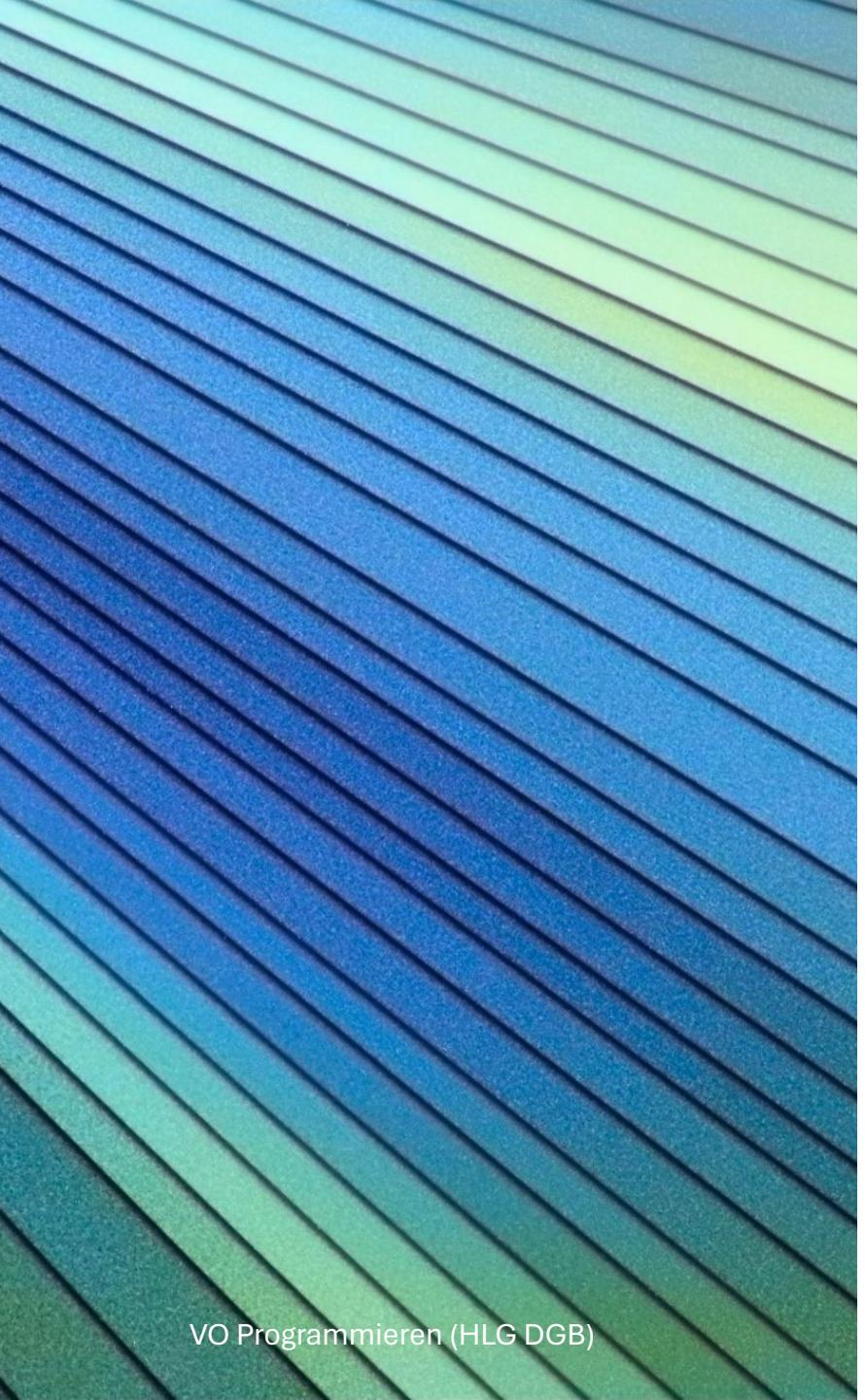


# Lerntheorie nach Lister



# Piaget's genetische Epistemologie

## Epistemologie

- Erkenntnistheorie
- Studium des menschlichen Wissens / Erkenntnisgewinnungsprozesses / Lernens

## Genese

- Das Entstehen von etwas
- im Sinne von Ursprung / Entwicklung

# Assimilation und Akkommodation

- Assimilation (Verallgemeinerung)
  - Neue Erfahrungen werden durch Verallgemeinerung bereits gemachter Erfahrungen in bestehende Strukturen eingepasst.
  - Neues Wissen wird generiert, indem man an bestehende Strukturen (Vorwissen) im Gedächtnis andockt.
  - Beispiel: Lernende haben verstanden, wie While-Schleifen funktionieren. Der Weg zum Verständnis eines weiteren Schleifentyps ist über Assimilation möglich.
- Akkommodation (Erweiterung)
  - Aufwändiger als Assimilation
  - Neues Wissen generieren, für das es noch keine Anknüpfungspunkte im Gedächtnis gibt.
  - Erfahrung kann mit bestehenden Schemata nicht erklärt / überwunden werden → bestehende Schemata müssen angepasst / erweitert werden
  - Bauen von neuen strukturierenden Elementen / neuen Schemata
  - Gleichbedeutend mit dem Aufbau und der Verstärkung von neuen Verbindungen zwischen Nervenzellen (Neurodidaktik).
  - Beispiel: Lernende wissen noch nichts von Schleifen und müssen für ein Verständnis der Funktionsweise von Schleifen (insb. incl. des Zusammenwirkens der konstitutiven Teile) völlig neue Strukturen aufbauen.

# Organisation, Adaption, Äquilibrium

- Organisation und Schemakonstruktion
  - Teilelemente / Objekte / Ereignisse werden kategorisiert und in einem zusammenhängenden System (Schema) miteinander in einen logischen Zusammenhang, in Verbindung gebracht (Organisation).
  - Schemata: Muster für Wissen und Verhalten
  - Erlernte Schemata erzeugen bei ihrer Anwendung in passenden Problemsituationen praktisch keinen kognitiven Aufwand mehr
- Adaption
  - Laufende Anpassung der Schemata durch Assimilation und Akkomodation
  - Assimilation und Akkomodation laufen parallel
- Äquilibrium
  - Streben nach Gleichgewicht zwischen Assimilation und Akkomodation
  - Stabilität der Schemata / des Verständnisses wird erhöht
  - Persönlich wahrgenommene Konsistenz der Schemata wird immer höher
  - Schema ist gelernt

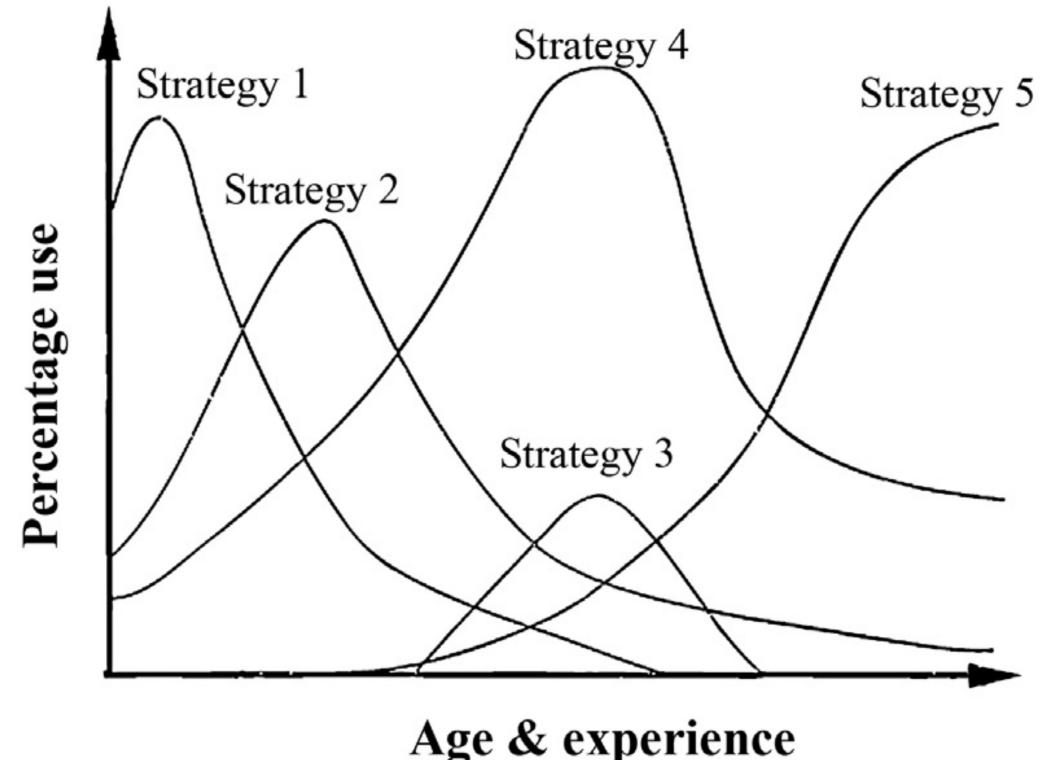
# Neo-Piaget'sche Stufentheorie

---

- Neue Fähigkeiten und Konzepte werden über Akkommodation und Assimilation gelernt
- Lernende durchlaufen unabhängig ihres Alters und unabhängig vom zu lernenden Inhalt im Rahmen des Lernprozesses 4 Stufen.
- Zu einem bestimmten Zeitpunkt, haben Kinder immer verschiedene Sichtweisen bzw. verschiedene kognitive Denkweisen auf Phänomene / Konzepte / Inhalte etc.
- Diese Sichtweisen auf Phänomene / dieses variantenreiche Denken über Phänomene / Konzepte / Inhalte führt über längere Zeit zu einem inneren Konflikt. Die konkurrierenden Sichtweisen existieren parallel.

# Neo-Piaget'sche Stufentheorie und Overlapping Waves Model

Kognitive Entwicklung bedeutet dann, dass ein gradueller Wechsel in der Häufigkeit der Anwendung der entsprechend konkurrierenden Sichtweise auftritt. Eine alte Sicht auf ein Phänomen wird zugunsten einer neuen Sicht (die etwa auf Basis neuer Erkenntnisse zustande kommt) langsam weniger häufig genutzt. Das ist der Entwicklungsprozess.



# Developmental Epistemology of Computer Programming (Raymond Lister und Donna M. Teague)

- Lister und Teague wenden die Stufentheorie nach Piaget auf die Computerprogrammierung im Unterricht an.  
[\(\[https://www.academia.edu/110073788/Toward\\\_a\\\_Developmental\\\_Epistemology\\\_of\\\_Computer\\\_Programming\]\(https://www.academia.edu/110073788/Toward\_a\_Developmental\_Epistemology\_of\_Computer\_Programming\)\)](https://www.academia.edu/110073788/Toward_a_Developmental_Epistemology_of_Computer_Programming)
- Sie beschreiben in ihrer Theorie Eigenschaften von Lernenden, die sich in den einzelnen Entwicklungsstufen und damit in den entsprechenden Phasen des Lernprozesses für einen neuen Inhalt befinden.
- Sie beschreiben damit auch einen Weg, den Lernende beim Lernen dieser Inhalte beschreiten.

# Developmental Epistemology of Computer Programming (Raymond Lister und Donna M. Teague)

Stufe	Beobachtungen (Lister)
Pre-Tracing-Phase	<ul style="list-style-type: none"><li>• Zeilenweise Codeerklärungen zu &lt; 50 % ok</li><li>• Vorherrschende Problemlösungsstrategie: Versuch und Irrtum</li></ul>
Tracing Phase 1	<ul style="list-style-type: none"><li>• Zeilenweise Codeerklärungen zu &gt; 50 % ok</li><li>• Zeilenweise Codeerklärungen ohne Abstraktion von den einzelnen Zeilen weg und damit ohne Bezug zur Semantik des gesamten Programmteils</li><li>• Keine Beziehungen zwischen den Codezeilen interpretierbar</li><li>• Ikonische Repräsentationen von Code (Diagramme) können nicht sinnbringend eingesetzt werden</li><li>• Vorherrschende Problemlösungsstrategie: Zufällige Codeänderungen und intensives Ausprobieren der Ergebnisse</li></ul>

Zusammenfassende Darstellung nach Nathan Bean: <https://textbooks.cs.ksu.edu/cis400/a-learning-programming/>

# Developmental Epistemology of Computer Programming (Raymond Lister und Donna M. Teague)

Phase	Beobachtungen
Tracing Phase 2	<ul style="list-style-type: none"><li>• Zeilenweise Codeerklärungen zu &gt; 50 % ok</li><li>• Zeilenweise Codeerklärungen ohne Abstraktion von den einzelnen Zeilen weg und damit ohne Bezug zur Semantik des gesamten Programmteils</li><li>• Keine Beziehungen zwischen den Codezeilen interpretierbar</li><li>• Ikonische Repräsentationen von Code (Diagramme) können nicht sinnbringend eingesetzt werden</li><li>• Vorherrschende Problemlösungsstrategie: Zufällige Codeänderungen und intensives Ausprobieren der Ergebnisse</li></ul>
Post Tracing Phase	<ul style="list-style-type: none"><li>• Hypothetisches und deduktives Denken ist ausgeprägt</li><li>• Deduktiv: vom Allgemeinen (=Schema / Konzept) auf das Konkrete (= kontextualisierte Anwendung des Schemas) schließen</li><li>• Code kann "gelesen" werden. Es geht nicht mehr um die Interpretation der einzelnen Zeilen und ihrer Beziehungen sondern um das Gesamtkonstrukt</li><li>• Schema (CLT) ist ausgebaut und wird getriggert</li></ul>

# Bezug zur CLT

- Lernende
  - durchlaufen beim Programmieren diese Phasen
  - bauen neues Wissen durch Schemakonstruktion auf
- Anfänger
  - muss Code zunächst zeilenweise interpretieren
  - dann einzelne Codezeilen zueinander in Beziehung bringen
  - dann im Gesamten ableiten, was der Code macht.
- Experte:
  - Code lesen durch Applikation von Schemata auf Teilstrukturen im Code,
  - ohne darüber nachdenken zu müssen,
  - was einzelne Zeilen bedeuten,
  - wie sie syntaktisch genau aufgebaut sind,
  - wie sie zueinander in Beziehung stehen.

