Algorithmenvisualisierung für verteilte Umgebungen

VADE – Visualisation of Alogorithms in Distributed Environments

Inhalt

1. Einleitung

1. Einleitung
    1. Entwickler
    2. Probleme bei der Visualisierung verteilter Systeme
    3. Ziel dieses Projekts (VADE)

1.1 Entwickler

- Die vier Entwickler sind vom Weizman Institute of Sience, Israel
- Wurde beim IEEE Symposium on Information Visualisation (INFOVIS ’9 8) vorgestellt

1.1 Entwickler

- Yoram Moses
- Institut für angewandte Mathematik

- Leonid Ulitsky
- Institut für angewandte Mathematik

1.1 Entwickler

- Zvi Polunsky
- Institut für angewandte Mathematik

- Ayellet Tal
- Institut für Elektrotechnik

1.2 Probleme

- Visualisierung unterstützt Entwicklung, Fehlerbeseitigung und das Lernen von Algorithmen
- Für Verteilte Systeme noch wichtiger
- Problem:
  - Lokal konsistent aber global inkonsistent
  - → Reihenfolge der Ereignisse muss kausal konsistent sein

Höhere Komplexität, Synchronisation, Gleichzeitigkeit

Kein sofortiger Schnappschuss möglich -> möglicher Schnappschuss muss konstruiert werden

1.2. Probleme

- Visualisierungssystem kann die Durchführung nicht exakt wiedergeben
- Aber es kann konsistent zum Durchlauf sein

1.2 Probleme

- Benutzer hat nicht die nötigen Ressourcen
- Der Algorithmus-Code ist nicht geschützt
- Schwierige Anpassbarkeit des Animationssystems an verschiedene Algorithmen

Ressourcen – mehrere Rechner um das System zu simulieren

Jeder der die Visualisierung sehen möchte braucht auch den Quellcode

1.3 Ziele von VADE

- Algorithmus läuft auf Server Rechnern
- Visualisierung auf Client-Seite über eine Webseite mit Java-Applet
- Sehr wenig Kommunikation nötig
- Fertige Bibliotheken um neue Animationen zu erstellen

1+2 dadurch ist der Code geschützt und der Benutzer hat die nötigen Ressourcen um das System laufen zu lassen

2. Benutzeransicht

- Benutzer geht auf eine Internetseite
- Wählt den zu visualisierenden Algorithmus
- Jeder Algorithmus hat eine eigene Seite mit einer oder mehreren Ansichten und Kontrollwerkzeugen

2. Benutzeransicht

- Kontrollwerkzeuge:
  - Starten und Pausieren der Animation
  - Festlegen des Startknotens für den Algorithmus
  - Konfigurieren des Netzwerkes

-Add node und Add Edge um weitere Knoten und Kanten hinzuzufügen

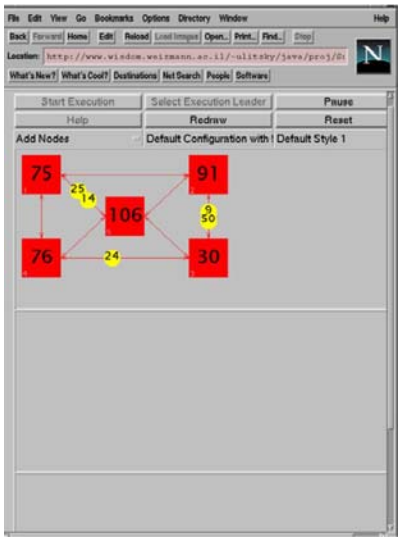
-Fall keine Konfiguration angegeben wird eine Standardkonfiguration verwendet

2. Benutzeransicht

- Beispiel: Schnappschuss-Algorithmus
  - Knoten verschicken zufällig Daten untereinander
  - Ein Prozess beginnt den Schnappschuss und verschickt eine Markierung an alle Nachbarn
  - Fährt dann fort mit dem Austausch, speichert aber ankommende Daten von jedem Kanal, bis Markierung auf diesem ankommt
  - Knoten beginnen beim erhalt einer Markierung ebenfalls mit dem Schnappschuss
  - Wenn Markierungen von allen Nachbarn erhalten, ist der Schnappschuss beendet

