Стили программирования как общий подход к системе понятий информатики *

Н.Н. Непейвода (nnn@uni.udm.ru)
Удмуртский государственный университет

Аннотация. Рассматриваются стили программирования как концептуальный базис системы понятий информатики.

Keywords: Теория программирования, преподавание программирования, информационные технологии, индустриальное программирование

1. Почему появилось понятие стилей?

Информатика накопила громадный багаж экспериментальных и теоретических данных, которые нуждаются в серьезном осмыслении. В информатике, поскольку она занимается сложнейшими умственными конструкциями и является одним из немногих мест в практике, где используются действительно искусственные объекты, многие проблемы представлены в наиболее ярком и поэтому благодарном для анализа виде. Развитие информатики невозможно без соединения высокой теории и высокой практики, что также делает ее одним из серьезнейших и важнейших объектов анализа. Накопленный багаж исключительно сильных и тонких практических и теоретических методов (достаточно вспомнить функциональное программирование и топологическую теорию областей, абстрактные типы данных и алгебраическую теорию представлений) показывает, что наука и практика подошли к уровню зрелости. Более того, информатика достигла такой точки, когда просто не может развиваться дальше без системы знаний и концепций[1].

Основаниями для произведенного анализа является основные выводы логической теории и методологии XX века

Первым основанием служит принцип:

> *Ни одна формализация не является универсальной. Каждая формализация ограничена, и, более того, при достаточно серьезном анализе сама подсказывает собственные альтернативы.*

* Работа частично поддержана РФФИ, грант 99-01-00116 и РГНФ, грант 02-03-18307а

[1] Некоторые примеры, когда отсутствие системы знаний уже повредило развитию информатики, приведены ниже.

Он является следствием теорем Геделя о неполноте и теоремы Тарского о невыразимости истины.

Таким образом, необходимо прежде всего четко знать границы применимости каждого метода и отказаться от постановки вопроса о каком-то единственно правильном методе, которым можно решить все задачи[2].

Уже конструктивные логики для различных классов задач противоречат друг другу в самих своих основаниях. Так, в линейной и интуиционистской логиках, описывающих построение программ на базе локальных действий и условий, принимается формула

$$A \Rightarrow (B \Rightarrow A) \tag{1}$$

В нильпотентной логике, описывающей построение программ на базе глобальных необратимых действий и локальных условий, принимается правило исключенного застоя:

$$\frac{A \Rightarrow A}{\neg A} \tag{2}$$

В реверсивной логике, описывающей построение программ на базе глобальных обратимых действий и глобальных условий (например, программ для квантового или сверхпроводящего компьютера), выполнена формула

$$(A \Rightarrow B) \Leftrightarrow (B \Rightarrow A) \tag{3}$$

Все указанные формулы опровергаются в двух других классах логик.

Даже если две формализации сложного естественного понятия формально не противоречат друг другу, они начнут противоречить и мешать при попытках более глубокого анализа либо развития системы (т. н. *концептуальные противоречия*). Даже явные противоречия упорно игнорируются и теорией, и практикой, зацикленной на позитивном мышлении, а концептуальные вообще не принимаются во внимание.

Поэтому соединять все мыслимые средства в одном и том же месте — верный путь к саморазваливающимся системам. Заметим,

[2] На первый взгляд кажется, что в алгоритмике эффекты неполноты не имеют места, поскольку имеется теорема об универсальном алгоритме. Но эта теорема перестает действовать, как только мы ограничиваем вычислительную сложность программ. Далее, нам важно не только, чтобы программа считала, но чтобы она получала нужные ответы для корректных входов и чтобы она была модифицируема. А корректность программы и ее перестраиваемость прежде всего приносятся в жертву при использовании неподходящих средств.

что такое соединение является общим местом в современной информатике, поскольку принято критиковать системы за то, чего в них нет. Возникающие при добавлении новых возможностей концептуальные противоречия никого не волнуют, пока не становится слишком поздно.

Вторым основанием служит гильбертовская концепция идеальных и реальных объектов и парадокс изобретателя. Большинство объектов и понятий в областях, пользующихся развитым математическим аппаратом, не имеют прямых интерпретаций[3] в реальности. Но их устранение, даже если оно в некоторых случаях теоретически возможно, приводит к неприемлемому (как минимум, башня экспонент) разрастанию длины доказательств. Таким образом, *окольные пути являются наиболее короткими и эффективными.*

Третьим основанием является система оценки сложности математических понятий по их *типам.* Первый тип — объекты, второй тип — их множества либо функции над объектами, третий тип — функции над объектами второго типа и т. д.

Система знаний тем более необходима потому, что информатика неразрывно связана с современной экономикой, в которой индустрия обмана и самообмана (реклама) занимает решающее место, и то, что влияние этого же духа проникло в современную науку в виде системы грантов[4], которая требует безудержной саморекламы и мешает действительно серьезному теоретическому и методологическому осмыслению проблем. Это лишь частный случай вредного влияния на прогресс квазирелигии прогресса[5].

