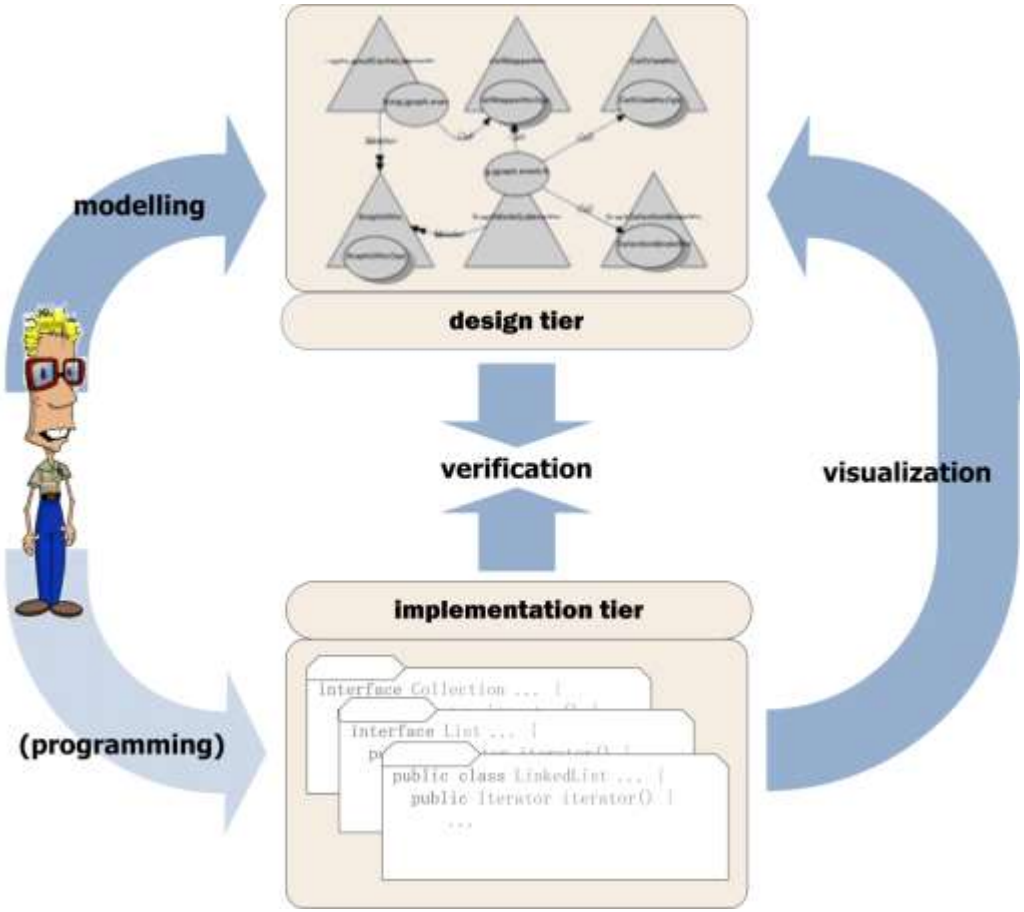
Modeling and visualizing object-oriented programs with codecharts



Hastings Research Seminar
Wednesday 14 November 2012

Paper details

- Based on a journal paper submitted to **Formal Methods in System Design**

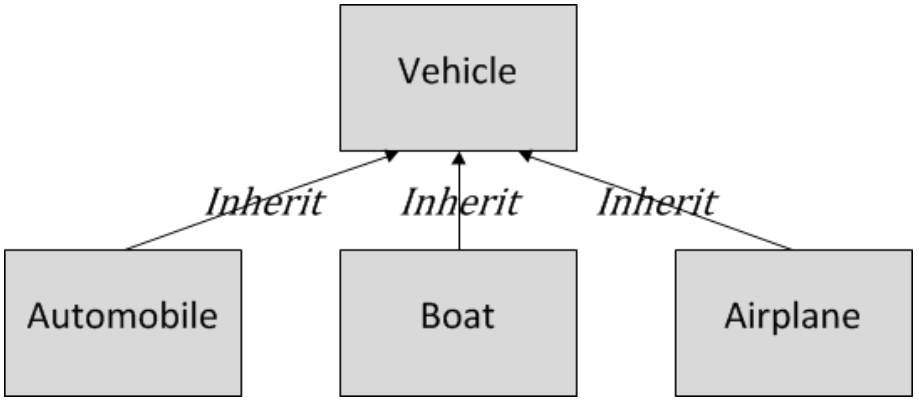
- Amnon H. Eden
  - School of Computer Science & Electronic Engineering, University of Essex
- Epameinondas (Notis) Gasparis
  - School of Computer Science & Electronic Engineering, University of Essex
- Jon Nicholson
  - School of Computing, Engineering and Mathematics, University of Brighton
- Rick Kazman
  - University of Hawaii, and
  - Software Engineering Institute, Carnegie-Mellon University

Software

- We've been writing software for decades
  - Ada Lovelace (1815-1852) considered the first "programmer"
- There are many different kinds of programming methodologies
  - Procedural, logic, object-oriented, etc.
  - Object-oriented is a very popular one …
  - … that has been around for half a century