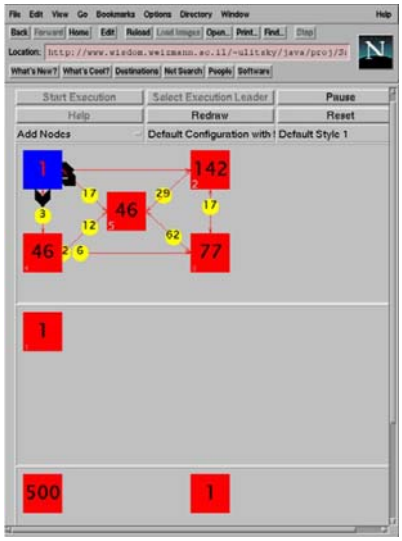
2. Benutzeransicht

- Knoten (rot) tauschen regulär Daten (gelb) untereinander aus



2. Benutzeransicht

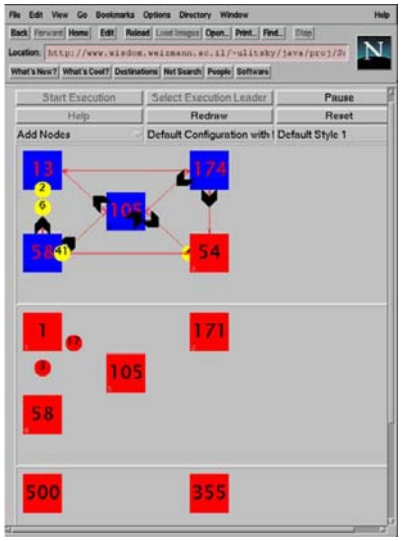
- Knoten (blau) beginnt mit dem Schnappschuss
- Schickt Markierungen (Pfeile) an seine Nachbarn



1. Ansicht – Netzwerk mit Animation der versendeten Daten und Status der Knoten
2. Ansicht – Derzeitiger Status des Schnappschusses
3. Ansicht – links Datenmenge insgesamt, rechts Datenmenge bei derzeitigem Schnappschuss (sollten gleich sein wenn Algorithmus beendet)

2. Benutzeransicht

- Weitere Knoten haben Markierung erhalten und versenden selbst Markierungen
- Nachrichten werden im Schnappschuss gesichert bis Markierung auf entsprechendem Kanal eintrifft



2. Benutzeransicht

- Knoten (grün) haben Schnappschuss-Algorithmus beendet
- Summe der Daten in Nachrichten und Knoten stimmt mit Gesamtsumme überein

3. Modell

- Vorraussetzungen:
  - Kommunikationsnetzwerk ist zuverlässig
  - Nachrichten die von einem Prozess gesendet werden, kommen in der selben Reihenfolge an, in der sie gesendet wurden (FIFO)
  - Netzwerk ist asynchron (keine globale Uhr)

-Jede abgeschickte Nachricht wird irgendwann ihr Ziel erreichen

-Jeder Prozess hat seine eigene Uhr … es gibt keine globale Uhr

-Keine Zusicherungen über Geschwindigkeiten der Prozesse oder Geschwindigkeit der Datenübertragung

3. Modell

- Das System besteht aus einer endlichen Anzahl Prozessen (1, …, n)
- Jedes benachbarte Paar i, j hat Kommunikationskanäle c(i,j) und c(j, i)
- Jeder Prozess hat einen wohl-definierten lokalen Zustand s
- Jeder Prozess kann zwei Arten von Aktionen ausführen:
  - send(i, j, m) senden der Nachricht m von i zu j
  - internal(i, a) interne Berechnung, die lokalen Zustand ändern kann

C(i, j) – Nachrichten von i nach j

C(j, i) - umgekehrt

3. Modell

- Jeder Prozess befindet sich zu Beginn in einem initialen Zustand in dem nur eine wakeup ausgeführt werden kann
  - → selbe Situation wie im Animationssystem, bevor es von den Prozessen weiß
- Es gibt ein Steuerprogramm, dass die Auslieferung der Nachrichten übernimmt
  - Führt Aktion receive(j, i, m) aus
- → Zustand von j ändert sich und Nachricht m wird aus dem Kommunikationskanal gelöscht
- Eine Aktion tritt immer bei dem Prozess auf, dessen Zustand er ändert

- also sind send(i, j, m), internal(i, a) und receive(i, j, m) treten bei Prozess i auf