# Bezug zur CLT

- Experte
  - Beim Codieren selbst hat der Experte bereits vor Beginn der Codierphase eine klare Vorstellung der betroffenen Schemata und implementiert diese zur Problemlösung direkt.
  - Er kennt die Syntax der Sprache.
  - Er kennt die Werkzeuge, die er dabei verwendet.
- Anfänger
  - müssen sich jede Codezeile sowohl syntaktisch als auch semantisch hart erarbeiten
  - müssen lernen, die Beziehungen zwischen den Zeilen korrekt herzustellen
  - müssen abschätzen, was ihr Code nun in Bezug auf die Lösung Problemstellung bedeutet
  - müssen darüber nachdenken, wie sie mit ihrem Code dann auch noch eine ganze Klasse von ähnlichen Problemstellungen lösen können (Abstraktion)
  - müssen sich außerdem mit Syntaxproblemen und Tooling-Problemen (Compiler, Kommandozeile, IDEs etc.) auseinandersetzen.

# Bezug zur CLT

- Für die Lösung von Programmierproblemstellungen ist ein hoher Grad an Elementinteraktivität nötig.
- Lösungen in der Programmierung bestehen aus zueinander passenden Anwendungen von vielen kleinen Lösungselementen (incl. deren Abstraktion in weiterer Folge).
- Das Arbeitsgedächtnis kann aber immer nur 3 bis 5 Lösungselemente gleichzeitig verarbeiten.
- Ziel muss es sein, diese Lösungselemente zusammengenommen als komplette Lösungsschemata im Langzeitgedächtnis zu verankern (= Schemakonstruktion).
- Entsprechend der CLT ist es wichtig, beim Aufbau der Schemata das Arbeitsgedächtnis nicht zu überlasten.

# Bezug zur CLT

- Ein Schema nach Sweller ist ein kognitives Konstrukt mithilfe dessen Informationen organisiert, strukturiert und (im Arbeitsgedächtnis) schnell verarbeitet werden können.
- Im Arbeitsgedächtnis wird ein automatisiertes Schema dabei als ein einziges Element betrachtet, selbst wenn das Schema eine Vielzahl von interagierenden Elementen betreffen kann.
- Durch die (automatisierte) Anwendung von Schemata können damit komplexe Herausforderungen mit relativ geringer Belastung des Arbeitsgedächtnisses gemeistert werden.

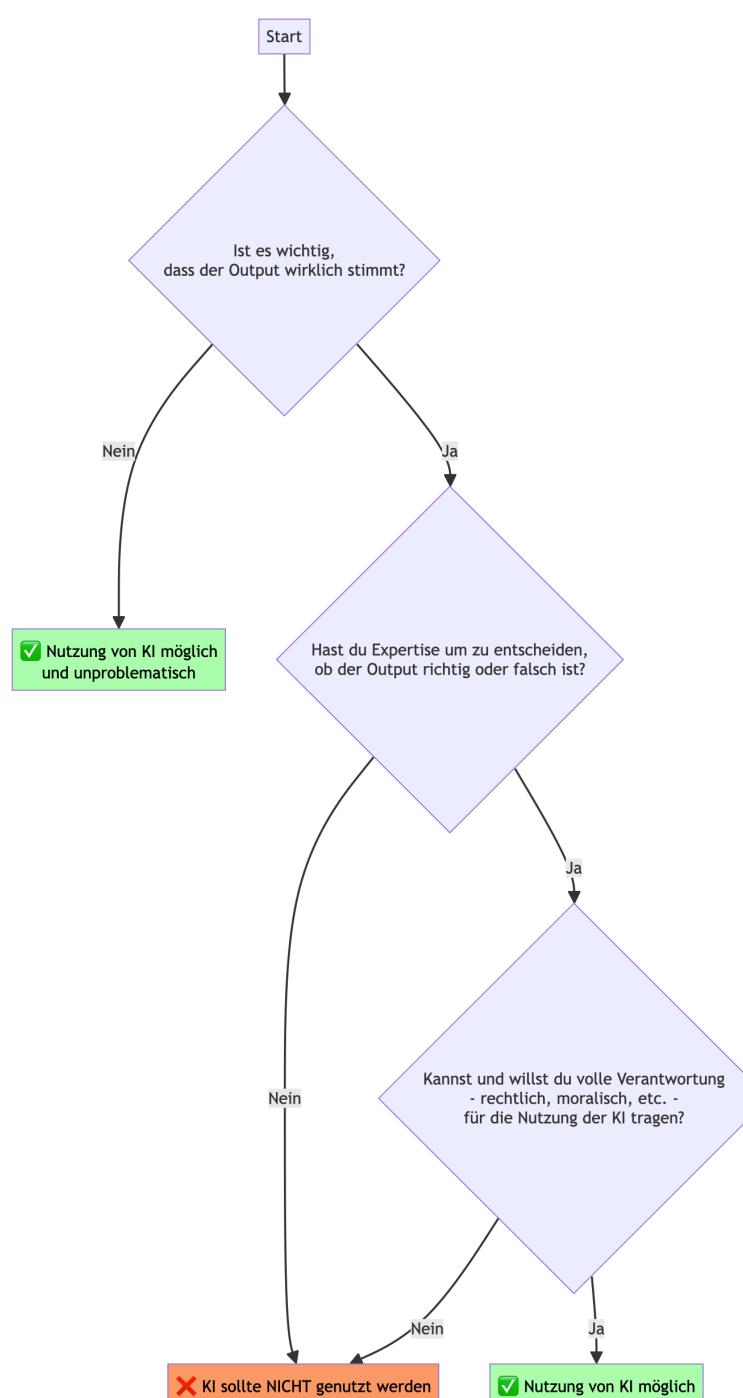
# Bezug zur CLT

- Ziel entsprechenden Unterrichts ist es, das Arbeitsgedächtnis beim Aufbau der Schemata nicht zu überbelasten.
- Dazu muss der extrinsische Cognitive Load (bedingt durch die Komplexität der Lernumgebung) sowie der intrinsische Cognitive Load (dem Lerngegenstand inhärenter CL) so weit reduziert werden, dass er vom Arbeitsgedächtnis bewältigt werden kann.
- Speziell im Bereich der Programmierung ist dabei Vorsicht geboten, weil der ICL aufgrund der hohen Elementinteraktivität der Inhalte ohne entsprechend ausgebildeter Schemata sehr schnell überhandnimmt.

# Chance: Generative KI im Codierprozess

- Generative KI kann den Codierprozess für einfache Probleme jetzt schon dramatisch vereinfachen.
  - Das bringt viele Vorteile auch für den Lernprozess in der Schule ...
  - Möglichkeit zur Konzentration auf zentrale Konzepte, auf Analyseprozesse, auf planende Tätigkeiten, auf kreative Tätigkeiten, auf Reviewprozesse etc. und weniger auf die Bedienung komplexer Werkzeuge wie Programmiersprachen, die für die Nutzung durch den Menschen nicht wirklich sehr gut gemacht sind ...
- Herausforderungen:
  - Wie argumentieren wir, dass wir trotzdem noch Programmierexperten brauchen?
  - Wie argumentieren wir, dass trotzdem jemand die Anstrengungen zum Erlernen der Konzepte (Akkommodation!) auf sich nehmen muss?
- Ansätze:
  - Ist Programmieren gleichzusetzen mit Codieren?
  - Wie artikulieren wir unsere Wünsche gegenüber der KI?
  - Wie können wir einschätzen, ob die KI „korrekt“ arbeitet?
  - Wie können wir Verantwortung übernehmen?

Quelle: Sabzalieva, E., & Valentini, A. (2023). ChatGPT and artificial intelligence in higher education: quick start guide



# Legitimation für Programmierung im DGB-Unterricht

# Warum sollen wir überhaupt noch Programmieren unterrichten?

- DGB-Curriculum
- Allgemeinbildung (Bussmann / Heymann)
- Entwicklungen rund um KI
- Türöffner

<https://github.com/clander/voprogrammieren/blob/main/Didaktik/Zentrale-Ideen.md>

# Das Erbe von Seymour Papert und Mitch Resnick

# Seymour Papert

- Seymour Papert (Professor für Mathematik und Erziehungswissenschaften am MIT) war Schüler von Piaget.
- Er hat Piaget's Ideen in die Informatikdidaktik getragen und unter dem Begriff "Konstruktionismus" weiterentwickelt:
  - Wissensrekonstruktion durch die Lernenden selbst
  - Herstellen (Konstruktion) von Lernproduktion (Artefakten) durch intensive Beschäftigung / Lernendenaktivität
  - Interesse an Produkten und Stolz auf Produkte als Motivation

# Papert steht für ...

- Situiertes Lernen
  - Problemorientierung in authentischen Situationen
  - verschiedene Kontexte
  - Artikulation und Reflexion zum Zwecke der individuelle Abstraktion
  - sozialer Kontext
- Frühes Eröffnen von **konkret operativen Zugangsmöglichkeiten** zu den neuen Technologien
- **Entdeckendes Lernen** mit Computerunterstützung
- Papert entwickelte die Programmiersprache LOGO (incl. Miniwelt-Konzept) und als geistiger Vater der LEGO Mindstorms und von Scratch.

# Handlungsorientierter Unterricht und die Unterrichtsprinzipien nach Piaget

operatives Prinzip (Lernen durch Handlungen an konkreten Objekten)

Integrationsprinzip (Lernen in Sinnzusammenhängen und Beziehungsnetzen, Lernen durch Wiederholen)

dynamisches Prinzip (Lernen durch Aktivität, problembasiertes Lernen, spielerisches Lernen)

Redundanzprinzip (Lernen durch Anknüpfung an Vorwissen und Interessen, Lernen in bekannten Kontexten, Lernen von Neuem mit bereits Bekanntem)

Stabilisierungsprinzip (Lernen mit genügend Zeit, Lernen durch wiederholen, Lernen durch Üben in neuen Kontexten)

Prinzip der Stufengemäßigkeit (Lernen mit Blick auf die Entwicklungsstufe)

# Entdeckendes Lernen

Dozentin S. erzählt ihrem Kollegen, dass sich ihre Studierenden für ein Projekt in ein Web-Applikations-Framework einarbeiten müssen. Frustriert berichtet sie, dass diese Mühe hätten, sich selbst mit dem Framework vertraut zu machen: "Ich hätte gedacht, die Studierenden könnten sich mit Hilfe von zwei Büchern und den vielen Materialien im Web selbst einarbeiten. Schließlich haben wir im Unterricht ein Framework verwendet, das von der Idee her ähnlich ist. Es fällt ihnen trotzdem ungeheuer schwer! Es gibt einfach zu viele Aspekte, als dass sie diese alle systematisch durchgehen könnten. Sie müssten sich also auf das Wesentliche konzentrieren. Aber damit haben sie echt Probleme. Ich hätte gedacht, dass sie nach meinem Unterricht dazu in der Lage sind!"

**Problem:** Informatikunterricht ist häufig geprägt durch das Vermitteln von Theorie mit anschließenden Übungen. Wichtige Aspekte wie selbstständiges Arbeiten, Kreativität und kritische Reflexion werden dabei zu wenig berücksichtigt. Im Berufsalltag kommtt aber der Fähigkeit, selbstständig neue Themen zu erarbeiten, eine zentrale Rolle zu.

**Lösung:** Die Fähigkeit des selbstständigen Arbeitens kann im Unterricht durch den Einsatz verschiedener Unterrichtsmethoden bewusst gefördert werden. Entdeckendes Lernen ist eine dieser Methoden und trägt zur Individualisierung des Unterrichts bei.