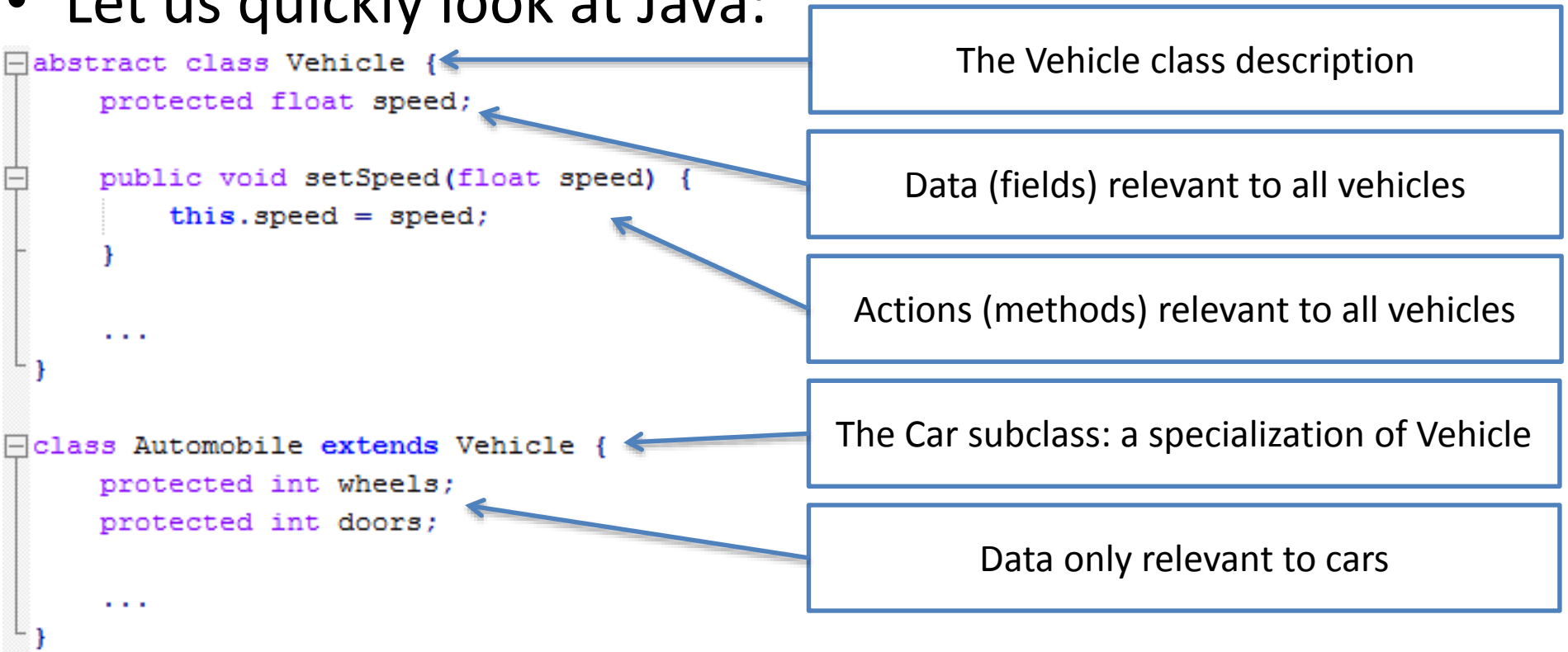
What are object-oriented programs?

- **Objects**
  - Represent 'things' from the real world or some problem domain
  - For example: people, vehicles, numbers
  - Have actions (*methods*) and data (*fields*)
  - Focus on *statically typed class-based* object-oriented languages
- **Classes**
  - Represent all objects of a particular kind (*type*)
  - For example: Student, Car, Integer
  - Objects are *instances* of classes



Object-oriented languages

- There are numerous statically typed class-based object-oriented languages
  - E.g. Java, C++, C#, Smalltalk, Objective-C, …
- Let us quickly look at Java:



Software lifecycle

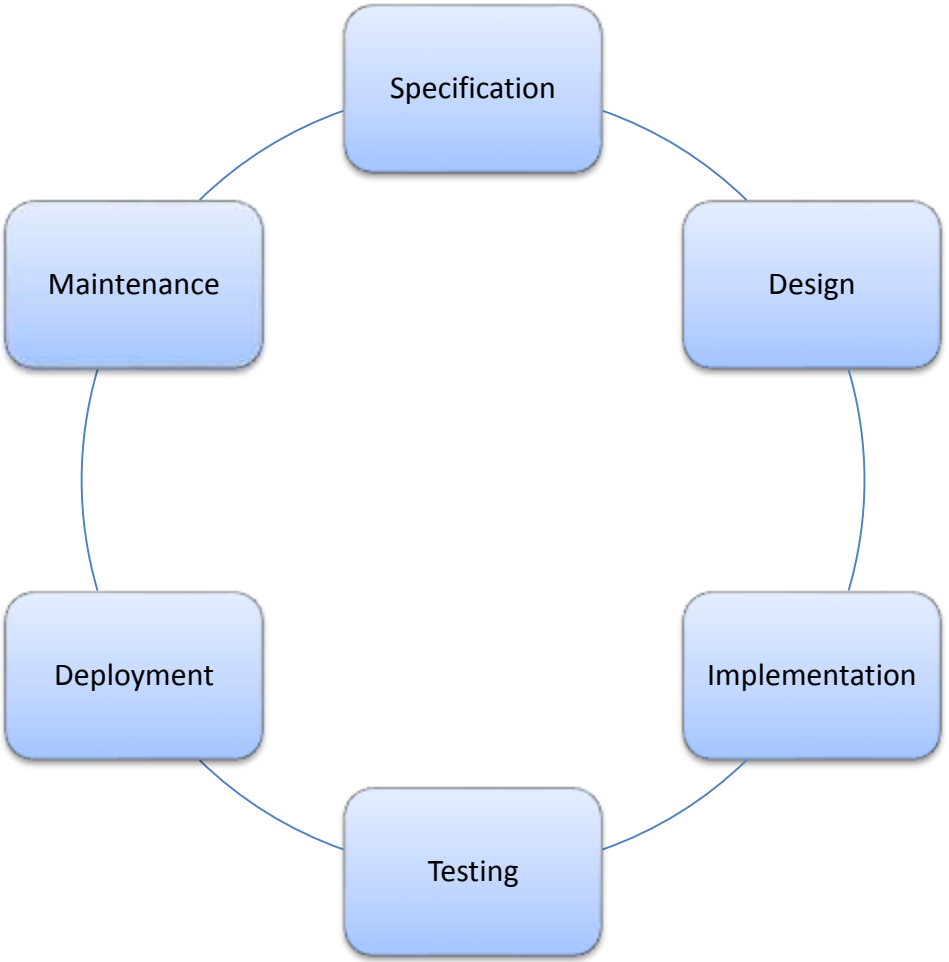
Lots of models for software development exist, e.g.

- Waterfall model
- Spiral model
- Metropolis model

The lifecycle of software is generally composed of stages (right)

They are not all equal…

- Some projects never finish the specification/design
- Testing is one of the most costly
- Maintenance can last decades
  - IE6 released 27 Aug 2001, but is still being supported on Windows XP until 8 Apr 2014



So what's the problem?

- We've been writing software for years…
- … surely we're good at it!

Software's chronic crisis

- Software is hard!
  - Software projects are growing ever bigger and more complex
  - We have huge amounts of legacy software we need to maintain
  - Software projects are often over budget and behind schedule
  - Many software projects never get released
  - Error ridden software has caused human deaths
- We accept more errors in software than other engineering disciplines
  - The bridge fell down, but it's ok…
    - Turn it off and on again
    - It'll be fixed in the next version
- We are still in the middle of **software's chronic crisis**
  - W. Gibbs, Software's chronic crisis, Scientific American 271 (3) (1994) 72–81.