3. Modell

- Aktionen nicht atomar, sondern um Nebenläufigkeit und Animation darzustellen besteht eine Aktion aus zwei Ereignissen:
  - 1.Ereignis – Einleitung der Aktion
  - 2. Ereignis – Beendigung der Aktion

3. Modell

- Globaler Status (Konfiguration) ist ein Tupel
  $(s_1, \ldots, s_n, c_{i_1j_1}, \ldots, c_{i_kj_k},)$
- Durch jedes Ereignis entsteht eine neue Konfiguration des Systems

S – Zustand für jeden Prozess

C – Zustand der Kanäle

3. Modell

- Modell eines Durchlaufs *r* durch eine Sequenz von Ereignissen mit folgenden Eigenschaften:
  1. Für jeden Prozess i stellen alle Ereignisse die bei i ausgeführt wurden und in *r* enthalten sind, dessen lokale Geschichte dar
  2. Ein receive(j, i, m) erscheint in *r* nur nach dem zugehörigen send(i, j, m)
  3. Zu jeden send(i, j, m) existiert das zugehörige receive(j, i, m) in *r*
  4. Für jedes Paar benachbarter Prozesse werden deren Nachrichten in der selben Reihenfolge geliefert, in der sie gesendet wurden

Zu 2) Nachrichten können nicht ausgeliefert werden, nachdem sie verschickt wurden

Zu 3) Kommunikationskanäle sind zuverlässig

Zu 4) FIFO

System ist asynchron, da jede Ausführung, der diese Kriterien erfüllt ein Durchlauf r des Algorithmus ist

3. Modell

- Visualisierungsprozess kann nicht die Reihenfolge der Ausführung von jedem Paar von Aktionen, die von verschiedenen Prozessen ausgeführt worden, wissen
- →Potentielle Abhängigkeit

Daher wird der Begriff der potentiellen Kausalität benutzt
(happened before Relation)

3. Modell

- Definition:
  Ereignis e‘ ist potentiell abhängig von e (e ~> e‘) wenn folgendes gilt:
  1. e trat vor e‘ in der lokalen Geschichte desselben Prozesses auf
  2. e und e‘ sind auf verschiedenen Prozessen, die durch Kommunikationskanal miteinander verbunden so ist e send() und e‘ receive()
  3. es existiert ein Ereignis e‘‘, so dass e ~> e‘‘ und e‘‘ ~> e‘

d.h. immer wenn e ein e‘ verursacht, sind sie potentiell abhängig

Jeder Durchlauf der durch die Reihenfolge (Ordnung) die durch potentialle Abhängigkeit verursacht wird, ist konsistent.

3. Modell

- Potentielle Abhängigkeit ist nur eine Annäherung an die kausale Abhängigkeit, da sie auch Ereignisse in eine Reihenfolge bringt die kausal oder sogar konzeptionell unabhängig sind
- → semantische Abhängigkeit

z.b. unnötig die Animation von Aktionen die nach einem send() geschehen aufzuschieben bis die sende-empfangs animation fertig ist

3. Modell

- Definition:
  Ereignis e‘ ist semantisch abhängig von e
  (e → e‘) wenn folgendes gilt:
  1. e trat vor e‘ in der lokalen Geschichte desselben Prozesses auf und sie sind semantisch nicht unabhängig
  2. e und e‘ sind auf verschiedenen Prozessen, die durch Kommunikationskanal miteinander verbunden so ist e send() und e‘ receive()
  3. es existiert ein Ereignis e‘‘, so dass e → e‘‘ und e‘‘ → e‘

Potentielle Abhängigkeit kann automatisch festgestellt werden, semantische nicht

Programmierer muss zusätzliche Informationen über Unabhängigkeit von Aktionen bereitstellen

3. Modell

- Definition:
  Menge Con(*r*) von Durchläufen ist semantisch konsistent mit *r* wenn folgendes gilt:
  1. Die selbe Menge von Ereignissen finden in *r* und *r*‘ statt
  2. Für jedes Paar von Ereignissen e und e‘ gilt, dass wenn e → e‘ , e findet vor e‘ in *r*‘ statt

d.h. es ist möglich die Reihenfolge von Ereignissen zu ändern, so dass der Durchlauf immer noch konsistent ist

Außerdem ermöglicht es gleichzeitige Ausführung/Animation von Ereignissen

3. Modell

- Animationsystem kann Modell der Ausführung des Algorithmus (*E*) erstellen und nicht nur die gemeldeten Ereignisse in eintreffender Reihenfolge darstellen

3. Modell

- Definition:
  Sei $F=<F_0, F_1, \ldots, F_t>$ eine Animation von *r* und $E=<E_0, E_1, \ldots, E_t>$ eine Sequenz von Modellen um die Animation zu konstruieren. Die Animation F ist konsistent mit *r*, genau dann wenn es ein *r*‘ e Con(*r*) gibt, so dass die Sequenz E eine Teilsequenz der Sequenz von golbalen Zuständen *r*‘ ist.