<http://www.swisseduc.ch/informatik-didaktik/informatikunterricht-planen-durchfuehren/4-methoden/entdeckendes-lernen.html>

# Charakteristika (Hartmann)

- Diese Art des Unterrichts soll selbstständiges Arbeiten, Kreativität und kritische Reflexion ermöglichen. Charakteristika sind:
- Offenes Thema
  - Erkunden, Hypothesen aufstellen, überprüfen, austauschen;
  - Lerngegenstand umfasst verschiedene Aspekte und verschiedene Entdeckungswege
  - Einfacher: Aspekt vorgegeben, aber verschiedene Lösungen möglich
  - Schwieriger: auch der zu explorierende Aspekt kann offen sein
- Vollständiges Material
  - Das notwendige Material wird zur Verfügung gestellt und ist vollständig bzw. selbsterklärend
  - Keine Betreuung durch den Lehrer nötig
  - Basierend auf dem vorgegebenen Material und dem eigenen Vorwissen sollen Dinge selbstständig entdeckt werden
  - Ziel ist die Entwicklung eigener Ideen, nicht das Herausfiltern und Strukturieren von Inhalten aus Sachbüchern oder Onlinetexten
- Aufgabe und Bewertung
  - Sollen so angelegt sein, dass verschiedene Lösungen, Herangehensweisen und Perspektiven möglich sind
  - Verschiedene Lösungen möglich
  - Bei Halbrichtigen Lösungen → Vorschläge ernst nehmen und weiterentwickeln, gesundes Maß an Freiheit

# Beispiele

- Experimente ohne Computer zu 13 Informatikthemen  
([https://www.swisseduc.ch/informatik/theoretische\\_informatik/paper\\_computer\\_science/](https://www.swisseduc.ch/informatik/theoretische_informatik/paper_computer_science/))
  - Paper Computer Science Experiment 1  
([https://www.swisseduc.ch/informatik/theoretische\\_informatik/paper\\_computer\\_science/docs/01\\_binaerzahlen.pdf](https://www.swisseduc.ch/informatik/theoretische_informatik/paper_computer_science/docs/01_binaerzahlen.pdf))
  - Paper Computer Science Experiment 3 – Nim Spiel  
([https://www.swisseduc.ch/informatik/theoretische\\_informatik/paper\\_computer\\_science/docs/03\\_nim\\_spiel.pdf](https://www.swisseduc.ch/informatik/theoretische_informatik/paper_computer_science/docs/03_nim_spiel.pdf))
- Behindertengerechte Webseiten (<http://www.swisseduc.ch/informatik-didaktik/informatikunterricht-planen-durchfuehren/4-methoden/docs/entdeckendes-lernen.pdf>)
- Schwierige Probleme der Informatik  
([https://www.swisseduc.ch/informatik/theoretische\\_informatik/hard\\_problems/index.html](https://www.swisseduc.ch/informatik/theoretische_informatik/hard_problems/index.html))
- Artikel mit weiteren Beispielen: <http://www.swisseduc.ch/informatik-didaktik/publikationen/docs/2005-02-15-infos-entdeckendes-lernen.pdf>

Kurs zum forschenden Lernen (TU München, Bundesministerium für Bildung und Forschung):

<https://clearinghouse-academy.de>

# Clearinghouse Academy

## **WAS STECKT DAHINTER?**

---

### **1. Basiskurs der Academy**

Vom wissenschaftlichen Experiment bis hin zur sachgemäßen Interpretation und Einordnung von Effekten führt dieser Kurs auf anschauliche Weise in Grundlagen und Hintergründe von empirischer Bildungsforschung ein. Auf diese Weise soll deutlich werden, welchen Beitrag Forschung für die Praxis in Unterricht und Schulentwicklung leisten kann.

### **2. Vertiefende Kurse der Academy**

#### Kurs zu Forschendem Lernen

Sie wollen mehr evidenzbasierte Informationen zur Methode des Forschenden Lernens haben, wie diese Methode lernwirksam eingesetzt werden kann, wie sie aufgebaut sein sollte und wie Sie diese vermitteln und einsetzen können? Dann sind Sie in diesem Kurs genau richtig.

#### Kurs zu evidenzbasierten Lehrstrategien

Hier finden Sie evidenzbasierte Informationen und Tipps und Tricks zu den Lehrstrategien Concept Maps, Selbsterklären, durch Schreiben lernen, Flipped Classroom, Kritischem Denken, Peer Assessment und zum Thema Hausaufgaben. Zertifiziert, wenn gewünscht, und alles OER!

#### Kurs zu Künstlicher Intelligenz in der Schule (neu)

Ein paar evidenzbasierte Informationen zum Thema KI, ChatGPT und Chatbots im Allgemeinen. Wie Prompt Engineering funktioniert, wie KI zur Unterrichtsunterstützung, als Lerngegenstand oder zur Lernunterstützung genutzt werden kann. Herausforderungen und Möglichkeiten ebenso wie mögliche Zukunftsszenarien werden dargelegt.

# Scratch

# Didaktische Aspekte von Scratch

Mitch Resnick on Seymour Papert:  
<https://www.youtube.com/watch?v=ZoczAscGYeQ>

- Scratch ist von Mitch Resnick im konstruktivistischen bzw. konstruktionistischen Geiste von Papert (dessen Schüler er war) als Lernumgebung konzipiert worden, die folgende didaktische Aspekte in den Vordergrund stellt:
- **Product:** Lernprodukte erzeugen, aktiv sein, Ideen entwickeln und umsetzen, ...
- **Passion:** stolz sein, sich engagieren,
- **Peers:** teilen, zusammenarbeiten, diskutieren
- **Play:** eigene Lebenswelt, Spieltrieb ausleben, Wirkprinzipien verstehen, Herausforderungen designen und bewältigen, ...

## Miniwelt-Konzept im Geiste von LOGO

- visuelles und enaktives Nachverfolgen von Programmierkonzepten
- sofortiges, visuelles und interaktives Feedback
- Code für zu sofort sichtbaren Zustandsänderungen in der Miniwelt
- damit wird die kognitive Weiterentwicklung im Sinne der Phasen nach Piaget / Lister unterstützt

## Konzeptorientierung vs. Werkzeugorientierung

- Umsetzung zentraler Konzepte der Programmierung (siehe Blockkategorien)
- Blockbasierung vermeidet hohen Zeitaufwand für Training von Syntax

Scratch greift in der Tradition von LOGO viele Ideen auf und entwickelt diese konsequent und mit weiterhin starkem Fokus auf die konstruktivistische bzw. konstruktionistische Sicht auf den Lernprozess weiter.

# Weitere Vorteile von Scratch

## Lernmaterialien

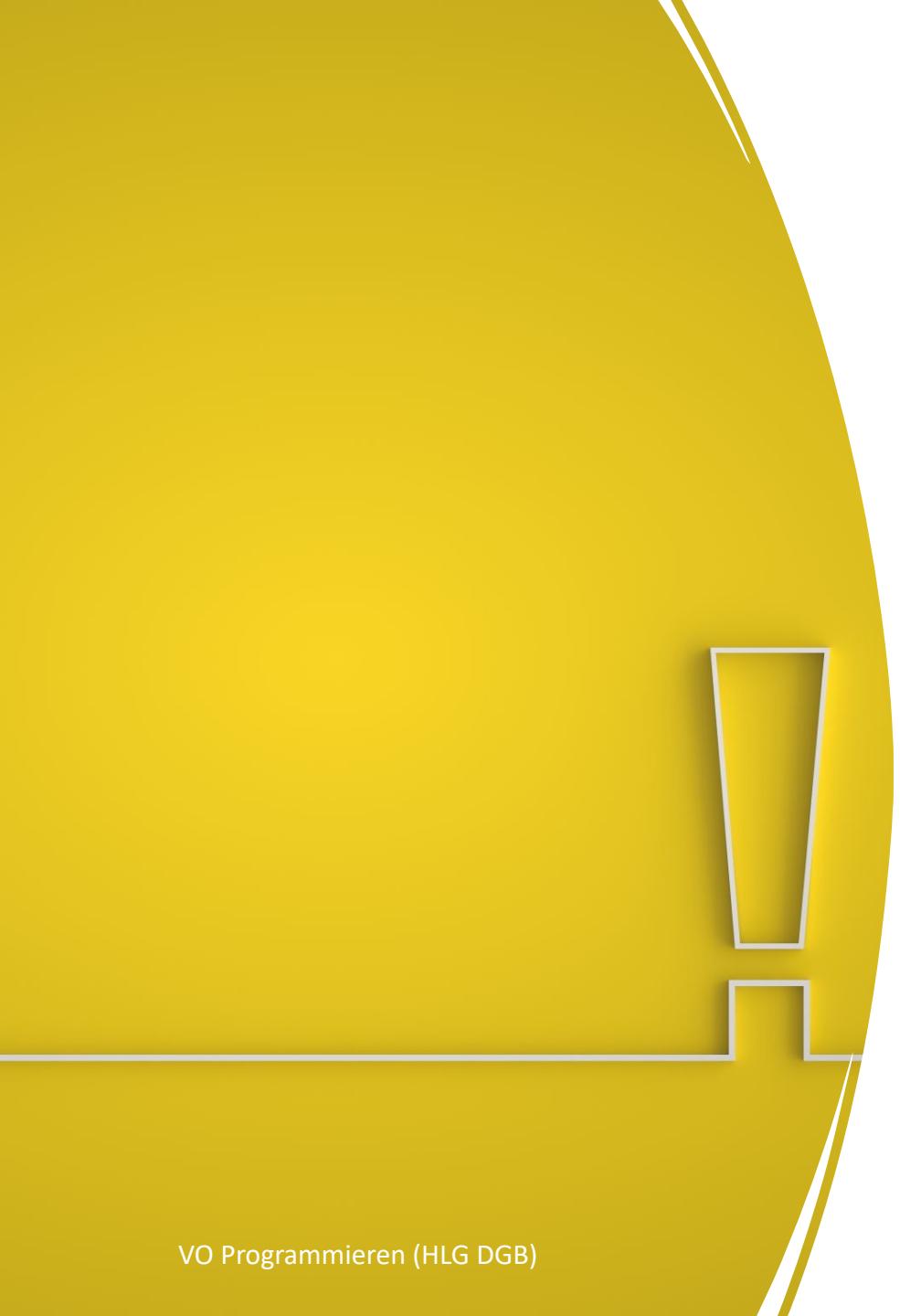
- Interaktive Tutorials
- Beispiele von Peers
- Sharing-Gedanke

## Lebenswelt der Schüler

- Problemdomäne spezialisiert auf Spiele, Multimedia
- Extensions für Robotik, Microcontroller, KI, Turtle-Grafik etc.
- Konzeptioneller Fokus auf Lebenswelt der Schüler:innen:
  - Interaktiv vs. EVA
  - Events vs. Batch
  - Parallel vs. Sequentiell
  - Multimedia vs. Kommandozeile
  - Austauschplattform für die Projekte

## Keine lokale Installation nötig

- Browser-Applikation



# Probleme mit Tools wie Scratch ...

---

- Codeblöcke werden bei größeren Programmen unübersichtlich
- Didaktische Tools werden manchmal nicht ernst genommen
- Zu starker Fokus auf Miniwelt oder auf Produkte und zu wenig Fokus auf zentrale Konzepte
- Zu starker Fokus auf Trial and Error und zu wenig Fokus auf Verständnis, auf etablierte Problemlösungsprozesse, kritisches Denken etc.
- Zu starker Fokus auf das mächtige Werkzeug mit alle den Möglichkeiten erhöht den Extrinsischen Cognitive Load

# WH und Abschluss der LV

# WH von zentralen Konzepten

## Modellbildung

Im Zentrum des DGB-Unterrichts, der sich mit den Konzepten, Prozessen und Ideen der Computerprogrammierung beschäftigt, steht aus Sicht der Informatikdidaktik daher ein informatischer Modellbildungsprozess zur Lösung authentischer Problemsituationen. Wichtige Konzepte und Werkzeuge der Informatik bzw. der Programmierung sollen zur Lösung von Problemen aus der Lebenswelt der Schüler:innen verwendet werden, um die zentralen Ideen und die zentralen Vorgangsweisen (Prozesse) zu thematisieren.

Ein Modell ist eine abstrahierte Beschreibung eines realen oder geplanten Systems, das die für eine bestimmte Zielsetzung wesentlichen Eigenschaften des Systems erhält. Modellbildung ist die Beschreibung eines solchen Systems, z. B. mittels:

- Systembeschreibungen
- Verarbeitungsvorschriften
- Datenstrukturen

Modellierungstechniken:

- Datenmodellierung
- Ablaufmodellierung
- Zustandsorientierte Modellierung
- Funktionale Modellierung
- Objektorientierte Modellierung

(nach Hubwieser, Didaktik der Informatik)

Wir lösen ein (authentliches) Problem im Rahmen des informatischen Modellbildungsprozesses also, indem wir geistige und praktische Techniken der Informatik zur Anwendung bringen. Wir lernen dabei zentrale informative Konzepte und Prozesse kennen und wir nutzen charakteristische Werkzeuge der Informatik, um Lösungen für Probleme zu realisieren. Dadurch entwickeln wir ein Verständnis über den Aufbau und die Wirkungsweise von Informatiksystemen (vgl. dazu auch informative Allgemeinbildung).

# WVH von zentralen Konzepten

## Zentrale Konzepte

Im Rahmen der Nutzung von Programmiersprachen zur Lösung von Problemen gibt es verschiedene „Werkzeugkästen“ - auch Paradigmen genannt, die wiederum unterschiedliche Modellierungstechniken betreffen.

