 적용되고 있다. 이 장에서는 다양한 분야의 영향력 있는 애플리케이션들을 분석하여, 이 원칙들이 어떻게 각기 다른 형태로 구현되고 창발적 가치를 만들어내는지 탐구한다.

3.1 보이는 계산기: 비지칼크(VisiCalc)의 직교적 혁명

1979년에 출시된 비지칼크는 최초의 스프레드시트 프로그램으로, 개인용 컴퓨터를 단순한 취미용 기기에서 필수적인 비즈니스 도구로 변모시킨 주역이다. 비지칼크의 혁신은 두 개의 강력하고 직교적인 핵심 원시 기능의 조합에서 비롯되었다.

- 핵심 원시 기능:
    1. 격자 (데이터 구조): 행과 열(A1 표기법)이라는 단순하고 직관적인 은유를 사용하여 데이터를 구성하는 방법. 이는 시스템의 상태(state)를 나타내는 직교적 구성 요소다.
    2. 자동 재계산 엔진 (로직): 하나의 셀 값이 변경될 때마다 그 셀을 참조하는 모든 다른 셀들의 값을 자동으로 즉시 업데이트하는 엔진. 이는 시스템의 연산(computation)을 담당하는 직교적 구성 요소다.
- 조합: 사용자는 격자라는 데이터 구조 위에 숫자(데이터)와 수식(로직)을 자유롭게 배치함으로써 자신만의 재무 애플리케이션을 '조합'한다. 비지칼크의 힘은 미리 만들어진 재무 모델을 제공하는 데 있지 않았다. 대신, 사용자에게 이 두 가지 단순하고 직교적인 도구를 제공하여 스스로 모델을 구축하게 한 데 있었다.
- 창발: 이 단순한 조합은 '만약 ~라면(what-if)' 시나리오 분석이라는, 이전에는 극도로 지루하고 비용이 많이 들었던 복잡한 행동을 가능하게 했다. 사용자는 단지 하나의 숫자만 바꾸면 그 파급 효과가 전체 시트에 즉시 반영되는 것을 보며 복잡한 재무적 결과를 직관적으로 탐색할 수 있었다. 이 창발적 능력이야말로 개인용 컴퓨터의 가치를 증명한 결정적 계기였다.

비지칼크의 사례는 상태(state)와 로직(logic) 사이의 직교성이 어떻게 사용자를 강력하게 만드는지를 명확히 보여준다. 비지칼크의 마법은 데이터가 '무엇'인지(격자 안의 값들)와 계산이 '어떻게' 일어나는지(재계산 엔진)를 깨끗하게 분리한 데서 나왔다. 이 분리 덕분에 사용자는 데이터와 수식을 자유롭게 조작할 수 있었고, 엔진은 그 결과를 신뢰성 있게 처리했다. 이는 데이터/상태와 행위/로직을 명확하게 분리하는 시스템이 본질적으로 최종 사용자에게 더 큰 유연성과 권한을 부여한다는 일반적인 원칙을 시사한다. 사용자는 프로그래머가 될 필요 없이 새로운 행위를 조합할 수 있게 된다.

3.2 무한한 레고 세트: 노션(Notion)의 '모든 것은 블록' 아키텍처

현대의 생산성 도구인 노션은 조합성의 개념을 한 단계 더 발전시켰다. 노션의 핵심 철학은 창립자 이반 자오(Ivan Zhao)가 설명했듯, 사용자가 자신만의 소프트웨어를 만들 수 있는 '레고 블록'을 제공하는 것이다.

- 핵심 원시 기능: '블록(Block)'. 유닉스의 텍스트 한 줄이나 비지칼크의 셀과 달리, 노션의 블록은 단락, 이미지, 데이터베이스, 코드 조각 등 그 자체로 풍부한 내용을 담을 수 있는 자율적인 객체다.
- 조합: 사용자는 이러한 블록들을 배열하고, 중첩하고, 연결함으로써 간단한 문서에서부터 위키, 프로젝트 관리 보드, 간단한 CRM에 이르기까지 다양한 결과물을

조합해낸다.