"a research innovation typically requires 18 years to wend its way into the repertoire of standard programming techniques"

W. Gibbs, Software's chronic crisis, Scientific American 271 (3) (1994) 72–81.

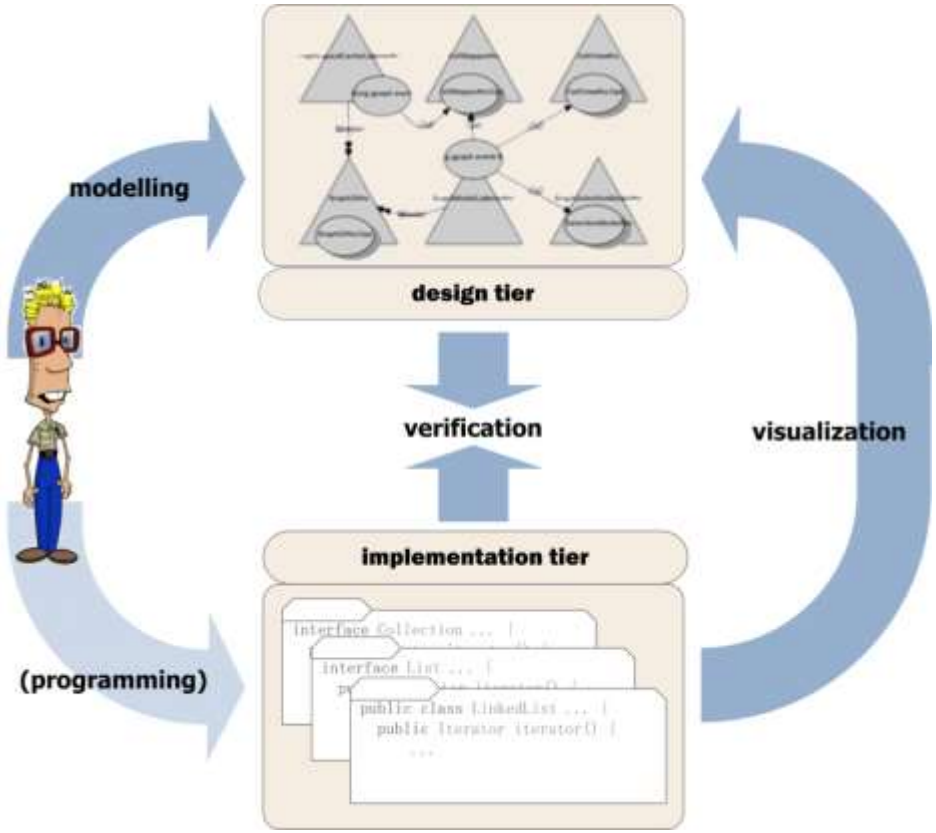
Why is it so hard?

- **Complexity**
  - Software systems constitute some of the most complex artifacts ever manufactured
- **Invisibility**
  - Software is intangible, making it hard to visualize and particularly difficult to detect and resolve its design flaws
- **Conformance**
  - Enforcing conformance to design decisions is largely an open problem. Manual verification, if at all possible, is largely impractical
- **Changeability**
  - Software and its documentation must be continuously evolved to meet ever changing requirements. Developers engage in software modeling and visualization throughout the project's lifecycle, for example by alternating use of forward and reverse engineering tools

- F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer Magazine*, vol. 20, no. 4, pp. 10–19, Apr-1987.
- M. M. Lehman, "Laws of Software Evolution Revisited," in *Proc. 5th European Workshop Software Process Technology—EWSPT'96*, Nancy, France, 1996.
  - "A program must be continuously adapted or else it becomes progressively less satisfactory"
  - "As a program evolves its complexity increases unless work is done to maintain it"

The state of the art

- Lots of software design languages exist
  - Statecharts, UML, DPML, PADL, Z, RBML, …
- Each with their own emphasis, benefits and limitations
- Few are designed for both forward and reverse engineering tasks
- Few offer conclusive automated design verification



What do we need?

(In our opinion…)

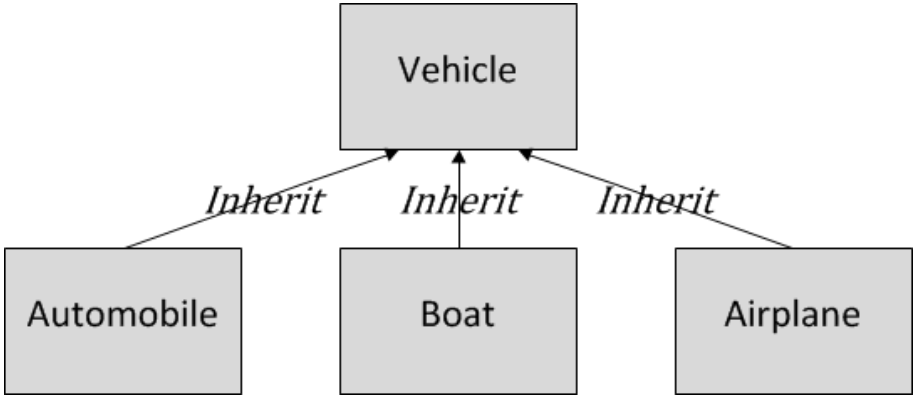
- A rigorously defined design language that …
  - is **visual** and intuitive to use
  - takes a theoretically sound, but **pragmatic** approach
  - says only what it needs to (**appropriately abstract**)
  - describes **patterns** as easily as specific programs
  - works on programs of any size (**scalable**)
  - can be used in **forward *and* reverse engineering**
    - I.e. that closes the re-engineering gap
  - focus on **object-orientated** programs
  - emphasises effective **tool support** (e.g. design verification)

Complexity: ontology

- Codecharts have a focused ontology for object-oriented programs
  - Elementary OO components:
    - Classes, methods and method signatures
  - Sets of elementary OO components
    - Must be finite, but can contain any number of elements
      - e.g. thousands of classes in a single symbol
    - Particularly interested in inheritance class hierarchies and sets of dynamically bound methods
  - Properties
    - E.g. Class x is "abstract"
  - Relationships
    - E.g. Class x "inherits" from class y

Complexity: visuality

- Our brains have natural aptitude for spatial reasoning
- We are very good at processing images/diagrams
- Diagrams can convey information more concisely and precisely
- Often requiring less training than symbolic (mathematical) notations



Vs.

$$
\begin{aligned}
&Class(\text{Vehicle}) \wedge \\
&Class(\text{Automobile}) \wedge \\
&Class(\text{Boat}) \wedge \\
&Class(\text{Airplane}) \wedge \\
&Inherit(\text{Automobile}, \text{Vehicle}) \wedge \\
&Inherit(\text{Boat}, \text{Vehicle}) \wedge \\
&Inherit(\text{Airplane}, \text{Vehicle})
\end{aligned}
$$