Ein etablierter Werkzeugkasten ist der imperativ-prozedurale. Er enthält viele wichtige informatische Konzepte, von denen einige sogar als **Fundamentale Ideen der Informatik** (siehe unten) gesehen werden, darunter (Auszug):

- [Programm / Quellcode / Anweisungen](#)
  - Kompilieren und interpretieren
  - [Syntaxfehler](#)
  - [Laufzeitfehler](#)
  - [Logische Fehler](#)
- [Eingabe - Verarbeitung - Ausgabe \(EVA\) in Python](#)
- [Bedingungen](#)
- [Logische Verknüpfungen](#)
- [Variablen / Sichtbarkeit / Gültigkeit](#)
- Werte (Literale)
- [Zuweisung \(Assignment\)](#)
- [Datentypen](#)
  - [Zahlen in Python](#)
  - [Zeichenketten in Python](#)
  - [Wahrheitswerte in Python](#)
  - [Typumwandlung in Python](#)
- Listen und andere Datenstrukturen in Python:
  - Listen: <https://www.inf-schule.de/imperative-programmierung/python/konzepte/listen>
  - Listen: <https://docs.python.org/3/tutorial/introduction.html#lists> (Lists)
  - Sequence Types Lists, Tuples, Range: <https://docs.python.org/3/library/stdtypes.html#typesseq>
  - Sequence Type String: <https://docs.python.org/3/library/stdtypes.html#textseq>
  - Sets, Dictionaries: <https://docs.python.org/3/tutorial/datastructures.html>
- [Kontrollstrukturen:](#)
  - Sequenzen von Anweisungen
  - [Fallunterscheidung \(bedingte Verzweigungen\)](#)
  - [Wiederholung \(Schleifen\)](#)
  - [Ausnahmebehandlung](#)

# WH von zentralen Konzepten in Scratch (Zahlenraten)

Scratch interface showing a script for a frog game.

**Code Tab:**

- My Blocks:**
  - gameover
  - goToLevel1
  - initialisieren
  - schritt
  - sprung
- Scratch Script:**

```
when green flag clicked
initialisieren
repeat until [ratezahl = zufallszahl]
    ask [Schon viele sind an diesem Rätsel gescheitert ...] and wait
    set [ratezahl v] to [answer]
    if [ratezahl > zufallszahl] then
        change [versuche v] by [1]
        play sound [Croak v] until done
        sprung [60]
        say [Nein, nicht ganz so viele ...] for [1] seconds
    else
        if [ratezahl < zufallszahl] then
            change [versuche v] by [1]
            play sound [Croak2 v] until done
            schritt
            say [Nein, etwas mehr waren es schon ...] for [1] seconds
        end
    end
    if [versuche > 10] then
        gameover
    end
end
goToLevel1
```
- Definitions:**
  - sprung: 

```
define sprung [number or text]
glide [0.2 secs to x: x position y: y position + number or text]
glide [0.2 secs to x: x position y: y position - number or text]
```
  - schritt: 

```
define schritt
glide [0.2 secs to x: x position - 20 y: y position]
glide [0.2 secs to x: x position + 20 y: y position]
```
  - gameover: 

```
define gameover
hide
broadcast [gameover v]
stop [all v]
```
  - goToLevel1: 

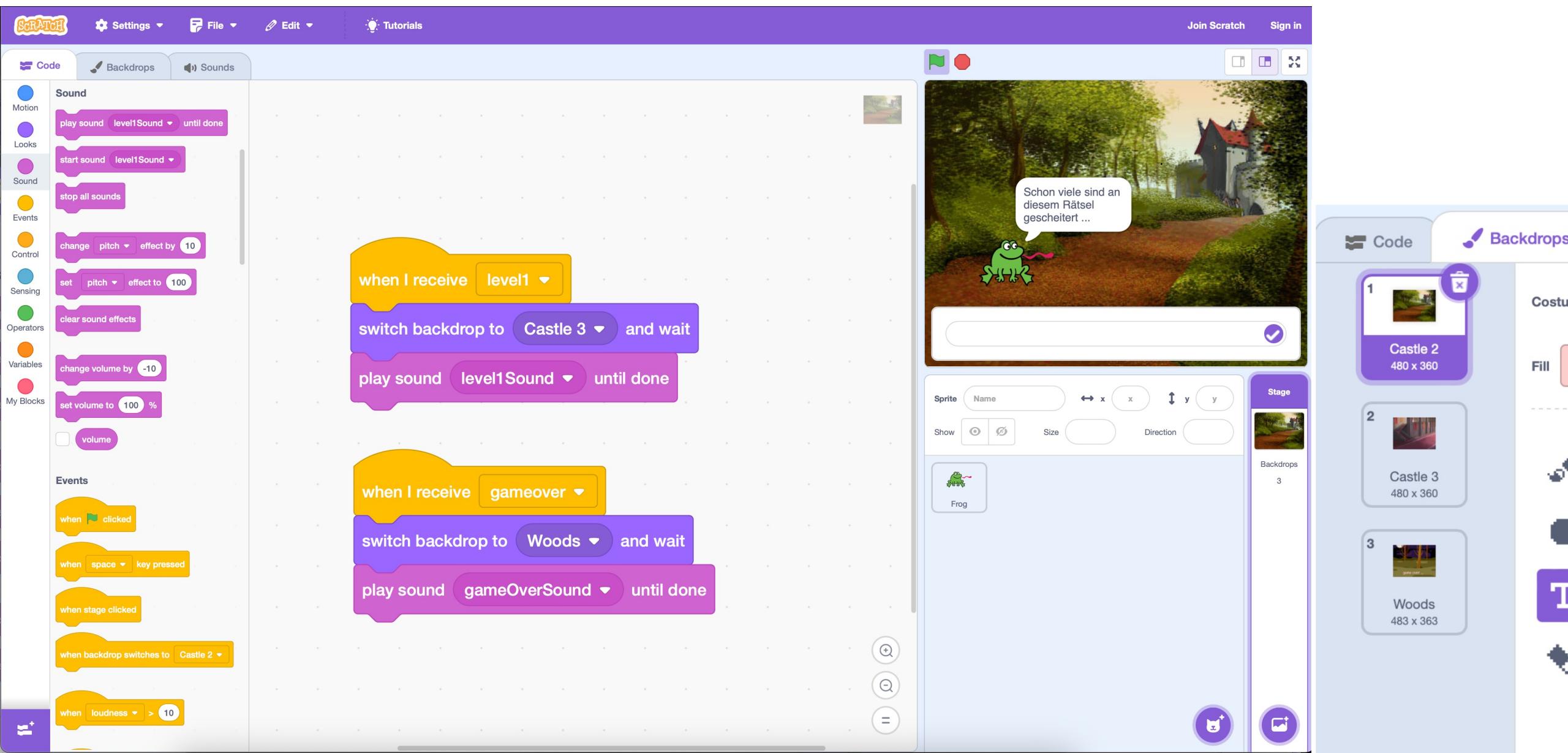
```
define goToLevel1
repeat (10)
    change size by [10]
    go to x: [-141] y: [40]
    switch costume to [Princess-b]
    say [Endlich, jetzt können wir los ... rein ins Abenteuer ...] for [3] seconds
    broadcast [level v]
end
```
  - Initialisieren: 

```
define Initialisieren
set size to [100] %
go to x: [-125] y: [-49]
switch costume to [frog]
switch backdrop to [Castle 2]
show
set [versuche v] to [0]
set [ratezahl v] to [-1]
set [zufallszahl v] to [pick random 1 to 100]
```

**Stage:**

- Sprite:** Frog (costume: frog, backdrop: Castle 2, size: 100, direction: 90).
- Backgrounds:** 3 backdrops are available.