- 설계 여정**:** 흥미롭게도, 자오의 초기 비전, 즉 비프로그래머를 위한 앱 제작 도구는 너무 추상적이어서 실패했다. 그는 이 아이디어를 '브로콜리'에 비유했다. 그는 이 강력하고 조합 가능한 핵심을 사람들이 이미 친숙하게 느끼는 노트 및 문서 앱이라는 '설탕 코팅'으로 감쌈으로써 성공을 거두었다. 사용자는 간단한 노트 작성을 위해 노션을 시작했다가, 점차 그 안에 숨겨진 '레고 블록'의 힘을 발견하고 자신만의 워크플로우를 구축하게 된다.

노션의 사례는 조합 가능한 단위(composable unit)의 진화를 보여준다. 유닉스의 조합 단위가 텍스트 한 줄이었다면, 비지칼크는 셀, 웹은 하이퍼링크/URL이었다. 노션의 조합 단위는 블록이다. 이는 조합이라는 핵심 원칙은 변하지 않지만, 그 원시 기능 자체는 시간이 지남에 따라 더욱 풍부해지고 추상화될 수 있음을 보여준다. 아키텍트의 과제는 특정 도메인에 가장 강력하고 유연한 '원시 기능'이 무엇인지 식별하는 것이다. 노션의 성공은 올바른 원시 기능이란 간단한 사용 사례(노트 필기)에는 충분히 단순하면서도, 복잡한 사용 사례(CRM 구축)에는 충분히 강력한 것이어야 함을 시사한다.

3.3 생각의 웹**:** 롬 리서치**(Roam Research)**와 양방향 링크

롬 리서치는 노트 필기 앱 시장에 파괴적인 혁신을 가져왔다. 그 혁신의 중심에는 단 하나의, 그러나 강력한 직교적 기능이 있다.

- 핵심 원시 기능**:** 양방향 링크(Bi-directional link). 전통적인 하이퍼링크가 단방향(mono-directional)인 것과 달리, 롬의 모든 링크는 양방향 연결을 생성한다. 즉, A 페이지에서 B 페이지로 링크를 걸면, B 페이지는 A 페이지가 자신을 링크하고 있음을 자동으로 인지하고 표시해준다.
- 직교성**:** 양방향 링크 메커니즘은 단순한 아웃라이너(outliner) 위에 계층화된 직교적 기능이다. 글머리 기호로 텍스트를 작성하는 핵심 기능과 페이지를 연결하는 기능은 서로 독립적이지만 강력하게 상호작용한다.
- 조합 및 창발**:** 이 단 하나의 기능이 사용자가 자신의 지식 기반과 상호작용하는 방식을 근본적으로 변화시킨다. 정보의 구조는 경직되고 하향식인 계층 구조(폴더)에서 분산되고 상향식인 그래프 구조('네트워크화된 생각')로 전환된다. 이는 사용자가 처음에는 의도하지 않았던 아이디어 간의 연결을 창발적으로 발견하게 해준다. 코너 화이트-설리번(Conor White-Sullivan) 창립자는 이를 "생각에 대한 복리(compound interest on your thoughts)"라고 표현했다. 이 방식은 노트가 폴더 구조 속에서 '버려지거나' 잊히는 문제를 해결한다.

롬 리서치의 사례는 단 하나의 직교적 기능이 어떻게 시스템의 핵심 은유를 완전히 바꿀 수 있는지를 보여주는 강력한 교훈이다. 대부분의 노트 앱은 더 많은 기능을 추가하는 방식으로 경쟁한다(더 풍부한 텍스트 서식, 더 많은 첨부 파일 유형 등). 롬은 개념적으로 다른 단 하나의 직교적 기능을 추가했다. 이 기능은 제품의 은유를 '디지털 파일 캐비닛'에서 '네트워크화된 생각을 위한 도구'로 바꾸기에 충분했다. 이는 엄청난 지렛대 효과를 보여준다. 10개의 점진적인 기능을 추가하는 대신, 기존 구성 요소들이 조합되고 인식되는 방식을 근본적으로 바꿀 수 있는 하나의 직교적 기능을 찾는 것이 훨씬 더 강력할 수 있다.