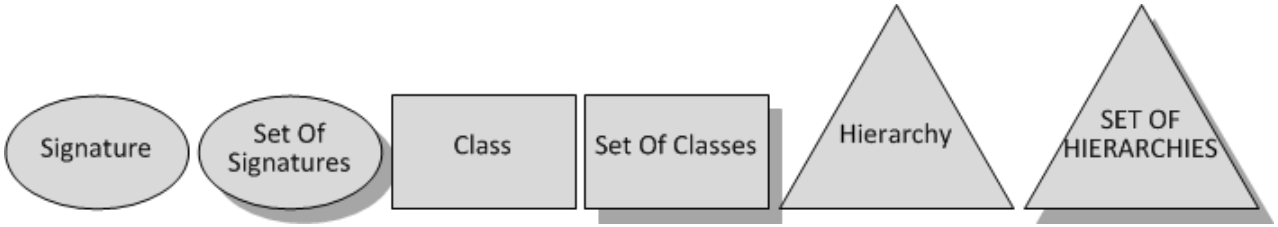
- NB: We believe the language of codecharts is a "good" visual language, i.e. easy to use/intuitive/etc., but an empirical study is needed to show that really is the case

E. Tufte, *Visual Explanations: Images and Quantities, Evidence and Narrative*. Cheshire, CT: Graphics Press, 1997.

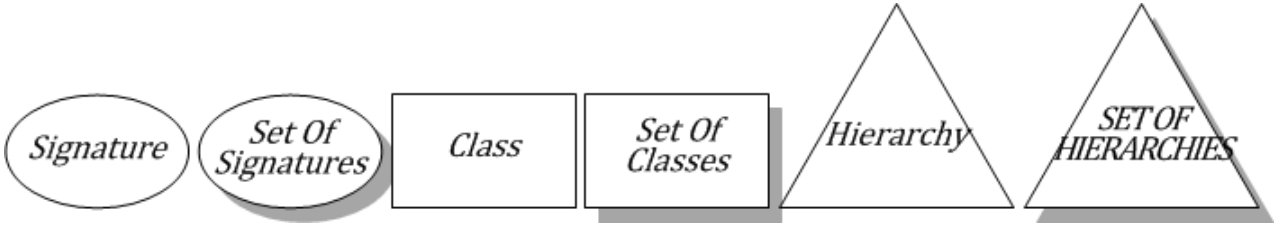
D. L. Moody, *The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering*. IEEE Transactions on Software Engineering, Vol. 35, No. 5, November-December 2009

Complexity: vocabulary

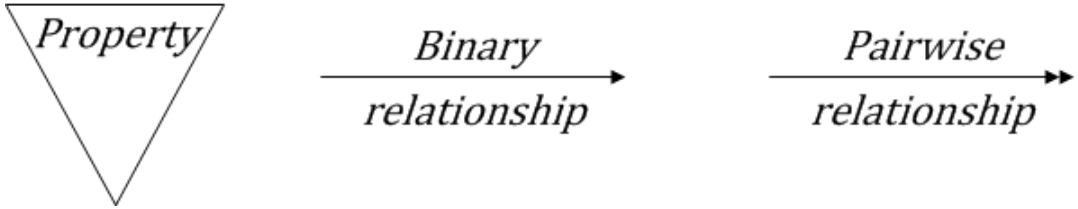
- Codecharts have a minimal (uncluttered) vocabulary tailored specifically for object-oriented program structure
- The main symbols represent specific bits of programs:



- Which are mirrored by those that represent placeholders (i.e. for patterns and frameworks):



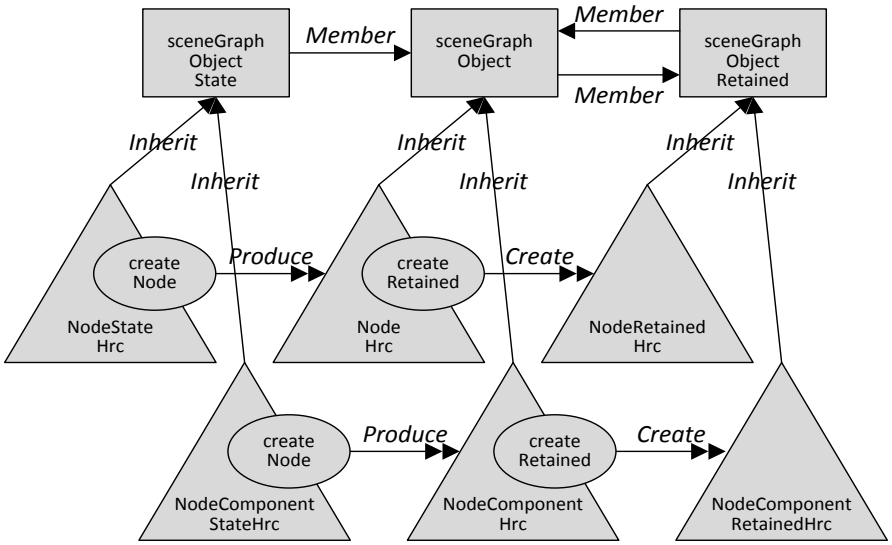
- And a small set of properties/relationship symbols:



Example

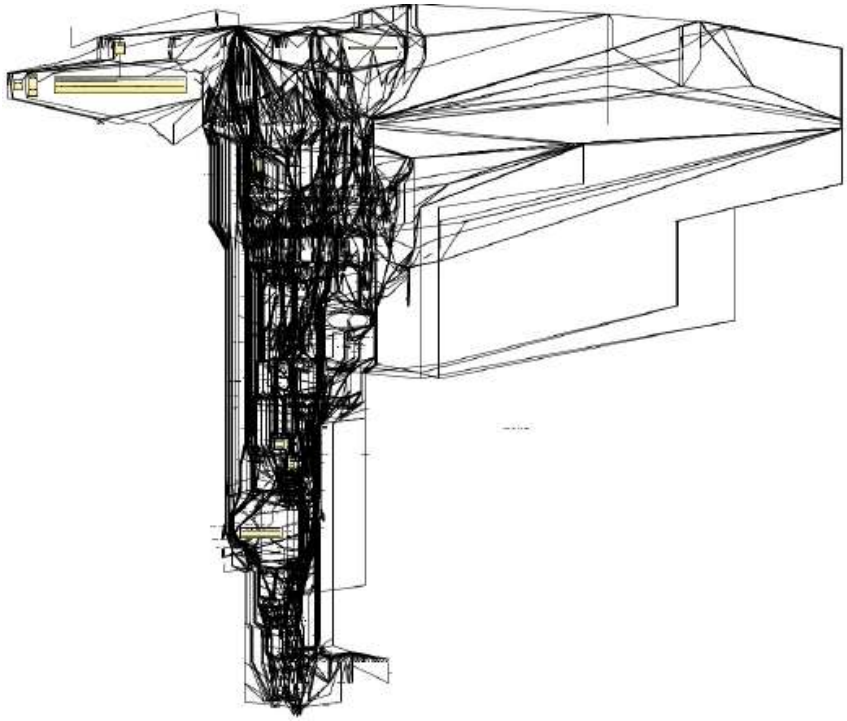
Java3D as a codechart

As generated by the Toolkit



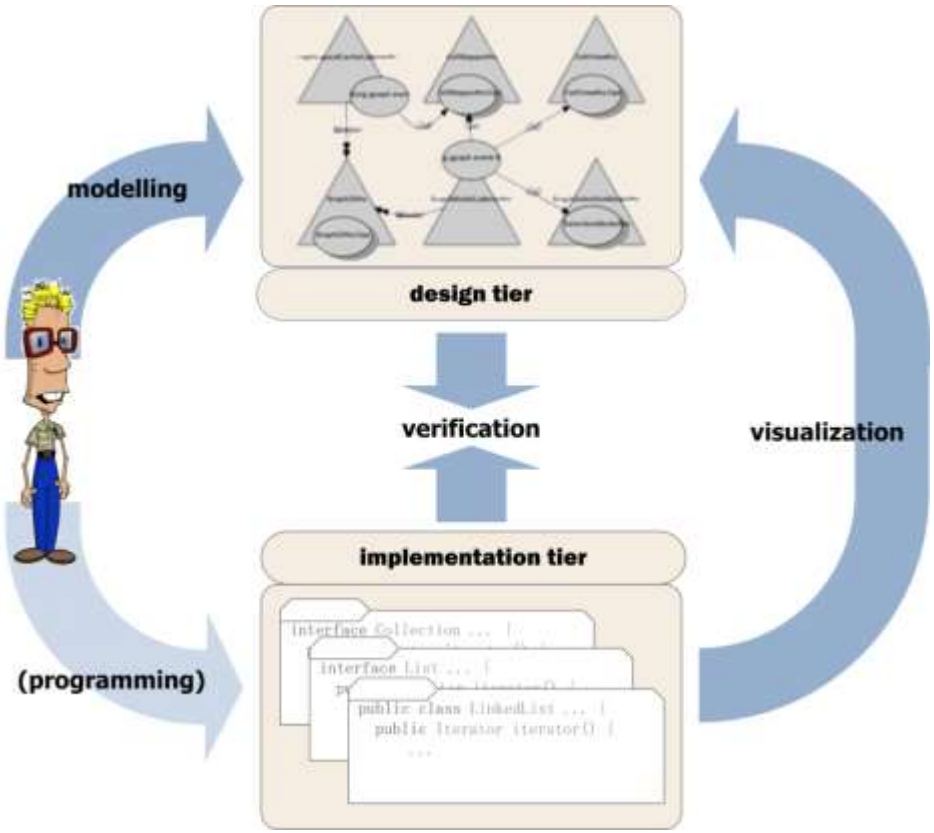
Java3D class diagram

As generated by NetBeans 6.1



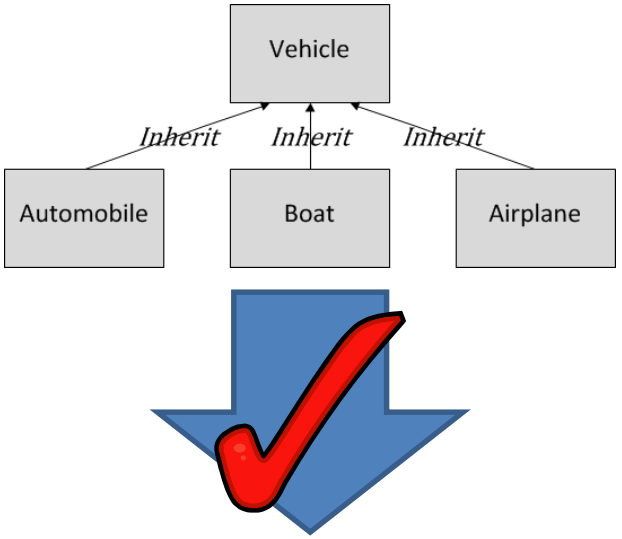
Invisibility

- Codecharts can be used for modeling:
  - Programs
  - Frameworks/libraries
  - Design patterns
- A codechart can be created manually or generated from a program
  - i.e. modeling and visualization
  - Generated codecharts can be used in the same way as manually created ones
  - Visualization is quick and user guided: the user has control
  - Helps users "get a feel" for the code: making it more tangible
  - It closes the re-engineering gap



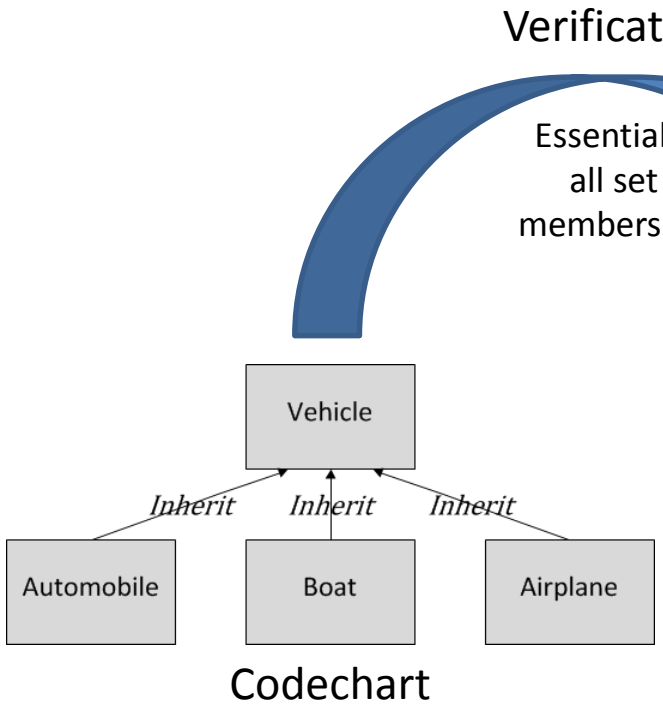
Conformance

- Checking that a program **conforms** to a specification
- "Software Verification"
  - Traditionally this is the verification that a program functions as expected
  - I.e. a program given some input responds with the expected output
- Software verification is too strong a notion
  - In the general case, automated software verification is undecidable
  - When it is possible it requires special training and lots of effort
- Instead we do **design verification**
  - Verifying only those properties that are decidable
  - I.e. there exists an effective method to conclusively establish a statement as being true or false
  - Codecharts only articulate decidable properties
  - A pragmatic approach to software verification
  - Possible (in principle) to provide automated tool support




```
abstract class Vehicle {
    protected float speed;

    public void setSpeed(float speed) {
        this.speed = speed;
    }

    ...
}

class Automobile extends Vehicle {
    protected int wheels;
    protected int doors;

    ...
}
```