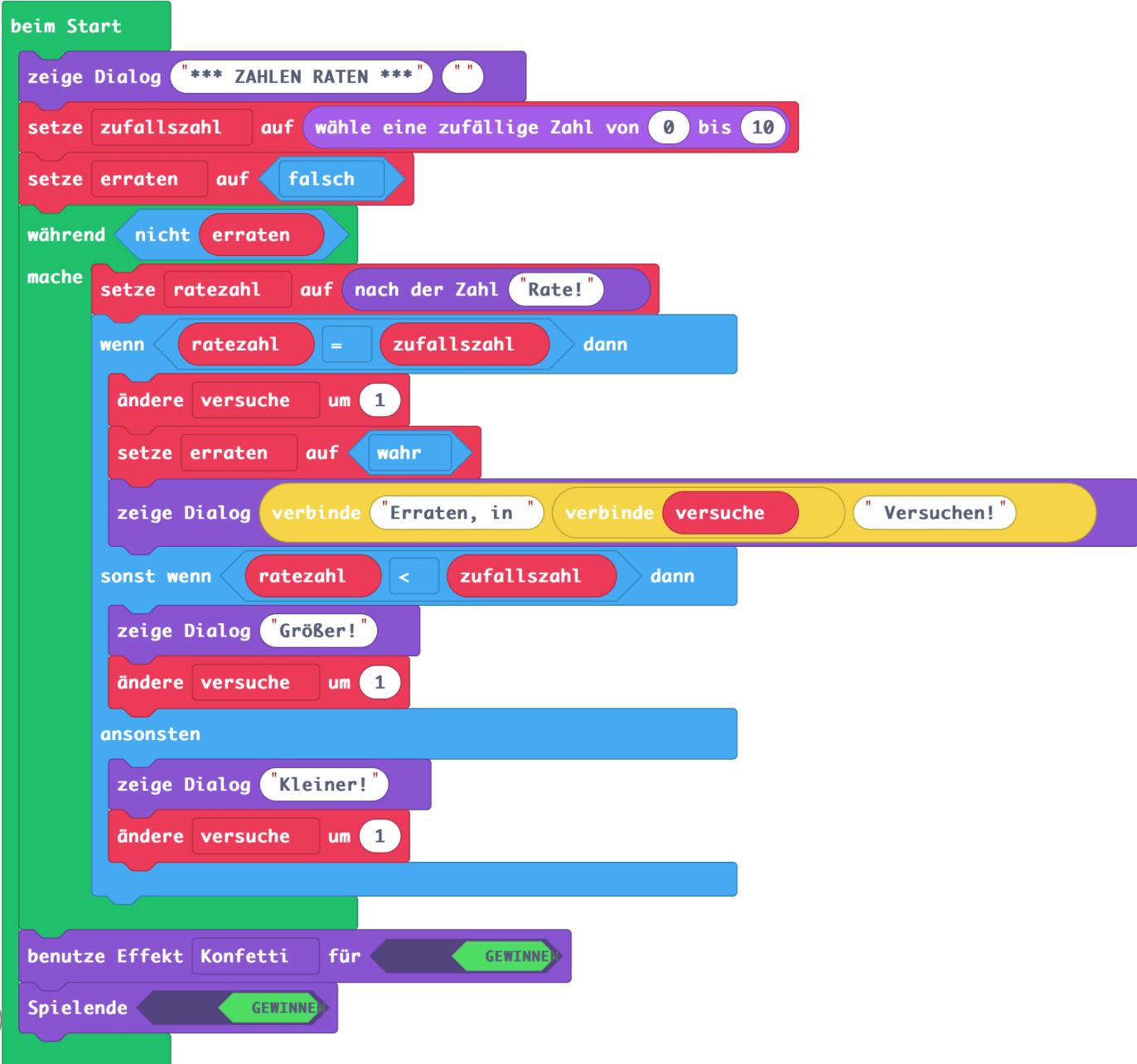
# WH von zentralen Konzepten in Scratch



# Zahlenraten mit MakeCode Arcade

## Lösung als Spiel mit Microsoft MakeCode Arcade

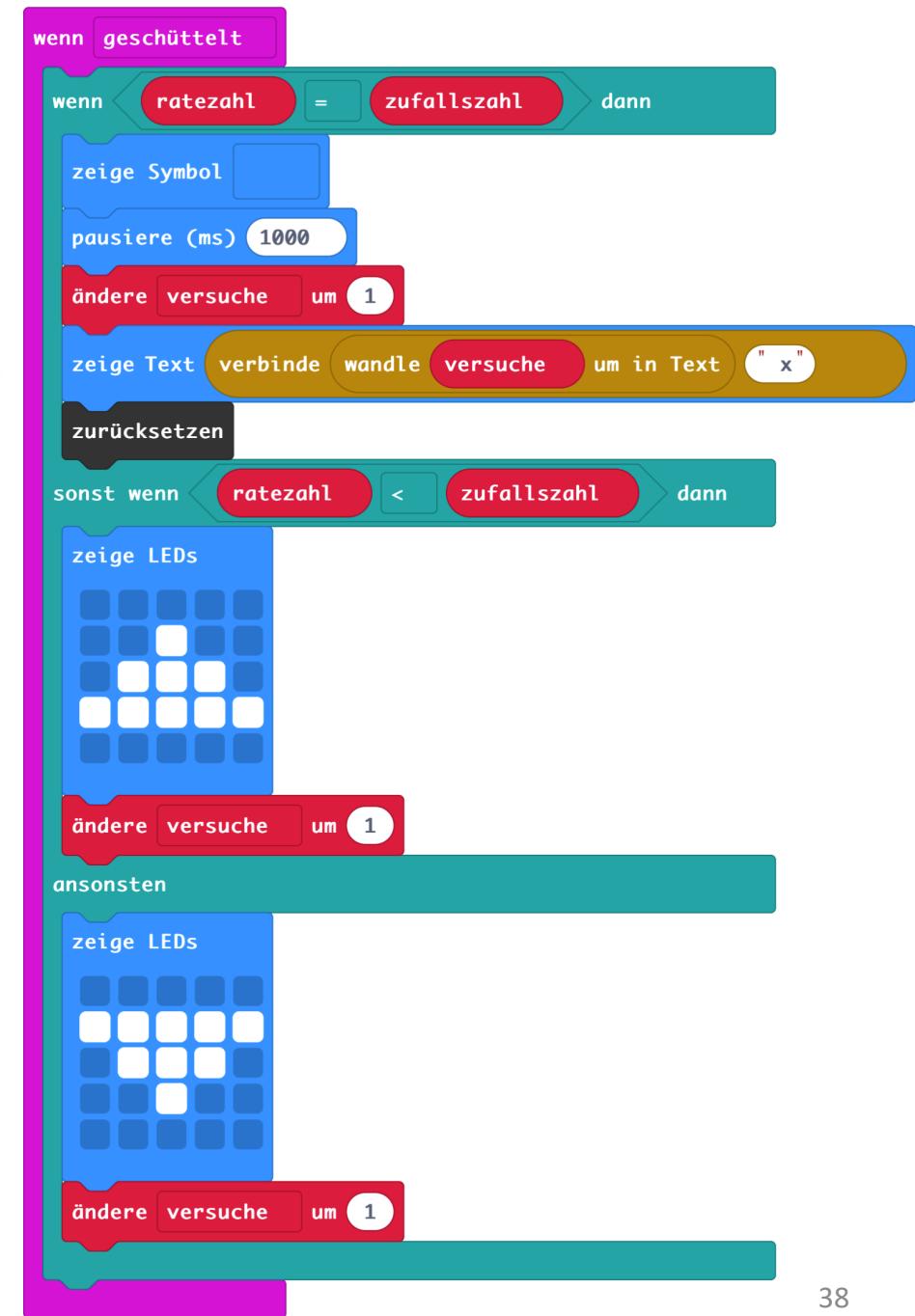
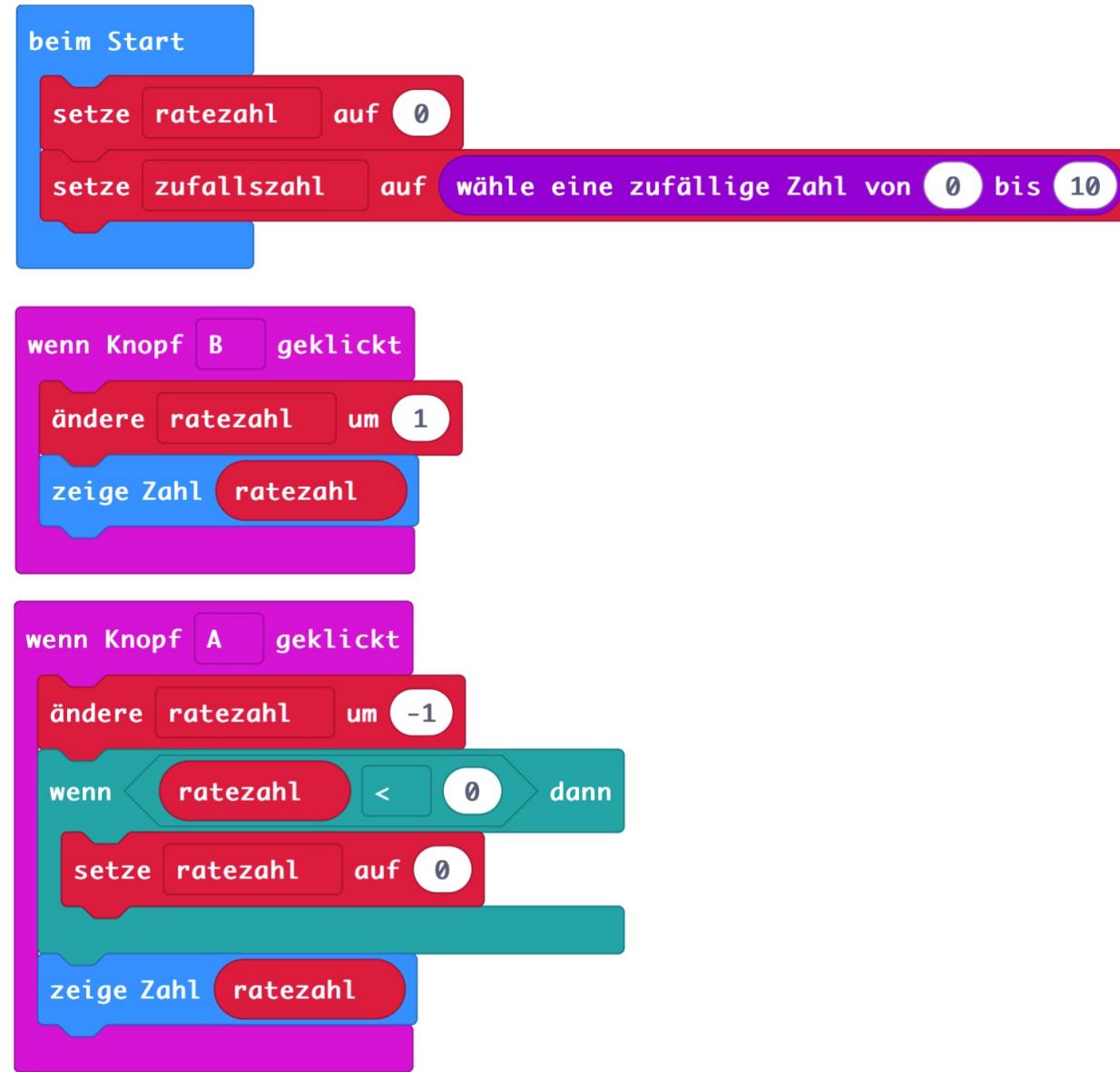
### Blöcke



# Zahlenraten mit MakeCode micro:bit

## Lösung als Spiel mit Microsoft MakeCode micro:bit

Blöcke



# Zahlenraten mit Python (Blöcke)

## Blöcke

```
import random
set min = 1
set max = 100
set anzahlVersuche = 0
set zufallszahl = random.randint(min, max)
set gefunden = False
while not gefunden:
    set eingabe = input("Zahl zwischen " + str(min) + " und " + str(max) + " eingeben!")
    if eingabe.isnumeric():
        set eingabeZahl = int(eingabe)
        if eingabeZahl == zufallszahl:
            if eingabeZahl == zufallszahl:
                set anzahlVersuche = anzahlVersuche + 1
                print("Du hast die Zahl erraten! Anzahl der Versuche: " + str(anzahlVersuche))
                set gefunden = True
            elif eingabeZahl < zufallszahl:
                print("Leider nicht, die gesuchte Zahl ist größer!")
                increase anzahlVersuche by 1
            else:
                print("Leider nicht, die gesuchte Zahl ist kleiner!")
                increase anzahlVersuche by 1
        VO Programmieren (HIC-BGB)
        print("Auf Wiedersehen!")
    
```

# Zahlenrätseln mit Python

## Python

```
import random
min = 1
max = 100
anzahlVersuche = 0
zufallszahl = random.randint(min, max)
gefunden = False
while not gefunden:
    eingabe = input("Zahl zwischen " + str(min) + " und " + str(max) + " eingeben!")
    if eingabe.isnumeric():
        eingabeZahl = int(eingabe)
        if eingabeZahl == zufallszahl:
            anzahlVersuche = anzahlVersuche + 1
            print("Du hast die Zahl erraten! Anzahl der Versuche:" + str(anzahlVersuche))
            gefunden = True
        elif eingabeZahl < zufallszahl:
            print('Leider nicht, die gesuchte Zahl ist größer!')
            anzahlVersuche += 1
        else:
            print('Leider nicht, die gesuchte Zahl ist kleiner!')
            anzahlVersuche += 1
    print('Auf Wiedersehen!')
```

# WH von zentralen Konzepten

<https://www.inf-schule.de/imperative-programmierung>

6: Startseite / Imperative Programmierung

## Imperative Programmierung

---

### In diesem Kapitel

- [Algorithmisches Problemlösen mit Kara](#)
- [Algorithmisches Problemlösen mit Scratch](#)
- [Imperative Programmierung mit Python](#)
- [Algorithmisches Problemlösen mit Spacebug](#)

- [Startseite](#)
- 
6. Imperative Programmierung
- 
1. Algorithmisches Problemlösen mit Kara
  2. Algorithmisches Problemlösen mit Scratch
  3. Imperative Programmierung mit Python
  4. Algorithmisches Problemlösen mit Spacebug

# WH von zentralen Prozessen

Softwareentwicklungsprozess + Computational Thinking

## 1. Problemanalyse, etwa durch:

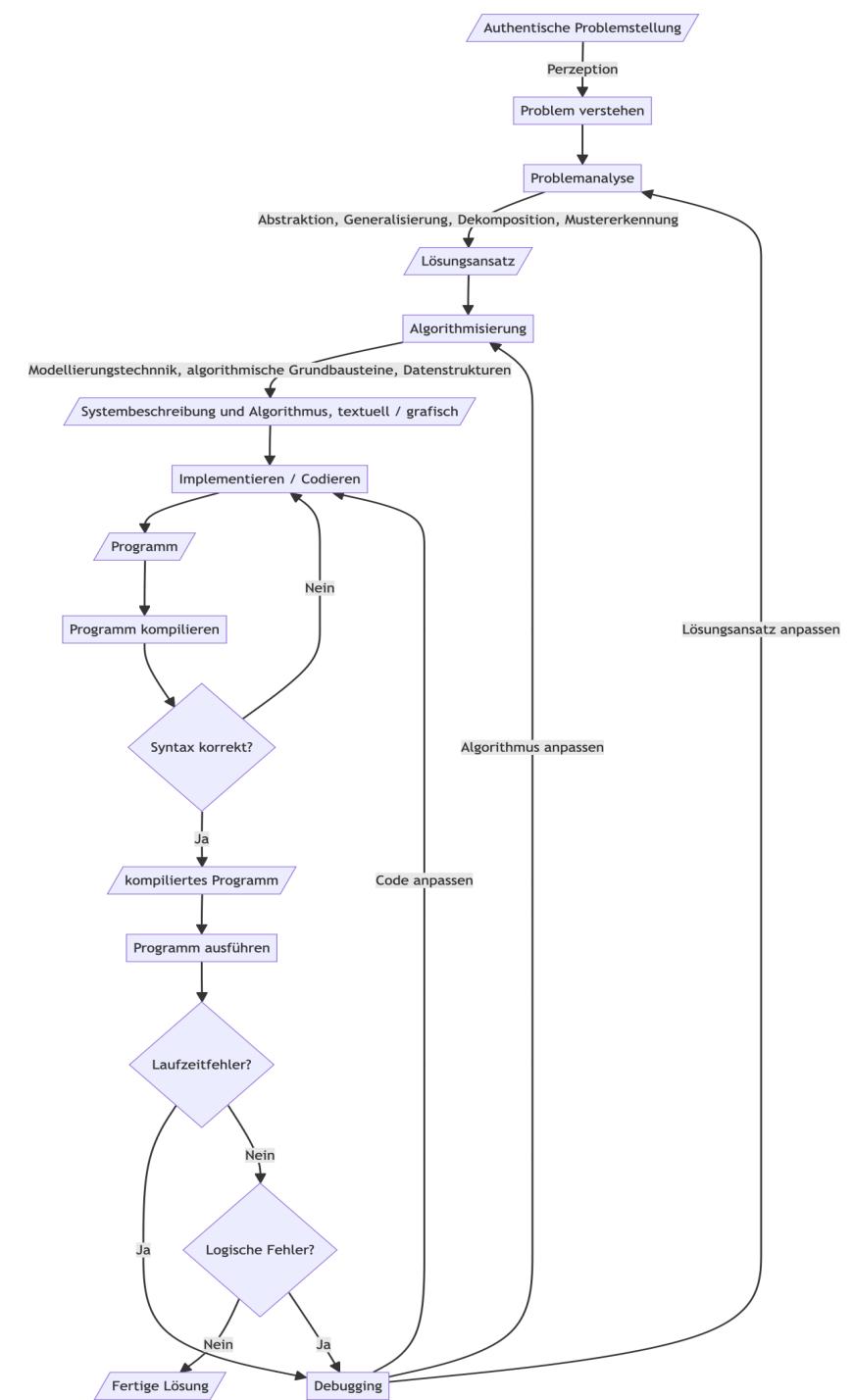
1. Problem spezifikation
  2. Abstraktion
  3. Generalisierung
  4. Mustererkennung
  5. Lösungsansätze

## 2. Algorithmisierung, etwa durch:

1. (grafische) Modellierungstechniken (z.B.  
Ablaufmodellierung, Zustandsmodellierung, funktionale  
Modellierung, objektorientierte Modellierung etc.)
  2. Algorithmische Grundbausteine
  3. Datenstrukturen

### 3. Implementierung und Test, etwa mit:

1. IDE
  2. Programmiersprache
  3. Laufzeitumgebungen und Compiler



## Problemanalyse

Das Problem wird analysiert und möglichst präzise formuliert. Dazu können z. B. Techniken des Computational Thinkings wie Abstraktion, Dekomposition oder Mustererkennung angewendet. Außerdem wird genau spezifiziert, wann das Problem als gelöst angesehen wird (Kriterien).