Jedes Frame F basiert auf Modell E

Animationssystem animiert nicht alle Ereignisse die geschehen, daher Teilsequenz

3. Modell

- Sei An(e) die Animation des Ereignisses e und An(e) < An(e‘) bedeutet, An(e) ist beendet bevor An(e‘ beginnt).

- Eine Animation ist konsistent mit der Ausführung eines Algorithmus genau dann wenn für jedes Paar von Ereignissen e und e‘ gilt, wenn e → e‘ dann An(e) < An(e‘).

Beweis …

3. Modell

- Implementierung der kausalen Ordnung:
  - Ausreichen, wenn für Paare implementiert, da transitiv
  - Ereignisse die nur in einem Prozess stattfinden werden sofort hintereinander animiert
  - Bei send() + receive() 2 Möglichkeiten:
    - Sendesynchronisation
    - Empfangssynchronisation

3. Modell

- Sendesynchronisation:
  - Dem Animationssystem wird das Senden einer Nachricht vor dem eigentlichen senden mitgeteilt und anschließend gewartet, bis das Animationssystem bestätigt.
  - Das Empfangen wird sofort nach dem Empfang dem Animationssystem gemeldet
  - Dadurch ist sichergestellt, dass das Senden vor dem Empfangen animiert wird

Nachteil: Ausführung wird vom Animationssystem beeinflusst

Trotzdem gut für Lernvorführungen … nicht zum debuggen

3. Modell

- Empfangssychronisation:
  - Synchronisation wird vom Animationsystem durchgeführt
  - Senden und empfangen wird erst nach der Ausführung der Aktion gemeldet
  - Animation wird erst ausgeführt, wenn zum Senden ein entsprechendes Empfangen gemeldet wurde
  - Implementation:
    - 2 Counter für jeden Kommunikationskanal, die bei jedem Senden und Empfangen erhöht werden
    - Empfang wird nur animiert, wenn Anzahl der Sendungen des Nachbarn größer als Anzahl der Empfangsereignisse

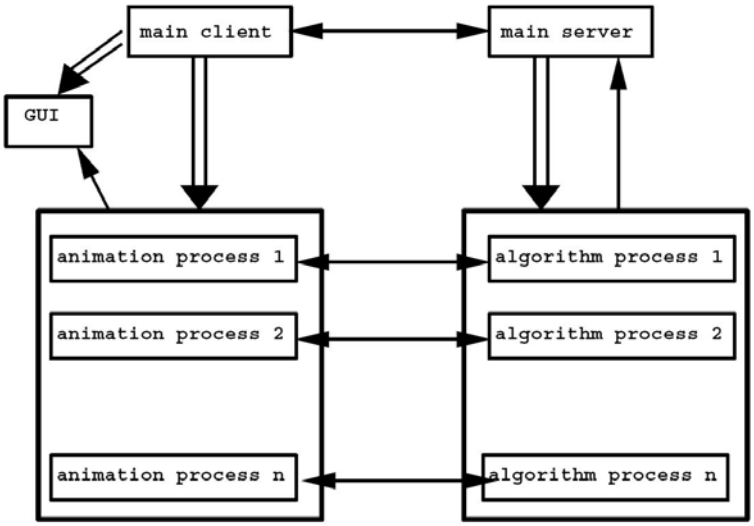
Vorteil: Algorithmusausführung unabhängig von der Animation

Nachteil: viele Meldungen müssen möglicherweise aufgeschoben werden

4. VADE Architektur

- Der Algorithmus läuft auf dem Server als Java Applikation
- Visualisierung läuft auf dem Client über den Browser als Java Applet

4. VADE Architektur



--> Kommunikationsrichtung

==> Abspalten von Prozessen als Threads

Applet wird gedownloaded und ausgeführt

Main Client startet GUI und informiert Main Server über Art des Algorithmus der ausgeführt werden soll