A little technical detail…



Design model

```
abstract class Vehicle {
    protected float speed;

    public void setSpeed(float speed) {
        this.speed = speed;
    }

    ...
}

class Automobile extends Vehicle {
    protected int wheels;
    protected int doors;

    ...
}
```

Program

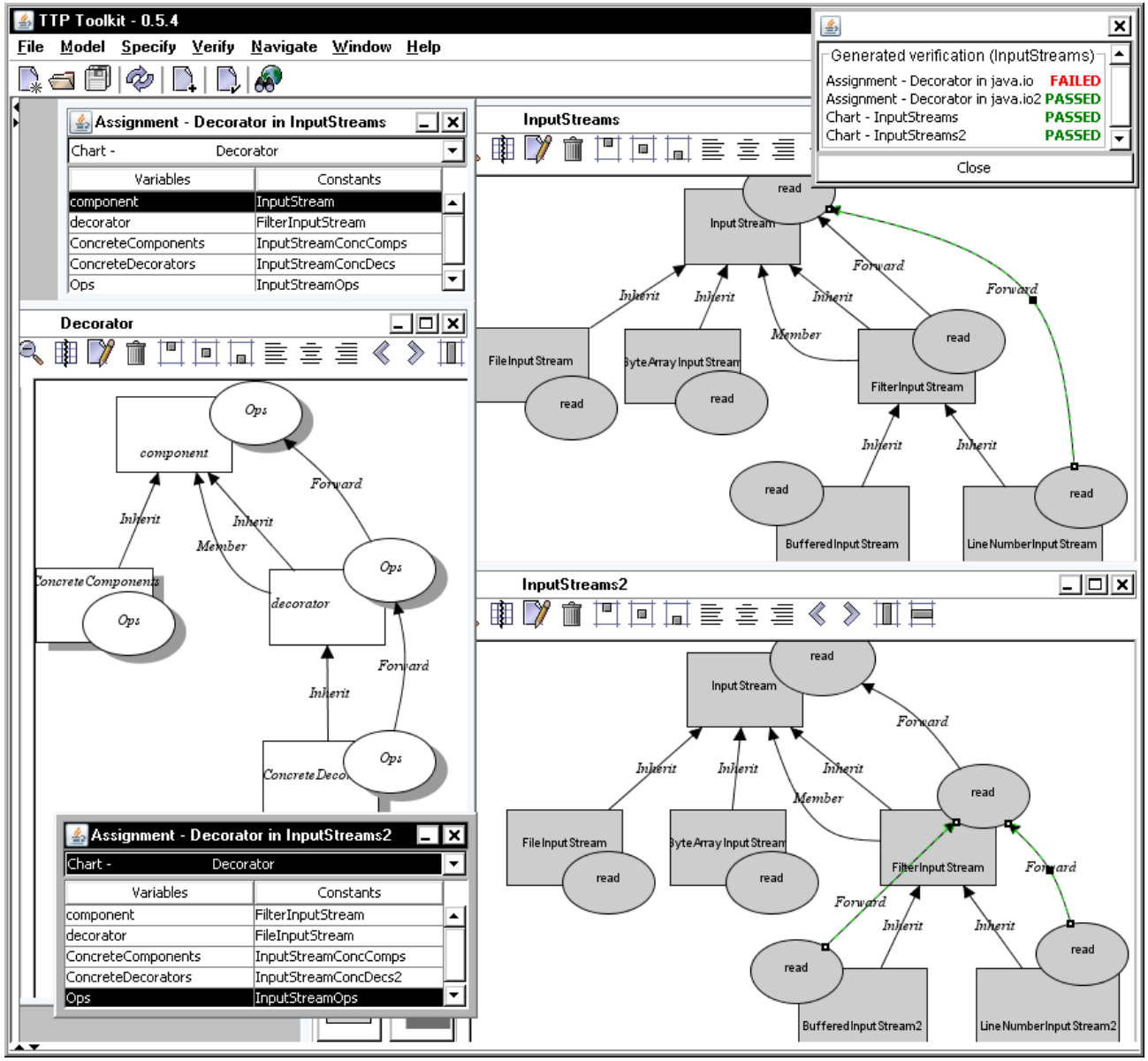
Abstract semantics function

Produces a "database of facts"

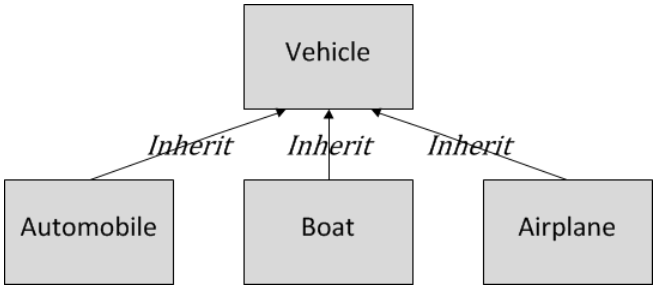
Consequently…

- The codechart language has a very **restricted scope** …
  - Targets a specific niche of software development
  - E.g. no behavioural or temporal information
  - Not designed to answer questions like:
    - Does the program terminate?
    - Does the program operate as expected?
- … and therefore **relatively inexpressive** …
  - But expressive enough to articulate program structure
  - I.e. key dependencies between program components
  - Designed to answer questions like:
    - What dependencies does this method have with that one?
    - Is the program structured as expected?
- … meaning that they are **computationally lightweight**
  - Reasoning about the relationship between programs and codecharts is tractable in principle

And we've done it!



Changability



Codechart

```
abstract class Vehicle {
    protected float speed;

    public void setSpeed(float speed) {
        this.speed = speed;
    }

    ...
}

class Automobile extends Vehicle {
    protected int wheels;
    protected int doors;

    ...
}
```

Program

Changeability

- Software should evolve to meet ever changing requirements…
  - Adding/removing functionality
  - Changing the environment (e.g. from WinXP to Win7)
- …but documentation should evolve with the software

- Design verification can help:
  - It's quick, can be done at the click of a button
  - It's conclusive, either it passes of fails
  - It's targeted, clearly states where the inconsistency is
    - Early identification of issues is key
    - Is it the code or design that should change? Maybe both?
- Codecharts can be used at any point in the software lifecycle
  - Doesn't exclude use of other languages (it's complimentary)
  - Doesn't require modifying the code (e.g. special code comments)

Maybe codecharts will make it easier…

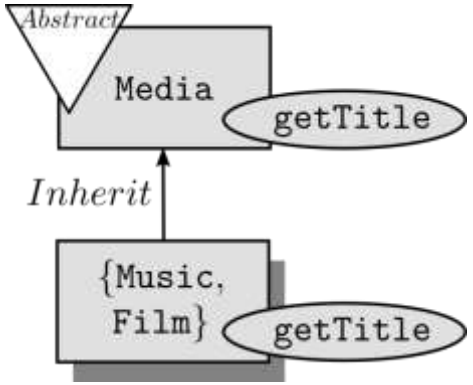
- **Complexity**
  - Codecharts have a focused ontology for object-oriented programs
  - Codecharts are visual, conveying complex information more concisely
  - Codecharts have a minimal (uncluttered) vocabulary
- **Invisibility**
  - Codecharts can be used for modeling and visualizing programs
    - I.e. closing the re-engineering gap
  - Quick, user driven, program visualization helps to make software more tangible
- **Conformance**
  - Codecharts take a pragmatic approach with an emphasis on tool support
  - The problem of verifying that a program satisfies a codechart is tractable in principle
- **Changeability**
  - Verification is fast and conclusive, can be run often to identify issues quickly
  - Codecharts can be introduced any time during the software lifecycle
  - Doesn't change the source code and can be dropped at any time

So what have I done recently?