3.4 협업 캔버스**:** 피그마**(Figma)**의 멀티플레이어 아키텍처

디자인 도구인 피그마는 실시간 협업 기능을 통해 업계의 표준을 재정의했다. 이 '멀티플레이어' 경험의 핵심에는 사용자 행동의 조합이라는 아키텍처가 있다.

- 핵심 원시 기능**:** moveObject, changeColor와 같은 사용자의 모든 행동은 개별적이고 원자적인 작업(operation)으로 취급된다.
- 직교성 및 조합**:** 피그마의 실시간 협업은 단일 기능이 아니라, 여러 사용자의 행동을

조합하기 위한 아키텍처다. OT(Operational Transformation)나 CRDT(Conflict-free Replicated Data Types)와 같은 기술은 여러 사용자의 동시적인 작업들이 직교적으로 적용될 수 있도록 보장하는 데 사용된다. 즉, 사용자 A가 객체의 색상을 변경하는 작업이 사용자 B가 동시에 같은 객체를 이동시키는 작업을 방해하거나 덮어쓰지 않는다. 문서의 최종 상태는 모든 사용자의 작업 로그를 일관되게 조합한 창발적 결과물이다.

- 창발**:** 이 아키텍처는 마치 마법처럼 느껴지는 유연한 '멀티플레이어' 디자인 세션을 가능하게 한다. 이 복잡한 사회적 상호작용은 작고 독립적인 변경 사항들의 단순하고 신뢰성 있는 조합 위에 구축된다.

피그마의 아키텍처는 조합성의 원리가 정적인 코드 모듈뿐만 아니라 동적인 사용자 이벤트 스트림에도 적용될 수 있음을 보여준다. 사용자 상호작용을 작고 직교적인 작업들의 스트림으로 분해함으로써, 강력한 실시간 협업 시스템을 구축할 수 있다. 이 관점에서 '문서'는 더 이상 저장하고 불러오는 파일이 아니라, 이벤트 스트림을 일관되고 재현 가능하게 '접은(fold)' 결과물이다. 이는 모든 생산성 소프트웨어의 미래에 중대한 시사점을 던진다.

3.5 시각적 로직 엔진**:** 언리얼 엔진**(Unreal Engine)**의 블루프린트

세계 최고의 게임 엔진 중 하나인 언리얼 엔진의 블루프린트(Blueprint) 비주얼 스크립팅 시스템은 프로그래머와 비프로그래머(아티스트, 디자이너) 간의 협업을 위해 직교성과 조합성을 영리하게 활용한 사례다.

- 핵심 원시 기능**:** 노드(Node). 블루프린트 시스템의 각 노드는 개별적인 함수, 이벤트, 또는 변수를 시각적으로 표현한다. 종종 이 노드들은 기본 게임 엔진의 강력한 C++ 함수에 대한 직접적인 시각적 대리인 역할을 한다.
- 직교성 및 조합**:** 블루프린트는 비프로그래머에게 힘을 실어주기 위해 설계되었다. 이 시스템은 복잡한 C++ 기능을 직교적으로 만듦으로써 이를 달성한다. 프로그래머는 명확한 입력과 출력을 가진 C++ 컴포넌트를 만들 수 있고, 이 컴포넌트는 디자이너가 사용할 수 있는 간단한 노드로 나타난다. 디자이너는 이 노드들을 다른 노드들과 시각적으로 '조합'하여 문을 열거나, 적의 인공지능(AI) 행동을 정의하는 등 복잡한 게임 로직을 C++ 코드를 전혀 건드리지 않고도 생성할 수 있다.
- 창발**:** 이 방식은 신속한 프로토타이핑과 반복 작업을 가능하게 한다. 디자이너는 독립적으로 복잡한 게임플레이 메커니즘을 만들고 테스트할 수 있으며, 이는 더 창의적이고 완성도 높은 게임으로 이어진다. 이 시스템은 프로그래머가 기초적인 '레고 블록'(C++ 컴포넌트)을 만들고, 디자이너가 최종적인 '성'(블루프린트의 게임 로직)을 짓는 협업 워크플로우를 촉진한다.