Client spaltet Animationsprozesse ab, diese informieren die GUI über Ereignisse

Main Server kennt Tabelle von zur Verfügung stehenden Rechnern und startet auf diesen den Algorithmus Prozess

Algorithmus Prozess stellt Verbindung zu Animationsprozess her

Nach beenden des Algorithmus Prozesses wird dies dem Main Server gemeldet

5. Programmiereransicht

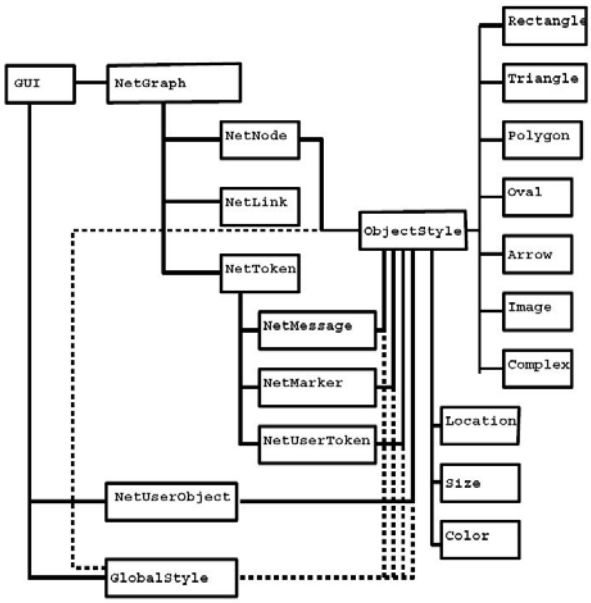
5 Schritte:

1. Identifizieren der für die Visualisierung interessanten Ereignisse
2. Implementieren des Algorithmus
3. Hinzufügen der Aufrufe an das Animationssystem (z.B. bei Sende- und Empfangsereignissen)
4. Implementieren der Animation
5. Erstellen der Webseite

5. Programmiereransicht

- VADE nimmt große Teile der Arbeit für Schritt 4 und 5 ab
  - Die GUI und einige Bibliotheken stehen bereits zur Verfügung
  - Der Programmierer braucht sich nicht um die Konsistenz der Animation zu kümmern
  - Vorgefertigte Webseite muss nur noch angepasst werden

5. Programmiereransicht

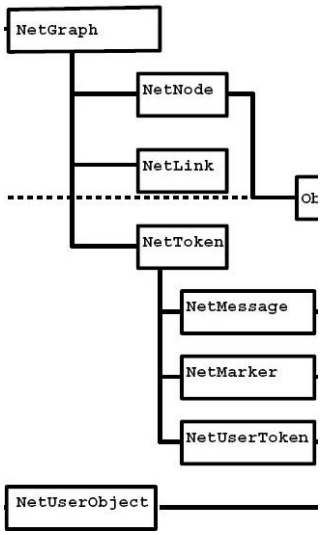


5. Programmiereransicht

- Das Framework unterscheidet zwischen dem Inhalt der angezeigt wird und der Art und Weise wie die Objekte visualisiert werden sollen
- Programmierer muss sich zunächst nur auf den Inhalt konzentrieren, da Voreinstellungen für die Visualisierung meist ausreichend sind (z.B. zum Debuggen)
- Visualisierungsoptionen können später separat geändert werden

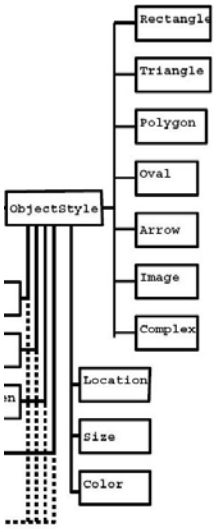
5. Programmiereransicht

- NetGraph – bildet den gesamten Graphen
- NetNode – stellt einen Knoten dar
- NetLink – stellt einen Kommunikationskanal dar
- NetToken – stellt die Nachrichten dar



5. Programmiereransicht

- GlobalStyle – legt allgemein das Aussehen fast (Hintergrund, Animationsgeschwindigkeit, Farben)
- ObjectStyle überschreibt das globale Aussehen für ein spezielles Objekt



6. Fallstudie

- ATS – Automatic Transport System
- Spezifikation:
  - ...