- Lösungsspezifikation:
  - Wir definieren möglichst genaue "Abnahmekriterien" für eine Lösung.
- Abstraktion:
  - Im Bereich Modellbildung: Wir fokussieren auf die für die Lösung absolut notwendigen Lösungselemente und lassen alles andere weg.
  - In Bezug zur Dekomposition: Wir verwenden vorgefertigte Teillösungen im Kontext größerer Lösungen (als Black-Box) weiter, ohne uns über die Implementierungsdetails Gedanken zu machen.
- Generalisierung:
  - Wir versuchen eine Lösung zu finden, die nicht nur Spezialfälle von Problemen, sondern eine ganze Problemkategorie löst.
- Dekomposition:
  - Wir teilen das Problem in verschiedene Teilprobleme.
  - Wir überlegen uns, wie wir diese Teillösungen wieder zu einer Gesamtlösung zusammenbauen müssen.
- Mustererkennung:
  - Wir halten nach Mustern im Bereich der verwendeten Daten oder auch in Bezug auf die Ablaufstruktur und damit nach Automatisierungspotential Ausschau.
  - Immer wiederkehrende Lösungsteile lassen sich durch Programmiersprachen sehr effizient implementieren.

Auf Basis der Problemanalyse können dann **erste Lösungsansätze** entwickelt werden, die dann im Rahmen der Algorithmisierung (siehe unten) aufgegriffen und als Algorithmus umgesetzt werden.

# Algorithmisierung

Im Rahmen der Algorithmisierung folgt die Beschreibung des Systems und eines dazu passenden Lösungsalgorithmus. Wir verwenden dazu (grafische) Methoden verschiedener Modellierungsparadigmen (darunter Datenmodellierung, Ablaufmodellierung, zustandsorientierte Modellierung, funktionale Modellierung, objektorientierte Modellierung), sowie dazu passende algorithmische Grundbausteine bzw. Datenstrukturen (siehe Thema "Zentrale Ideen" bzw. "Konzepte" weiter oben).

Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. Algorithmen bestehen aus endlich vielen, wohldefinierten Einzelschritten. Damit können sie zur Ausführung in ein Computerprogramm implementiert, aber auch in menschlicher Sprache formuliert werden. Bei der Problemlösung wird eine bestimmte Eingabe in eine bestimmte Ausgabe überführt. (<https://de.wikipedia.org/wiki/Algorithmus>)

Die Beschreibung des Algorithmus erfolgt z.B. über:

- Textuelle Beschreibung des Algorithmus (formlos, konzeptorientiert)
- Pseudocode
- grafische Darstellungen zur Systembeschreibung, darunter:
  - Flussdiagramme bzw. [Programmablaufpläne] (<https://de.wikipedia.org/wiki/Programmablaufplan>)
  - Struktogramme bzw. Nassi-Shneidermann-Diagramm
  - Zustandsdiagramme
  - weitere UML-Diagrammarten

## Implementierung und Test

Nachdem wir den Lösungsalgorithmus entworfen haben, gehen wir in die Umsetzung über. Dazu suchen wir Werkzeuge (IDE, Programmiersprache, Compiler etc.), die es uns aufgrund der unterstützten Konzepte bzw. aufgrund des gewählten Modellierungsparadigmas ermöglichen, den Algorithmus als Computerprogramm umzusetzen, auszuführen und zu testen.

Die Implementierung erfolgt iterativ in Zyklen. D.h. es werden Schritt für Schritt (weitere) Teile der Lösung implementiert und immer wieder getestet.

Im Rahmen der Implementierung wird es aufgrund von Syntaxfehlern, Laufzeitfehlern oder auch logischen Fehlern im Ansatz zu Anpassungen des Lösungsansatzes, des Algorithmus und damit der Implementierung kommen.

Die Lösung soll so weit wie möglich generalisiert werden, d.h. sie soll eine Klasse von Problemen und nicht nur ein spezifisches Problem mit bestimmten spezifischen Daten lösen.

# WH von zentralen Prozessen

POLYGONIX-Beispiel

<https://github.com/clander/voprogrammieren/tree/main/VO-Teil-1/GrundkonzepteProgrammierung/Polygonix>

# Was machen wir in der Übung?

# Erster Prototyp eures Abschlussprojektes

- In der Übung zur VO:
  - Idee für Unterrichtsbeispiel / Projekt, das zentrale Konzepte und zentrale Prozesse der Programmierung aufgreift
  - Werkzeug für die Umsetzung wählen
  - Iterative Implementierung des ersten Prototyps (mehrere Versionen)
  - Explikation der betroffenen Konzepte und Prozesse
- Abschlussprojekt (nächstes Semester)
  - Auf Basis des Prototyps:
  - Unterricht vollständig planen / Materialien vollständig bereitstellen:
    - Planungsmodell wählen (z.B. Lernziele – Lernzielkontrolle – Unterrichtsverlauf)
    - Methodik wählen:  
<https://github.com/clander/voprogrammieren/blob/main/Didaktik/Methodik.md>
    - Materialien produzieren

# Exkurs: Konstruktivismus

Methodische Implikationen



# Akkomodation und Lernen im konstruktivistischen Sinne

---

- Akkomodation erfordert entsprechende Stimuli / Anreize / kognitive Konflikte:
  - Eigenartiges, Kontroversielles, Unerklärliches, Erstaunliches
  - soziale Stimuli
- Akkomodation erfordert entsprechende Haltung und Motivation
  - lernen wollen, verstehen wollen, weiterkommen wollen
  - intensive Beschäftigung, Fehlschläge, viel Arbeit
  - Motivation und Willen zur intensiven Beschäftigung

# Akkomodation und Lernen im konstruktivistischen Sinne

---

- Lernen als Verschärfungsprozess (Modrow)
  - nebelhafte, unzureichende Vorstellungen der Welt (Ungleichgewicht in Bezug auf das eigene Verständnis)
  - wird durch intensive Beschäftigung in die bestehenden Erfahrungen eingepasst (Gleichgewicht herstellen).
- EUREKA-Moment / Aha-Erlebnisse
  - Das Gleichgewicht in den kognitiven Strukturen ist wieder hergestellt.
  - CLT: Ein neues Schema ist ausgebildet.
  - Aha-Erlebnisse → wichtiges Instrument für Reflexion
- Üben, üben, üben
  - Intensive praktische Beschäftigung hilft beim Bewusstmachen des kognitiven Ungleichgewichts.
  - Das wiederum ist als Anreiz Voraussetzung für Akkomodation.

# Konstruktivismus

- Lernen = Einbau von neuen Inhalten in bestehende kognitive Strukturen
- Neuartige Erfahrungen führen zu kognitiver Störung
- Störungen erzeugen kognitive Zustandsänderungen

Assimilation: Wahrnehmung wird in bestehendes Schema eingepasst

Akkommodation: Erweiterung bestehender Schemata an neue Wahrnehmung

Im Mittelpunkt der konstruktivistischen Sicht auf Lernprozesse steht die aktive Konstruktion von Wissen. Mit Blick auf die bereits beschriebene Vorstellung eines Lernprozesses als Informationsverarbeitungsprozess würde ein Konstruktivist von Informationskonstruktion und nicht von Informationsverarbeitung sprechen, um damit (bei relativem Bedeutungsverlust der Instruktion) die aktiveren Rolle der Lernenden zu betonen. An der kognitiven Grundvorstellung des Lernens als Informationsverarbeitungsprozess (bzw. eben als Konstruktionsprozess) ändert diese etwas anders gelagerte Schwerpunktsetzung jedoch wenig. ([KL07, Seite 44])

**Pragmatische Position:** Lernende konstruieren aktiv ihre eigene Wirklichkeit indem sie auf kognitive Konflikte mit Assimilation und Akkommodation reagieren und so kognitive Konflikte auflösen.

Modrow spricht von „Verschärfungsprozess“ → nebelhafte, unzureichende Vorstellung der Welt wird durch intensive Beschäftigung den Erfahrungen angepasst.

Für die Informatikdidaktik ist die konstruktivistische Sicht auf den Lernprozess besonders wichtig. Jerome Bruner gilt als einer der Urväter des Konstruktivismus UND als Urvater des didaktischen Konzeptes der **Fundamentalen Ideen**.

# Methodische Implikationen (Modrow)

Lernen erfolgt nur durch kognitive Konflikte / Kollisionen

können nur auftreten, wenn sich Lernende aktiv mit dem Gegenstand auseinandersetzen, sich dafür interessieren, ihn für relevant halten



Aufgabe der Lernenden: Bereitschaft zur aktiven Auseinandersetzung und zur Persönlichkeitsentwicklung

eigene Vorstellungen erproben, ändern, erweitern

Blockaden dieser Bereitschaft müssen beseitigt werden (dauerhaft und ggf. mit Konsequenzen)



Rolle der Lehrenden

beziehen sich auf individuelle Vorkenntnisse, Misskonzeptionen, Lernprozesse und Lebenswelt der Lernenden (Betroffenheit erzeugen)

konfrontieren die Lernenden mit ihrem unvollständigen Verständnis zu den Inhalten

müssen fachlich / fachdidaktisch souverän reagieren, um individuell zugeschnittene Anregungen geben zu können

# Methodische Implikationen(Hubwieser)

---

Aktive Beteiligung, Motivation und Interesse sind die Grundvoraussetzungen für das Lernen.

---

Die Lernenden übernehmen bis zu einem gewissen Grad selbst die Steuerung und Kontrolle über den Lernprozess.

---

Lernen kann nur mit Bezug auf die Erfahrungen, auf das Vorwissen und durch Interpretationen der Lernenden selbst erfolgen.

---

Lernen erfolgt immer in Kontexten, d. h. situativ.

---

Lernen ist ein sozialer Prozess.

# Charakteristika für konstruktivistische Lernumgebungen (Humbert, Helmke, Klauer, Hubwieser)

Handlungsorientierung, entdeckendes Lernen und aktive Beteiligung in möglichst vielen Bereichen des Unterrichts fördern

Eigenverantwortung für den Lernprozess fördern

metakognitive Fähigkeiten fördern (Lernen lernen, Selbstbeobachtung, Selbstbewertung, Selbstverstärkung etc.)

Authentizität für alle Belange des Unterrichts garantieren (authentische Lernsituationen, lernen in verschiedenen Kontexten, situiertes Lernen)

kooperatives Lernen fördern (z. B. durch Einbettung des Lernens in einen sozialen, kooperativen Kontext)

Perspektivenwechsel bei der Erschließung der Inhalte ermöglichen

Erschließung von Problemlösemethoden durch die Lernenden selbst ermöglichen

genügend Bearbeitungszeit zur Verfügung stellen

formatives Feedback intensiv nutzen

Instruktionsteile nicht vernachlässigen (denn fachspezifisches Vorwissen fördert den Lernprozess nachweislich am stärksten)

neue Lehrendenrolle als Coach wahrnehmen

# Kognitivismus

- **Lernen**

Im Kognitivismus wird Lernen als Prozess der Informationsverarbeitung gesehen. Unter Einwirkung der metakognitiven Strategien und des Motivationsniveaus nehmen Lernende Informationen auf und verarbeiten sie im Kurzzeitgedächtnis. Diese Informationen werden dann im Langzeitgedächtnis gespeichert und in neuen Situationen zur Anwendung gebracht. Die gespeicherten Elemente sind als Vorwissen wiederum relevant für die Aufnahme und Verarbeitung neuen Wissens. ([KL07, Seite 44])

- **Praxistipps (Hubwieser, Helmke)**

- individuelle Lernvoraussetzungen analysieren (Vorkenntnisse, Lerngewohnheiten, Motivation, Interesse)
- Anknüpfungspunkte an das Vorwissen der Lernenden bieten
- Lerninhalte sachlich strukturieren (Kategorienbildung), hierarchisieren, in übergeordnete Sinnzusammenhänge einordnen, mit Blick auf Vorwissen und Struktur sequenzieren und gut organisiert vermitteln (Optimierung der Instruktion)
- Lernziele und Sinn der Inhalte bekanntgeben
- Lernhilfen definieren (genaue Zielvorgaben, Strukturierungshilfen, Lernaufgaben)
- Steuerung des Lernprozesses durch die Lehrenden bzw. die Umgebung
- Vermeidung von kognitiver Überbelastung (siehe Abschnitt 1.2)