블루프린트의 사례는 직교성이 어떻게 서로 다른 기술을 가진 집단 간의 협업 장벽을 허물 수 있는지를 보여준다. 이는 유닉스 철학의 도메인 특화된 구현이다. C++ 함수들은 "한 가지 일을 잘하는 도구"이고, 블루프린트 그래프는 디자이너라는 다른 유형의 사용자가 "함께 작동하는 프로그램(게임 로직)을 작성"할 수 있게 해주는 "파이프라인"이다. 이는 직교성이 서로 다른 기술 집합을 가진 그룹 간의 협업을 촉진하는 도구를 설계하는 핵심 전략임을 시사한다.

제**4**장**:** 아키텍트의 사고방식 **–** 창발을 위한 설계 실용 가이드

앞선 장들에서 탐구한 역사적 기원과 구체적인 사례 연구들은 하나의 결론으로 수렴한다. 조합 가능하고 직교적이며 최소한의 기능을 가진 시스템을 설계하는 것은 특정 기술이나 프레임워크를 따르는 것이 아니라, 특정 '사고방식'을 채택하는 것이다. 이 장에서는 앞서 논의된 통찰들을 개발자와 아키텍트가 실제 업무에 적용할 수 있는 실용적이고 실행 가능한

프레임워크로 종합하고자 한다.

핵심 소프트웨어 공학 원칙과의 연결

우리가 탐구한 세 가지 원칙은 완전히 새로운 개념이 아니다. 오히려, 수십 년간 검증된 소프트웨어 공학의 핵심 원칙들을 가장 본질적인 형태로 증류한 것이다.

- 높은 응집도와 낮은 결합도 **(High Cohesion & Loose Coupling):** "한 가지 일을 잘하라"는 유닉스 철학은 '높은 응집도' 원칙의 직접적인 적용이다. 각 모듈은 단일하고 명확한 책임을 갖는다. "보편적인 인터페이스"를 통해 프로그램을 연결하는 것은 '낮은 결합도'를 강제하는 전략이다. 각 모듈은 다른 모듈의 내부 구현에 의존하지 않고, 오직 표준화된 인터페이스를 통해서만 상호작용한다. 결국, 직교성은 이 두 원칙이 성공적으로 결합되었을 때 나타나는 바람직한 속성이다.
- **SOLID** 원칙**:** 객체 지향 설계의 다섯 가지 기본 원칙인 SOLID 역시 이 철학과 깊이 연결된다.
    - 단일 책임 원칙 **(Single Responsibility Principle, SRP):** 이는 유닉스 철학의 핵심 교리와 거의 동일하다. 하나의 클래스나 모듈은 변경되어야 할 단 하나의 이유만을 가져야 한다.
    - 인터페이스 분리 원칙 **(Interface Segregation Principle, ISP)** 및 의존성 역전 원칙 **(Dependency Inversion Principle, DIP):** 이 두 원칙은 조합성에 필요한 낮은 결합도를 달성하기 위한 구체적인 전략들이다. 클라이언트는 자신이 사용하지 않는 인터페이스에 의존해서는 안 되며(ISP), 상위 수준 모듈은 하위 수준 모듈에 의존해서는 안 된다. 둘 다 추상화에 의존해야 한다(DIP). 이는 컴포넌트들이 구체적인 구현이 아닌, 안정적인 '보편적 인터페이스'에 의존하게 만들어 교체와 재조합을 용이하게 한다.
- **DRY, KISS, YAGNI:** 이 세 가지 약어는 최소주의를 실천하는 개발자의 만트라다.
    - 반복하지 말라 **(Don't Repeat Yourself, DRY):** 중복을 최소화하라는 이 원칙은 자연스럽게 재사용 가능한 컴포넌트나 함수의 생성을 장려한다.
    - 단순하게 유지하라 **(Keep It Simple, Stupid, KISS):** 이는 최소주의의 정수다. 불필요한 복잡성을 피하고 가장 간단한 해결책을 찾으려는 노력은 직교적이고 조합 가능한 시스템의 기반이 된다.
    - 그것은 필요 없을 것이다 **(You Aren't Gonna Need It, YAGNI):** 이 원칙은 미래에 필요할 것이라는 '추측'에 기반하여 기능을 추가하는 과잉 엔지니어링을 방지한다. 불필요한 기능과 의존성을 추가하는 것은 시스템의 직교성을 파괴하고 조합을 어렵게 만드는 주된 원인이다.