- My PhD work just looked at the logic behind codecharts
  - I used it as a foundation for theory of classes
  - Identified a few issues (solved some of them…)
- In working with the VMG we identified some further issues:
  - Codecharts are "formal", but not formal enough
  - From the visual modelling community view, the syntax is not formally defined
  - If the syntax isn't formally defined then neither is the semantics
- We now have a formal version of the syntax

A formal syntax

- **Concrete syntax**
  - Shape, colour, position, size, …
  - I.e. the actual diagram



Is a drawing of

- **Abstract syntax**
  - Occurrences of labels and their relevant topological relationships
  - An abstract codechart can be **drawn** any number of ways
  - Distinctly different codecharts may have the same abstract syntax
  - The abstract syntax gives a structure to which we can define a formal semantics

- $\mathcal{R} = \{(\texttt{Media}, 1)\}$
- $\mathcal{OR} = \{(\{\texttt{Music}, \texttt{Film}\}, 1)\}$
- $\mathcal{E} = \{(\texttt{getTitle}, 1), (\texttt{getTitle}, 2)\},$
- $\mathcal{T}_v = \{(\mathit{Abstract}, 1)\},$
- $\mathcal{A}_r = \{((\mathit{Inherit}, (\{\texttt{Music}, \texttt{Film}\}, 1), (\texttt{Media}, 1)), 1)\}$
- $\omega((\texttt{getTitle}, 1)) = (\texttt{Media}, 1),$
  $\omega((\texttt{getTitle}, 2)) = (\{\texttt{Music}, \texttt{Film}\}, 1),$
  $\omega((\mathit{Abstract}, 1)) = (\texttt{Media}, 1).$

- J. Howse, F. Molina, J. Taylor, and S.-J. Shin. Type-syntax and token syntax in diagrammatic systems. In Proceedings of the international conference on Formal Ontology in Information Systems, FOIS '01, pages 174–185, NY, USA, 2001. ACM.
- J. Nicholson, A. Delaney, and G. Stapleton. Formalizing the syntax of codecharts. In Proceedings of the 2012 International Workshop on Visual Languages and Computing (VLC 2012), Miami, USA, August 2012.

Just for Phil…

Definition 1. *A* **concrete codechart** *is a tuple*

$$(R, T_{\triangle}, E, OR, OT_{\triangle}, OE, T_{\triangledown}, A_{\blacktriangleright}, A_{\blacktriangleright\blacktriangleright}, \lambda)$$

where:

1) R *is a finite set of rectangles*
2) $T_{\triangle}$ *is a finite set of triangles*
3) E *is a finite set of ellipses*
4) OR *is a finite set of offset rectangles*
5) $OT_{\triangle}$ *is a finite set of offset triangles*
6) OE *is a finite set of offset ellipses*
7) $T_{\triangledown}$ *is a finite set of inverted triangles*
8) $A_{\blacktriangleright}$ *is a finite set of single headed arrows*
9) $A_{\blacktriangleright\blacktriangleright}$ *is a finite set of double headed arrows*
10) λ *is a labelling function that ensures*
    a) *for each* $x \in R$, $\lambda(x) \in \mathcal{C}$
    b) *for each* $x \in T_{\triangle}$, $\lambda(x) \in \mathcal{CH}$
    c) *for each* $x \in E$, $\lambda(x) \in \mathcal{S}$
    d) *for each* $x \in OR$, $\lambda(x) \in \mathbb{P}(\mathcal{C})$
    e) *for each* $x \in OT_{\triangle}$, $\lambda(x) \in \mathbb{P}(\mathcal{CH})$
    f) *for each* $x \in OE$, $\lambda(x) \in \mathbb{P}(\mathcal{S})$
    g) *for each* $x \in T_{\triangledown}$, $\lambda(x) \in \mathcal{UR}$
    h) *for each* $x \in A_{\blacktriangleright} \cup A_{\blacktriangleright\blacktriangleright}$, $\lambda(x) \in \mathcal{BR}$

such that the following constraints hold:

1) *each* $x \in E$ *overlaps a unique* $s \in PG$, $int(x) \cap int(s) \neq \varnothing$
2) *each* $x \in OE$ *overlaps a unique* $s \in PG$, $int(x) \cap int(s) \neq \varnothing$
3) *each* $x \in T_{\triangledown}$ *overlaps a unique* $s \in PGC$, $int(x) \cap int(s) \neq \varnothing$
4) *no two distinct* $a, b \in PG$ *overlap*, $int(a) \cap int(b) = \varnothing$
5) *each* $a \in A_{\blacktriangleright} \cup A_{\blacktriangleright\blacktriangleright}$ *is sourced on an element of* PGC, *written* $s(a) \in PGC$, *and targets an element in the same set, written* $t(a) \in PGC$

Definition 2. *An* **abstract codechart** *is a tuple:*

$$(\mathcal{R}, \mathcal{T}_{\triangle}, \mathcal{E}, \mathcal{OR}, \mathcal{OT}_{\triangle}, \mathcal{OE}, \mathcal{T}_{\triangledown}, \mathcal{A}_{\blacktriangleright}, \mathcal{A}_{\blacktriangleright\blacktriangleright}, \omega)$$

where all bags are finite and:

1) $\mathcal{R}$ *is a finite bag whose elements are chosen from* $\mathcal{C}$
2) $\mathcal{T}_{\triangle}$ *is a finite bag whose elements are chosen from* $\mathcal{CH}$
3) $\mathcal{E}$ *is a finite bag whose elements are chosen from* $\mathcal{S}$
4) $\mathcal{OR}$ *is a finite bag whose elements are chosen from* $\mathbb{P}(\mathcal{C})$
5) $\mathcal{OT}_{\triangle}$ *is a finite bag whose elements are chosen from* $\mathbb{P}(\mathcal{CH})$
6) $\mathcal{OE}$ *is a finite bag whose elements are chosen from* $\mathbb{P}(\mathcal{S})$
7) $\mathcal{T}_{\triangledown}$ *is a finite bag whose elements are chosen from* $\mathcal{UR}$
8) $\mathcal{A}_{\blacktriangleright}$ *and* $\mathcal{A}_{\blacktriangleright\blacktriangleright}$ *are finite bags whose elements are chosen from*

$$\{(l, s, t) : l \in \mathcal{BR} \wedge s, t \in \mathcal{PGC}\}$$

9) $\omega : \mathcal{E} \cup \mathcal{OE} \cup \mathcal{T}_{\triangledown} \rightarrow \mathcal{PGC}$ *specifies overlaps such that:*
    a) $\omega|_{\mathcal{E}}$ *has codomain* $\mathcal{PG}$
    b) $\omega|_{\mathcal{OE}}$ *has codomain* $\mathcal{PG}$

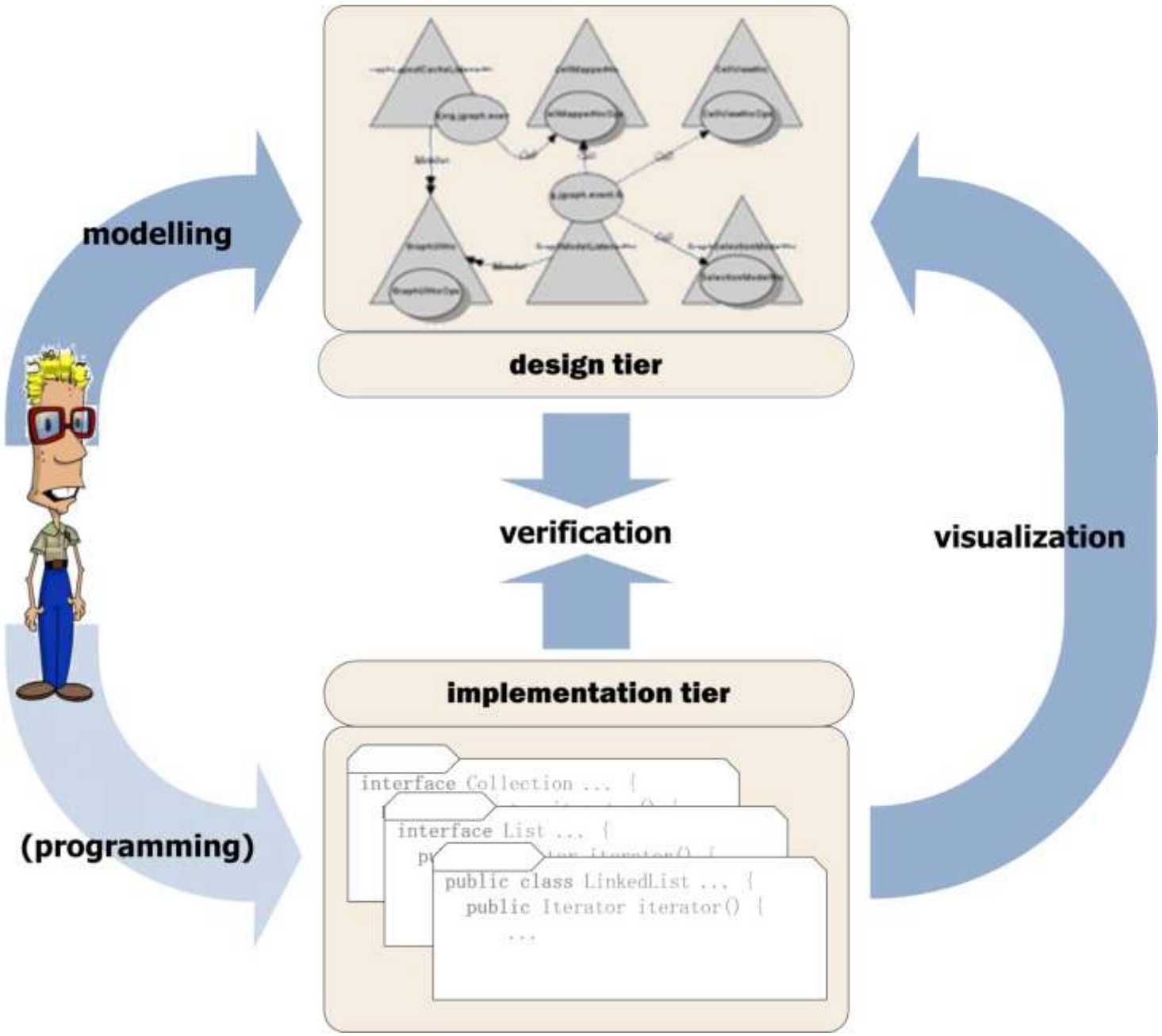
Definition 3. *Let* c *be a concrete codechart and let* C *be an abstract codechart. We say that* c *is a* **drawing** *of* C *provided there exists a function* $f : PGC \cup T_{\triangledown} \rightarrow \mathcal{PGC} \cup \mathcal{T}_{\triangledown}$ *such that the following hold:*

1) *for each* i *where* $1 \leq i \leq 7$, $f|_{\pi(c,i)}$ *is bijective with codomain* $\pi(C, i)$ *such that for each* $x \in \pi(c, i)$ *and some* j: $f(x) = (\lambda(x), j)$
2) *for each* i *where* $8 \leq i \leq 9$, $f|_{\pi(c,i)}$ *is bijective with codomain* $\pi(C, i)$ *such that for each* $x \in \pi(c, i)$ *and some* j: $f(x) = ((\lambda(x), f(s(x)), f(t(x))), j)$
3) *for all* $x \in E$, x *overlaps* $s \in PG$ *iff* $\omega(f(x)) = f(s)$
4) *for all* $x \in OE$, x *overlaps* $s \in PG$ *iff* $\omega(f(x)) = f(s)$
5) *for all* $x \in T_{\triangledown}$, x *overlaps* $s \in PGC$ *iff* $\omega(f(x)) = f(s)$

If c *is a drawing of* C *then* C *is an* **abstraction** *of* c.

What am I doing now?

- Investigating the semantics of codecharts
  - Required for any further work
  - Blindly reusing the existing semantics will not fly
  - There are issues that need to be resolved
  - Every change has consequences
  - Proving to be harder than we thought…
- Thinking about the next step, e.g.
  - User studies
    - E.g. Are codecharts a "good" visual language?
  - Program metrics
    - Formally capturing and representing useful program metrics
    - Investigate how this helps users in reverse engineering tasks
  - Software evolution (e.g. CVS commits)
  - Gestural interface to reverse engineering


modelling
design tier
verification
visualization
implementation tier
(programming)
interface Collection ... {
interface List ... {
public class LinkedList ... {
public Iterator iterator() {
...