# Instruktion vs. Konstruktion

---

- Unterricht muss Instruktion und Konstruktion bieten
  - hängt mit den verschiedenen Zieldimensionen und Constructive Alignment zusammen
  - Ziele höherer Ebenen → eher konstruktivistisch zu erreichen
  - Ziele niederer Ebenen → eher kognitivistisch / behavioristisch zu erreichen
- Reinmann / Mandl sprechen vom Wissensbasierten Konstruktivismus:
  - Im wissensbasierten Konstruktivismus wird Lernen als eine persönliche Konstruktion von Bedeutung interpretiert, die nur dann gelingt, wenn eine ausreichende Wissensbasis zur Verfügung steht. Zum Erwerb dieser Wissensbasis 182 4 Unterrichtsgestaltung kann auf instruktionale Anleitungen nicht verzichtet werden. ([RM06, Seite 638])
- Auch entdeckendes Lernen funktioniert nach Klauer besonders gut, wenn es als Guided Discovery im Sinne von Jerome Bruner ausgeführt ist, d.h. wenn:
  - [...] der Prozess des Entdeckens behutsam gelenkt wird, fehlendes, aber notwendiges Wissen im Bedarfsfall direkt vermittelt wird, die Komplexität des Problems nicht zu hoch ist oder angemessen reduziert wird und variiierende Aufgaben für eine hinreichende Generalisierung der Erkenntnis und für die Einübung des Transfers sorgen. ([KL07, Seite 99])
- Modrow spricht in seiner Dissertation von pragmatisch konstruktivistisch (und ideenorientierten) Lehr-Lern-Situationen

# Exkurs: Cognitive Load Theorie

Einführung in die Theorie und ihre methodischen Implikationen

# Vorstellung von Lernen nach der CLT (Rey)

- Verarbeitung von Informationen im Arbeitsgedächtnis (sehr klein)
  - Gespeist vom Langzeitgedächtnis (Vorwissen) und vom sensorischen Speicher (gerade aufgenommene, neue Informationen)
  - Kapazität begrenzt → Gleichzeitig können nur 3 – 5 Elemente bereitgehalten, kombiniert, verglichen, manipuliert werden.
  - Zeitlich begrenzt → ohne sofortige Wiederholung 20 bis 30 Sekunden Behaltensleistung
- Verstehen = erfolgreiche, gleichzeitige Verarbeitung von Informationselementen im Arbeitsgedächtnis
- Lernen = Veränderung von Strukturen im Langzeitgedächtnis (sehr groß) auf Basis der Informationsverarbeitung im Arbeitsgedächtnis

# Schemakonstruktion

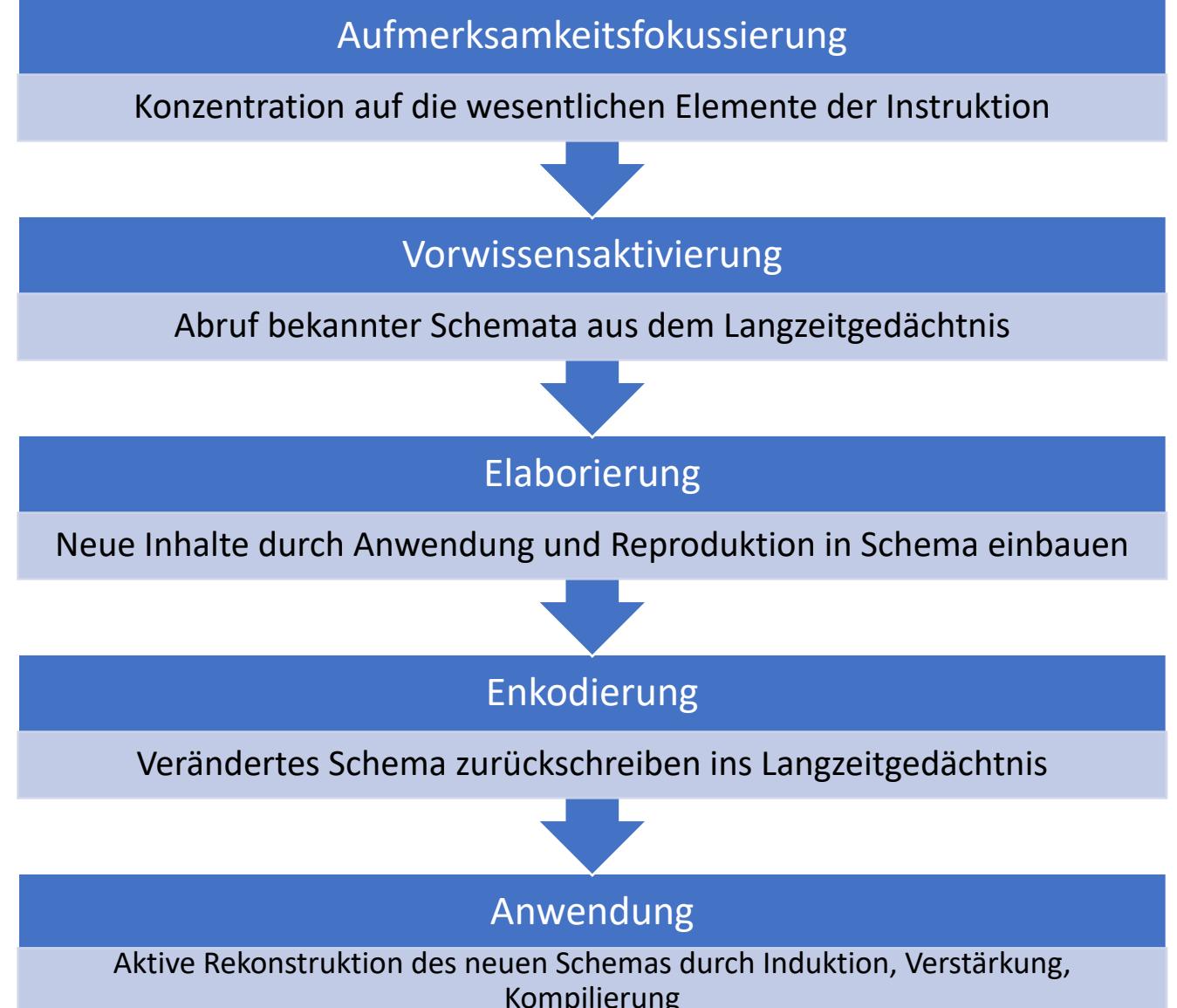
- Experten bringen zur Lösung von Problemen sog. „Schemata“ (Chunks, Frames, Scripts) zur Anwendung
- Aufbau von Schemata und deren automatisierte Anwendung primäres Lernziel

Ein **Schema** ist ein kognitives Konstrukt mit Hilfe dessen Informationen organisiert, strukturiert und (im Arbeitsgedächtnis) schnell verarbeitet werden können. Im Arbeitsgedächtnis wird ein **automatisiertes Schema dabei als ein einziges Element betrachtet**, selbst wenn das Schema eine Vielzahl von interagierenden Elementen betreffen kann. Durch die (automatisierte) Anwendung von Schemata können damit komplexe Herausforderungen mit relativ geringer Belastung des Arbeitsgedächtnisses gemeistert werden.

# Schemakonstruktion

Elaboration („Erarbeiten“)	Induktion	Automatisierung
<ul style="list-style-type: none"><li>• Neue Informationen werden mit vorhandenem Wissen verknüpft, in vorhandene Schemata eingebaut</li><li>• Es entstehen neue Strukturen in neuen Zusammensetzungen</li></ul>	<ul style="list-style-type: none"><li>• Konkrete Lernerfahrungen werden zu abstrakten Schemata generalisiert</li><li>• Transfer</li></ul>	<ul style="list-style-type: none"><li>• Unbewusste Verarbeitung von Informationen / Anwendung von Schemata im Arbeitsgedächtnis fördern → macht Kapazitäten frei</li><li>• Kompilierung = vollautomatisierte Anwendung von Schemata</li><li>• Verstärkung = Erhöhung der Auftrittswahrscheinlichkeit des Schemas in passenden Problemsituationen (wiederholen, neue Kontexte etc.) → Transferlernen</li></ul>

# Lernprozess in der CLT



- Wesentliche Herausforderungen für den Unterricht daher:
- Schemakonstruktion ermöglichen / anstoßen
- Überbelastung des Arbeitsgedächtnisses vermeiden
- Vergessen gerade gelernter Schemata vermeiden (üben, anwenden, verschiedene Kontexte)

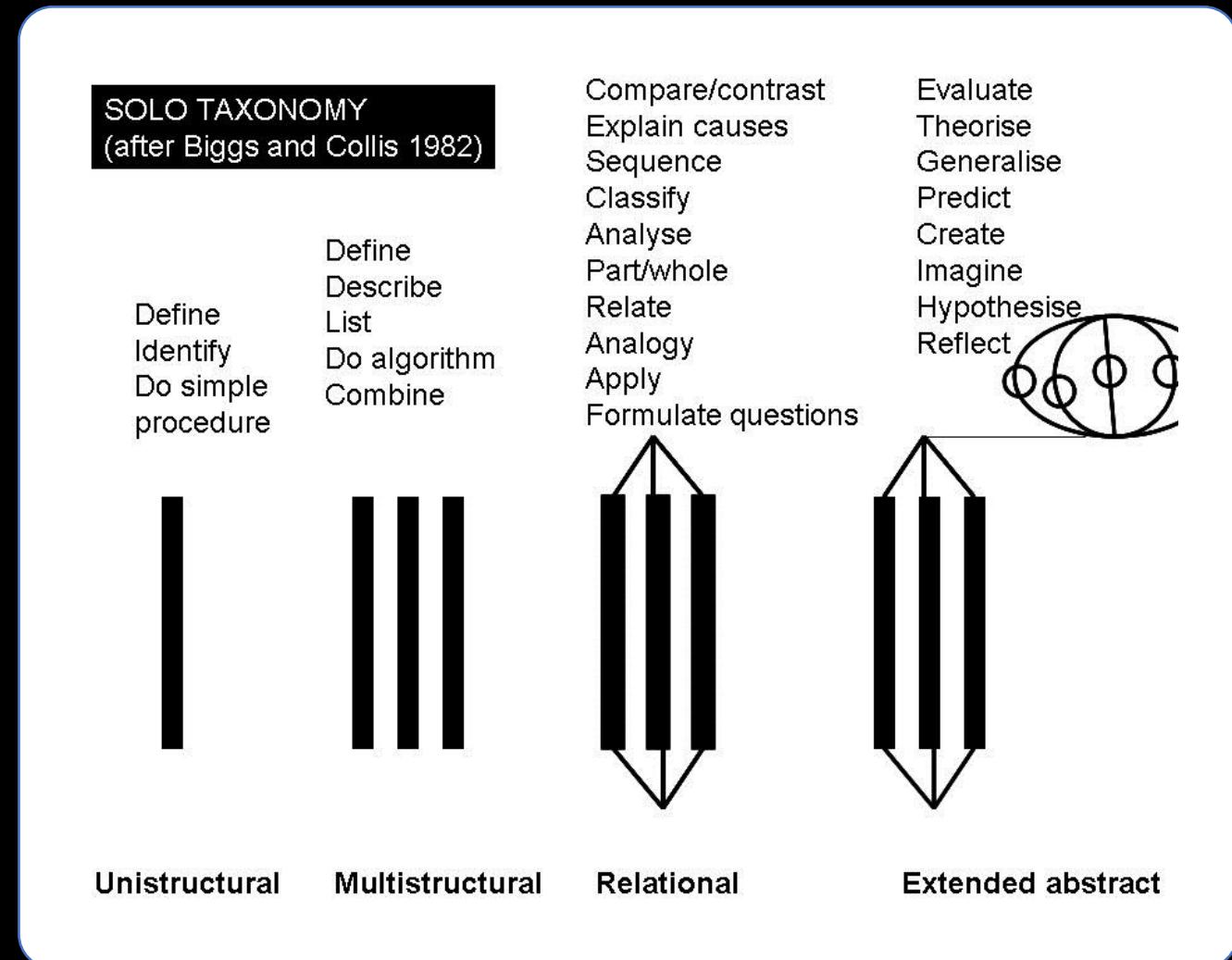
Lernen bedeutet in der CLT also die Herausbildung von Schemata, deren Speicherung im Langzeitgedächtnis und deren automatisierte Anwendung im Arbeitsgedächtnis in (möglicherweise neuen) Problemsituationen. Dies passiert primär durch Elaboration (Anknüpfen an Vorwissen), Induktion (Transferlernen), Kompilierung und Verstärkung (intensive Übung der Anwendung des Schemas und metakognitive Strategien, um Automatisierung zu fördern) im Rahmen von problembasiertem oder beispielbasiertem Lernen (vgl. [Nie+08, Seite 49]).

# Arten von kognitiver Belastung

- *Intrinsischer Cognitive Load*: Dieser ist bestimmt durch die Komplexität der zu erlernenden Fachinhalte bzw. der zu bewältigenden Aufgaben und ist damit abhängig von Aufgabenschwierigkeit, Komplexität oder Umfang. Aufgrund von vorhandenem Vorwissen und Schemata ist Komplexität in diesem Zusammenhang relativ, d. h. je nach Vorwissen der Lernenden unterschiedlich hoch.
- *Extrinsischer Cognitive Load*: Dieser ist bestimmt durch die Komplexität der Lernumgebung bzw. des Lernmaterials (Strukturierung der Wissensvermittlung, Präsentation der Inhalte, Navigation durch die Inhalte etc.).
- *Germane Cognitive Load*: Dabei handelt es sich um den lernrelevanten CL. Es geht um die kognitive Belastung für den reinen Wissenserwerb, d. h. für die Schemakonstruktion im Arbeitsgedächtnis.