직교적 설계를 위한 프레임워크**:** 아키텍트의 질문 목록

이론을 실천으로 옮기기 위해, 새로운 시스템이나 기능을 설계할 때 스스로에게 던져볼 수 있는 발견적 질문 목록을 제시한다. 이 질문들은 앞선 사례 연구에서 도출된 교훈들을 바탕으로 구성되었다.

책임에 대하여 **(**직교성**)**

- "이 컴포넌트의 분리할 수 없는 단일 책임은 무엇인가**?** 한 문장으로 간단하게 설명할 수 있는가**?"** (참고: 유닉스 철학)
- "이 컴포넌트를 변경하면 시스템의 관련 없는 다른 부분들도 변경해야 하는가**?** 만약 그렇다면, 결합은 어디에서 발생하고 있는가**?"** (참고: 낮은 결합도)
- "이 기능의 상태(데이터)와 행위(로직)는 명확하게 분리되어 있는가**?** 사용자가

프로그래밍 없이 상태를 조작하여 새로운 행위를 조합할 수 있는가?" (참고: 비지칼크)

연결에 대하여 **(**조합성**)**

- "이 컴포넌트가 노출할 수 있는 가장 단순하고 보편적인 '인터페이스'는 무엇인가? 텍스트 스트림이나 표준 REST API처럼 단순할 수 있는가?" (참고: 유닉스 파이프)
- "이 컴포넌트는 '누가' 자신을 사용할지 알고 있는가, 아니면 단순히 입력을 받아 출력을 생성할 뿐인가? 어떻게 하면 이 컴포넌트를 더욱 사용 맥락에 무관하게(agnostic) 만들 수 있는가?" (참고: 파이프와 필터)
- "기존 컴포넌트들을 근본적으로 다른 방식으로 조합하게 만들 수 있는, 단 하나의 새로운 직교적 기능을 추가할 수 있는가?" (참고: 롬 리서치)

범위에 대하여 **(**최소주의**)**

- "가치를 전달하면서도 작동할 수 있는, 이 기능의 가장 간단한 버전은 무엇인가?" (참고: 워드 커닝햄)
- "우리는 지금 당장 필요하기 때문에 이 복잡성을 추가하는가, 아니면 나중에 필요할지도 모른다고 '생각'하기 때문에 추가하는가? (YAGNI)" (참고: YAGNI 원칙)
- "사용자의 99%를 신뢰한다고 가정하면, 어떤 복잡한 기능(예: 권한 관리, 유효성 검사)을 제거하거나 나중으로 미룰 수 있는가?" (참고: 위키)

점진적 아키텍처

이러한 사고방식은 자연스럽게 '점진적 아키텍처'라는 전략으로 이어진다. 이는 처음부터 모든 것을 완벽하게 계획하는 '빅 디자인 업 프론트(Big Design Up Front)' 방식과 대조된다. 대신, 최소한의 직교적 핵심 기능으로 시작하여 시스템을 외부로 확장해 나가는 방식이다.
워드 커닝햄의 위키가 이 접근법의 완벽한 예시다. 그는 편집과 링크라는 절대적인 최소 기능으로 시작했다. 사용자 관리, 권한, 복잡한 마크업과 같은 기능들은 시스템이 성장하고 그 필요성이 부인할 수 없는 현실로 증명되었을 때에만 점진적으로 추가되었다. 이는 "계획을 따르기보다 변화에 대응하는 것"을 더 가치 있게 여기는 애자일 선언문의 정신과 정확히 일치한다.
점진적 아키텍처의 목표는 처음부터 경직되게 계획된 시스템이 아니라, 유기적으로 진화할 수 있는 시스템을 설계하는 것이다. 이를 위해서는 초기 설계 단계에서부터 조합성과 직교성을 염두에 두어야 한다. 잘 정의된 인터페이스를 통해 연결된 작고 독립적인 컴포넌트들은 미래에 새로운 컴포넌트를 추가하거나 기존 컴포넌트를 교체하기 쉽게 만들어, 시스템이 예상치 못한 요구사항 변화에 우아하게 적응할 수 있도록 한다.