6. Fallstudie

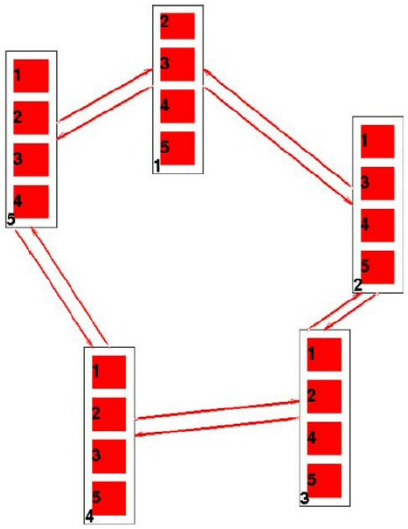
- ...

6. Fallstudie

- Objektrepräsentation
  - ...

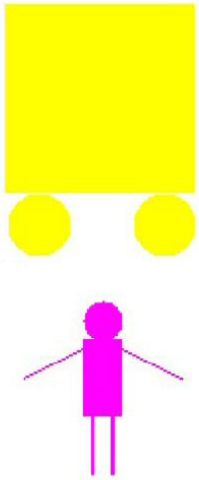
6. Fallstudie

- Knoten
- …



6. Fallstudie

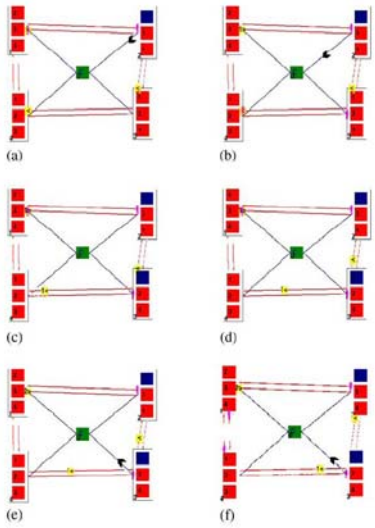
- PKW
- Passagier
- ...



6. Fallstudie

- … Erklärung

6. Fallstudie



6. Fallstudie

- … mehr Erklärung

6. Fallstudie

```
Public void setAnimationActions(){
    animDataTable = new AnimData[9];

    animDataTable[0] = new AnimData("PressButton",
                                    new TrainPressButton() );
    animDataTable[1] = new AnimData("SendCarToterminal",
                                    new TrainSendCartoTerminal() );

    ........

    animDataTable[8] = new AnimData("ReceiveMarker",
                                    new TrainReceiveMarker() );
}
```

6. Fallstudie

```
import ui.*;
import geometry.*;
import network.NetProtocol;
import java.awt.*;

public class TrainPressButton implements AnimationAction {
    static TrainAnimCommonData commData;

    public String perform(StringBuffer args[], int argsNum,AnimationProcess ap){

        if (argsNum != 1){
            return NetProtocol.BAD_ARGUMENTS_NUM_REPLY;
        }
        String neighbour = new String(args[0]);
        NetGraph graph = ap.parent.framedArea[0].GUIScreen.graph;
        int idx = graph.getNode(ap.processIndex).index.intValue();

        NetNode nd = commData.getCurrentNode(graph, neighbour, idx);
        nd.setState(commData.BUTTON_PRESSED, graph.getAnimStyle());

         ap.parent.framedArea[0].GUIScreen.paint(
                    ap.parent.framedArea[0].GUIScreen.getGraphics() );

        return NetProtocol.OK_REPLY;
    }
}
```

6. Fallstudie

```
TrainGeomObjectComplexCar (Point location, int size1, int size2, Color color) {

        super(size1, location );

        GeomObject components[] = new GeomObject[3];

        Point location1 = new Point(location.x - size1/3,
                                    location.y + size2/2 + size1/6);
         Point location2 = new Point(location.x + size1/3,
                                    location.y + size2/2 + size1/6);

        components[0] = new GeomObjectRectangle(size1, size2, location);
        components[1] = new GeomObjectCircle(size1/3, location1);
        components[2] = new GeomObjectCircle(size1/3, location2);

        create(components);

        setColor(color);

    }
```

6. Fallstudie

- Erläuterung

7. Kritik

- Vorteile:
- ...

7. Kritik

- Nachteile:
- …

7. Kritik

- Fazit:
- ...