# SOLO Taxonomie (Biggs and Collis)

- Die SOLO-Lernzieltaxonomie nach Biggs nimmt die piaget'schen Grundgedanken bzw. das Prinzip der Elementinteraktivität nach Sweller ebenfalls auf.
- Structure of Observed Learning Outcome
- qualitative und hierarchisierte Beschreibung bzw. Kategorisierung von Lernerfolgen bei Aufgaben
- unabhängig von den konkreten Wissensarten anwendbar
- Elementinteraktivität (siehe CLT) als Kriterium der Schwierigkeit von Aufgaben
- Zuordnung von Aufgaben / Kompetenzen auf Basis von:
  - Anzahl der interagierenden Aspekte
  - Komplexität der Beziehungen zwischen den Aspekten
  - Komplexität der nötigen Schlussfolgerungen zur Lösung der Aufgaben



[http://www.kawaha-point.school.nz/Site/Curriculum/SOLO\\_TAXONOMY.ashx](http://www.kawaha-point.school.nz/Site/Curriculum/SOLO_TAXONOMY.ashx)

# SOLO-Taxonomie

- **Prästrukturell** (Misserfolg, Inkompetenz, Kernpunkte nicht verstanden, zusammenhangslos): Die Aufgabe wurde nicht korrekt angegangen, die Lernenden haben den Kern der Aufgabe nicht verstanden.
- **Unistrukturrell** (identifizieren, benennen, nachmachen): Ein Aspekt oder einige wenige Aspekte einer (komplexeren) Aufgabe wurden aufgegriffen und bearbeitet.
- **Multistrukturrell** (kombinieren, beschreiben, aufzählen, einfache Fertigkeiten, auflisten): Mehrere Aspekte der Aufgabe wurden gelernt, aber separat behandelt und können nicht integriert werden.
- **Relational** (verstehen, analysieren, anwenden, argumentieren, vergleichen, kritisieren, in Beziehung setzen, rechtfertigen): Die Teilkomponenten der Aufgabe wurden gelernt und (unter Anwendung entsprechender Konzepte) zusammen verwendet, um ein großes Ganzes zu bilden. Die Anwendung erfolgte durch (nahen) Transfer.
- **Erweitert abstrakt** (erzeugen, formulieren, generieren, Hypothesen bilden, reflektieren, Theoriebildung): Eine Lösung aus der vorhergehenden Stufe wurde abstrahiert und auf ein neues Thema angewendet bzw. reflektiert (nicht spezifischer Transfer).

# Elementinteraktivität

- Element = Abstraktion von allem was gelernt werden soll
- Elementinteraktivität als Gradmesser für die Belastung des AG
  - Niedrig: Elemente können unabhängig voneinander gelernt werden
  - Hoch: Elemente können nur in Kombination erfasst werden
  - Auch viele unabhängige Elemente die gleichzeitig gelernt werden müssen, erzeugen Überbelastung
- Extrinsischer CL (ECL)
  - Elementinteraktivität kann verändert werden, ohne Lerninhalte zu ändern
- Intrinsischer CL (ICL)
  - Elementinteraktivität kann nur verändert werden, wenn die Lerninhalte verändert werden.
- Germane CL (GCL, lernrelevanter CL)
  - Anteil des AG für die Verarbeitung der Inhalte (ICL)
  - Keine unabhängige Größe sondern direkte Funktion des ICL
  - Wenn ECL steigt → mehr Ressourcen für die Bewältigung der Lernmaterialien → weniger Ressourcen für die Verarbeitung der Inhalte → GCL könnte sinken, weil AG kapazitiv und zeitlich beschränkt ist
  - Kognitive Gesamtbelaestung = ECL + ICL (mit GCL = jene Ressourcen, die für die Bewältigung des ICL verwendet werden können)

## Verfügbare Kapazität im Arbeitsgedächtnis



Grad der  
Elementinteraktivität  
abschätzen /  
Schemakonstruktion

$$X = X + 1;$$

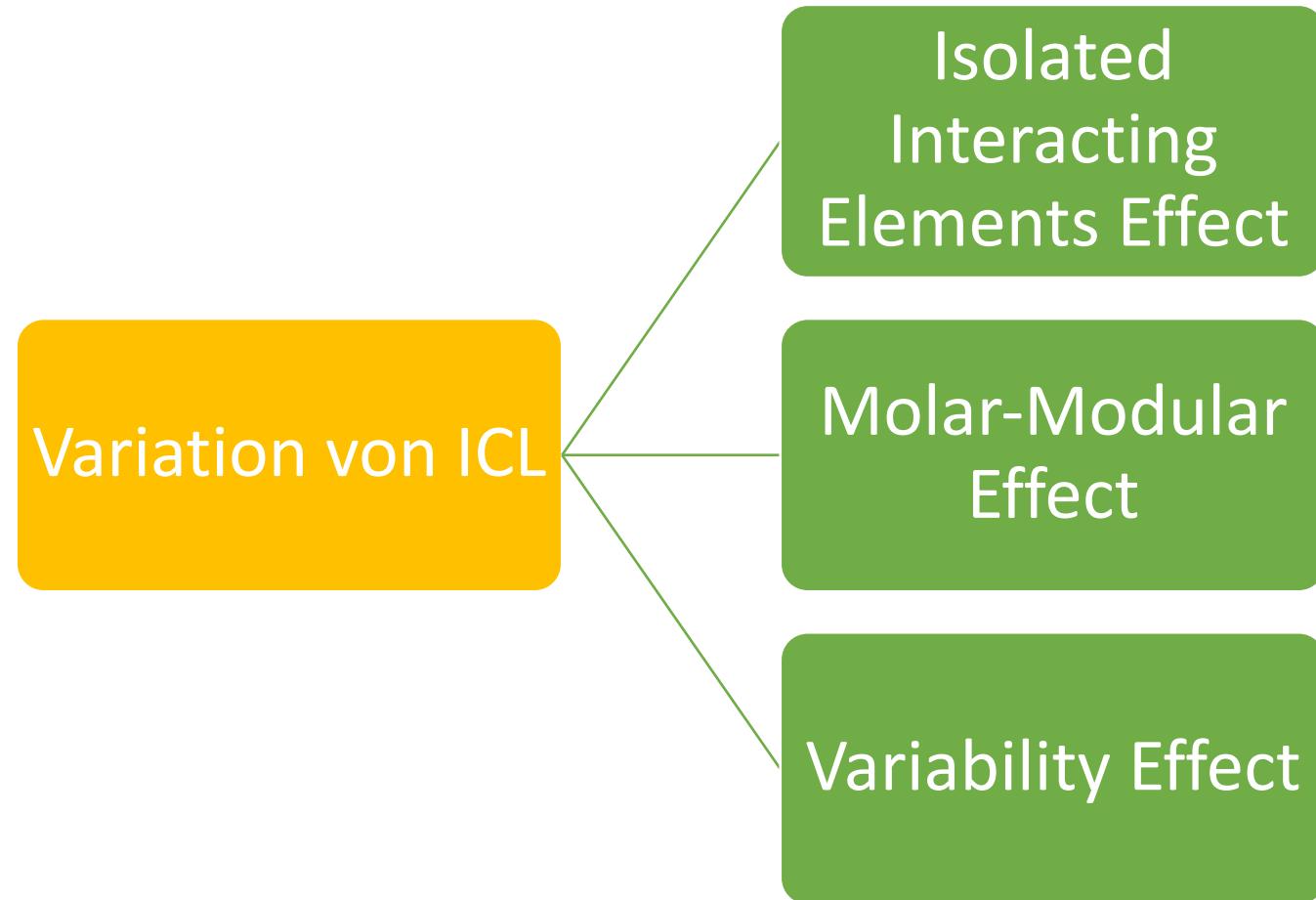
# Grad der Elementinteraktivität abschätzen / Schemakonstruktion

```
1 int[] liste = {3, 4, 65, 3, 22, 3};  
2 int i = 0;  
3 double summe = 0;  
4 while(i<liste.length)  
5 {  
6     summe = summe + liste[i];  
7     i = i + 1;  
8 }  
9 System.out.println("Mittelwert: " + (summe/i));
```

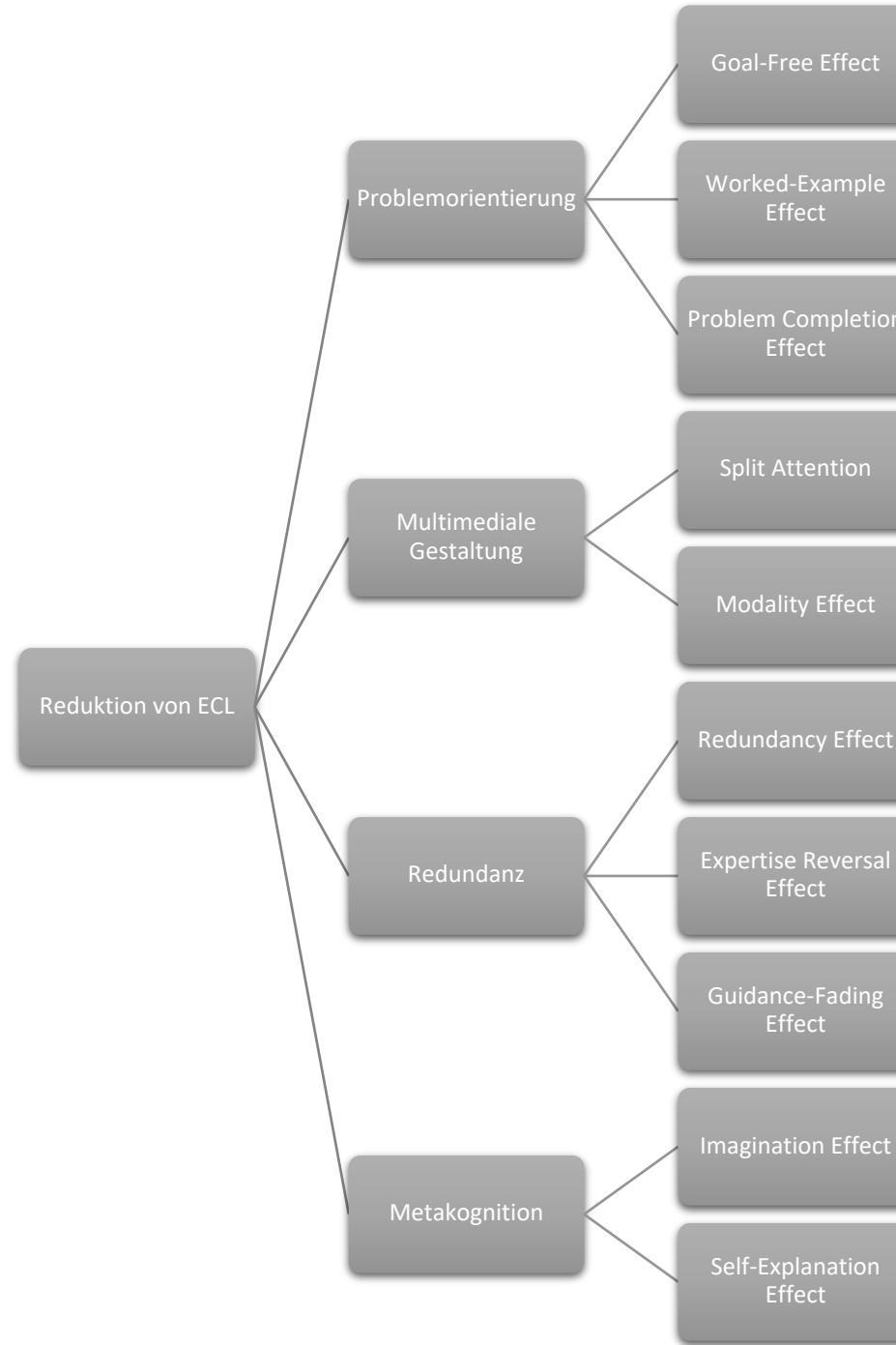
# Gestaltungsempfehlungen für Unterricht

- Element Interactivity Effect
  - Bei hohem ICL -> besonderes Augenmerk auf die Gestaltung der Lernumgebung (ECL) legen, sonst Überbelastung wahrscheinlicher
- Segmentierung
  - ICL didaktisch nicht reduzierbar, kann durch Segmentierung aber handhabbarer gemacht werden
  - Wichtig dabei: sinnvolle Klassifizierung / Strukturierung / Organisation von Lernzielen, Wissensbasis und Aufgaben
  - Aber Achtung: das Große Ganze nicht aus den Augen verlieren!

# Gestaltungs-empfehlungen für Unterricht



# Gestaltungsempfehlungen für Unterricht



# Instruktionsdesigns mit CLT-Bezug

Es gibt zwei wichtige und sehr detailliert beschriebene Instruktionsdesigns, die eine Vielzahl von Gestaltungsempfehlungen der CLT konkret umsetzen:

Cognitive Apprenticeship von Collins, Brown, Newman  
Vierkomponenteninstruktionsdesign (4C/ID) von van Merriënboer