결론**:** 더 나은 블록으로 짓기

이 보고서는 조합성, 직교성, 최소주의라는 세 가지 원칙이 어떻게 소프트웨어 설계의 근간을 이루는지 탐구하는 여정이었다. 유닉스의 커맨드 라인에서 시작하여 워드 커닝햄의 위키를 거쳐 현대의 다양한 애플리케이션에 이르기까지, 우리는 이 원칙들이 특정 기술이나 시대에 국한된 것이 아니라, 우아하고 지속 가능한 시스템을 만들기 위한 시대를 초월한 사고방식임을 확인했다.
이는 단순함을 향한 끊임없는 추구, 단일 책임에 대한 엄격한 집중, 그리고 구성 요소들이 어떻게 연결될 수 있는지에 대한 창의적인 비전에 관한 것이다. 시스템의 진정한 힘은 개별 기능의 총합이 아니라, 그 기능들이 상호작용하며 만들어내는 창발적 가능성의 공간에서

나온다.
소프트웨어의 복잡성이 인공지능, 분산 시스템 등으로 인해 기하급수적으로 증가하는 오늘날, 시스템을 단순하고 직교적이며 조합 가능한 원시 기능들로 분해하는 이 원칙들은 그 어느 때보다 중요하다. 혁신의 미래는 더 크고 복잡한 단일 애플리케이션을 만드는 데 있는 것이 아니라, 더 좋고 강력한 '레고 블록'을 설계하고 사용자들이 스스로 해결책을 구축할 수 있도록 힘을 실어주는 데 있다.
궁극적인 목표는 인간의 창의성과 지성을 증폭시키는 도구를 만드는 것이다. 아키텍트가 처음에는 상상조차 하지 못했던 해결책들이 사용자들의 손에서 창발적으로 탄생할 수 있는 시스템, 그것이 바로 이 철학이 지향하는 바다. 이 보고서가 독자 여러분이 그러한 '더 나은 블록'을 만들고, 그 블록들로 미래를 짓는 데 영감을 주는 작은 디딤돌이 되기를 바란다.

참고 자료

1. Unix philosophy - Wikipedia, https://en.wikipedia.org/wiki/Unix_philosophy 2. Orthogonality in Software - by Hilal Koçak - Medium, https://medium.com/@hhilalkocak/orthogonality-in-software-c2fd6f73af81 3. en.wikipedia.org, https://en.wikipedia.org/wiki/Orthogonality_(programming)#:~:text=In%20computer%20programming%2C%20orthogonality%20means,sets%2C%20as%20orthogonal%20instruction%20set. 4. Orthogonality (programming) - Wikipedia, https://en.wikipedia.org/wiki/Orthogonality_(programming) 5. Orthogonality in Programming and Software Engineering | by Jose Tello V. | AI monks.io, https://medium.com/aimonks/orthogonality-in-programming-and-software-engineering-4991366f8a91 6. Orthogonality in Computer Programming | Baeldung on Computer Science, https://www.baeldung.com/cs/orthogonality-cs-programming-languages-software-databases 7. Orthogonality in Software Engineering - freeCodeCamp, https://www.freecodecamp.org/news/orthogonality-in-software-engineering/ 8. Orthogonality in software engineering explained in five minutes, https://adriantostega.wordpress.com/2019/02/03/orthogonality-in-software-engineering-explained-in-five-minutes/ 9. Composability in Software Development: A Deep Dive - CodeStringers, https://www.codestringers.com/insights/composability-in-software-development/ 10. Design Principles for Composable Architectures | by Ashan Fernando - Bits and Pieces, https://blog.bitsrc.io/design-principles-for-composable-architectures-2a8dcfb11998 11. Understanding composability: Definitions and explanations - Contentful, https://www.contentful.com/blog/what-is-composability/ 12. Composable Architectures, The Next Evolution of Software Design | Bytex Technologies, https://bytex.net/blog/composable-architectures-the-next-evolution-of-software-design/ 13. What is Composability: Increase Agility and Accelerate Application Development - Ionic.IO, https://ionic.io/resources/articles/what-is-composability 14. What is composability? - Mulesoft, https://www.mulesoft.com/integration/what-is-composability 15. The Case for Minimalism in Software Architecture, https://www.neverletdown.net/2014/11/the-case-for-minimalism-in-software-architecture.html 16. Designing orthogonal software systems - Software Architect's Handbook [Book], https://www.oreilly.com/library/view/software-architects-handbook/9781788624060/3291dbd2-72f4-40da-b3f9-662f88fb5bb5.xhtml 17. UNIX philosophy - ruivieira.dev, https://ruivieira.dev/unix-philosophy.html 18. Basics of the Unix Philosophy, https://cscie2x.dce.harvard.edu/hw/ch01s06.html 19. Understanding the Unix Philosophy - Miika Nissi, https://miikanissi.com/blog/understanding-unix-philosophy/ 20. what is the "unix philosophy"? - Reddit,

https://www.reddit.com/r/unix/comments/11wgy5k/what_is_the_unix_philosophy/ 21. PROGRAMMING - USENIX, https://www.usenix.org/system/files/login/articles/login_spring16_05_mcilroy.pdf 22. Pipeline (Unix) - Wikipedia, https://en.wikipedia.org/wiki/Pipeline_(Unix) 23. Introduction to Pipes in UNIX systems - Logdy, https://logdy.dev/blog/post/introduction-to-pipes-in-unix-systems 24. Pipes and Filters - Neward & Associates, https://blogs.newardassociates.com/patterns/behavioral/PipesAndFilters/ 25. Streamlining Data: Pipes and Filters in Unix | by Akshaya's LOGICVERSE | Medium, https://medium.com/@akshayatechcontent/streamlining-data-pipes-and-filters-in-unix-6bebd7f95e07 26. On Unix Philosophy - Lack of Imagination, https://lackofimagination.org/2024/01/on-unix-philosophy/ 27. What Is Composable Architecture? A Concise Guide - Boomi, https://boomi.com/blog/concise-guide-to-composability/ 28. WikiWikiWeb - Wikipedia, https://en.wikipedia.org/wiki/WikiWikiWeb 29. History of wikis - Wikipedia, https://en.wikipedia.org/wiki/History_of_wikis 30. Ward Cunningham on Wikis, Patterns, Mashups and More -- ADTmag, https://adtmag.com/articles/2006/08/14/ward-cunningham-on-wikis-patterns-mashups-and-more.aspx 31. Wiki - Wikipedia, https://en.wikipedia.org/wiki/Wiki 32. WikiWikiWeb to Wikipedia. WikiWikiWeb is the first ever wiki or… | by Tilak Lodha - Medium, https://medium.com/@tilak.l/wikiwikiweb-to-wikipedia-fcc537a5de9a 33. The Simplest Thing that Could Possibly Work - artima, https://www.artima.com/articles/the-simplest-thing-that-could-possibly-work 34. Ward Cunningham on Simplicity - Adventures, http://dvt.name/2010/07/27/ward-cunningham-on-simplicity/ 35. Wiki Design Principles, https://wiki.c2.com/?WikiDesignPrinciples 36. Design Principles of Wiki: How can so little do so much?, http://c2.com/doc/wikisym/WikiSym2006.pdf 37. VisiCalc - Wikipedia, https://en.wikipedia.org/wiki/VisiCalc 38. A vision of computing's future - Harvard Gazette, https://news.harvard.edu/gazette/story/2012/03/a-vision-of-the-computing-future/ 39. VisiCalc: The Spreadsheet That Started It All - Making Data Meaningful, https://makingdatameaningful.com/visicalc/ 40. Implementing VisiCalc - Bob Frankston, http://rmf.vc/ImplementingVisiCalc 41. Ivan Zhao on Notion's “Lost Years,” Surviving Near Collapse, and ..., https://www.thewantrepreneurshow.com/blog/ivan-zhao-on-notions-lost-years-surviving-near-collapse-and-building-for-creativity/ 42. The philosophy behind Notion: Make a blank paper | by odeson - Medium, https://medium.com/@odeson/the-philosophy-behind-notion-make-a-blank-paper-e6c55eca8344 43. How to create a design system that works - Notion, https://www.notion.so/blog/how-to-create-a-design-system 44. Best Design System Templates from Notion | Notion Marketplace, https://www.notion.com/templates/category/design-system 45. Project Management Templates for Every Team | Notion Marketplace, https://www.notion.com/templates/category/projects 46. A Thorough Beginner's Guide to Roam Research - The Sweet Setup, https://thesweetsetup.com/a-thorough-beginners-guide-to-roam-research/ 47. An introduction to Roam Research - elaptics.co.uk, https://elaptics.co.uk/journal/introduction-roam-research/ 48. How to use Roam Research: a tool for metacognition - Ness Labs, https://nesslabs.com/roam-research 49. Roam Research – A note taking tool for networked thought., https://roamresearch.com/ 50. Getting compound interest on your thoughts with Conor White-Sullivan, https://nesslabs.com/conor-white-sullivan-interview 51. Creating Tools For Networked Thought with Roam Research | Conor White-Sullivan on Venture Stories, Hosted by

Erik Torenberg - Podcast Notes, https://podcastnotes.org/venture-stories/conor-white-sullivan/ 52. How Figma Achieved Seamless Real-Time Multi-user Collaboration ..., https://medium.com/frontend-simplified/deconstructing-the-magic-how-figma-achieved-seamless-real-time-multi-user-collaboration-37347f2ee292 53. You might not need a CRDT - Jamsocket, https://jamsocket.com/blog/you-might-not-need-a-crdt 54. How Figma's Multiplayer Technology Works? - Peerlist, https://peerlist.io/omjogani/articles/figmas-multiplayer-tech-summary 55. Enhancing Collaboration in UI/UX Design: Best Practices for Using Figma Team Features, https://www.kaarwan.com/blog/ui-ux-design/collaboration-ui-ux-design-best-practices-using-figma-team-features?id=882 56. Blueprint For Artists | Unreal Engine - YouTube, https://www.youtube.com/watch?v=c6Hc_RBA7ww 57. Visual Scripting - Inspired by game engines-Noodl, https://www.noodl.net/post/visual-scripting-how-noodl-was-inspired-by-the-world-of-game-engines 58. Blueprints Visual Scripting in Unreal Engine - Epic Games Developers, https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine 59. Blueprint for artists - Unreal Engine, https://www.unrealengine.com/en-US/blog/blueprint-for-artists 60. Best practices for project architecture when using C++ and Blueprints, https://forums.unrealengine.com/t/best-practices-for-project-architecture-when-using-c-and-blueprints/20624 61. Blueprints vs. C++ in Unreal Engine - Codefinity, https://codefinity.com/blog/Blueprints-vs.-C-plus-plus-in-Unreal-Engine 62. Brand new to Unreal Engine. Not a fan of visual scripting - Blueprint, https://forums.unrealengine.com/t/brand-new-to-unreal-engine-not-a-fan-of-visual-scripting/136314 63. Unity Visual Scripting, https://unity.com/features/unity-visual-scripting 64. Networking Imagination: Ward Cunningham, https://www.wirfs-brock.com/jordan/PP/cunningham.html 65. Agile software development - Wikipedia, https://en.wikipedia.org/wiki/Agile_software_development 66. Ivan Zhao, Co-founder and CEO of Notion: The Visionary Behind Notion's Success, https://cordmagazine.com/business/entrepreneurship/ivan-zhao-co-founder-and-ceo-of-notion-the-visionary-behind-notions-success/