2. Стили программирования

В книге [1] нами введено понятие стиля программирования.

Под *стилем программирования* понимается общая и внутренне согласованная совокупность базовых конструкций программ и способов их композиции, обладающая общими фундаментальными особенностями, как логическими, так и алгоритмическими. Стиль включает также совокупность базовых концепций, связанных с этими программами.

[3] Порою не имеют даже косвенных.

[4] В первые годы своего появления система грантов имела многие положительные черты, но, как и любая формализация неформализуемого, она через некоторое время изживает сама себя и начинает заводить в тупик, который грозит стать глобальным, если она вдобавок глобализована.

[5] Любое абсолютизированное человеческое понятие превращается в идола и вредит прежде всего здоровью и развитию именно того, для чего оно было первоначально полезно.

Стиль программирования реализуется через *методологии программирования*. В большинстве случаев методология заключается в совокупности соглашений о том, какие базовые концепции языков программирования и какие их сочетания считаются приемлемыми или неприемлемыми для данного стиля. Методология включает в себя, в частности, модель вычислителя для данного стиля.

Методология реализуется через *методики*, которые состоят из следующих компонент:

1. Поощрение (или прямое предписание) использования некоторых базовых концепций программирования.

2. Запрещение (или ограничение) применения некоторых других базовых концепций. Иногда запрещение либо ограничение может быть неявным, через исключение нежелательных концепций из предписываемого языка или его диалекта.

3. Требования и рекомендации по оформлению и документированию программ.

4. Совокупность инструментальных и организационных средств, поддерживающих все вышеперечисленные требования и рекомендации.

Заметим, что в программировании происходит систематическая 'возгонка' понятий. То, что в других местах называется орудием, здесь называется методом; то, что называется методом или методикой, называется методологией; то, что называется методологией, возводится в ранг парадигмы, и т. д. Здесь мы придерживаемся выражений, более адекватно передающих смысл используемых понятий.

Стили программирования естественно классифицируются по трем координатам: первоуровневые и высокоуровневые, глобальность либо локальность действий и условий.

Высокоуровневые стили отличаются тем, что функции и преобразования могут служить полноправными значениями.

В соответствии с приведенными характеристиками, естественно выделить четыре первоуровневых стиля.

1. Структурное программирование (действия и условия локальны);

2. Программирование от состояний (действия глобальны, условия локальны);

3. Программирование от событий (действия локальны, условия глобальны);

4. Сентенциальное программирование (действия и условия глобальны);

С программистской точки зрения эти стили различаются по следующим характеристикам.

В структурном программировании, которому сейчас учат как монопольному первоуровневому стилю, управляющие структуры образуют иерархию, а потоки передачи данных в принципе должны согласовываться с данной иерархией. Математической моделью программ являются здесь вычислимые функции. Логической моделью программирования является линейная конструктивная логика. Структуры современных традиционных языков программирования (Pascal, C, Ada и др.) поддерживают прежде всего данный стиль.

В программировании от состояний процесс представляется как смена состояний системы. Новое состояние возникает в результате действия, изменяющего старое состояние, а выбор этого действия зависит от проверки условий. Математической моделью программы служит конечный автомат. Логической моделью служит нильпотентная конструктивная логика. Естественным способом программирования такой задачи на современных языках программирования является использование операторов **goto** либо объектов, обменивающихся информацией через общее поле памяти.

В сентенциальном стиле (Рефал, Пролог) каждый шаг программы проверяет все поле зрения на соответствие образцу, находит тем самым применимое правило преобразования, и, согласно найденному правилу, преобразует все поле памяти.

В программировании от событий условие состоит в том, что в системе произошло некоторое событие, лучшим обработчиком которого оказалось данное действие. Математические и логические модели такого подхода еще недостаточно развиты[6]. В качестве математических моделей можно упомянуть сети Петри. Кандидатом на роль логических моделей являются релевантные логики.

Высокоуровневое программирование *в принципе* должно иметь набор стилей, соответствующий тому же морфологическому ящику. Но пока в наличии из четырех имеются $1\frac{1}{4}$ стиля.

Это прежде всего *функциональный стиль*, наиболее ярким и показательным примером которого является LISP. Здесь реализованы вычисления над функциями и структуры данных полностью соответствуют структуре функций. Математическим описанием данного стиля является λ-исчисление. Логической моделью является интуиционистская логика.

[6] Впрочем, и программистская поддержка его, несмотря на его важность, наименее развита.

"Объектно-ориентированный"[7] стиль является четверть-шагом к высокоуровневому стилю, соответствующему инварианту: "Действия глобальны, условия локальны". При действиях целиком меняются объекты, представляющие собой модели, состоящие из данных и методов их обработки. К сожалению, нет теории функционалов высших типов, описывающих действия, изменяющие мир, а нильпотентная конструктивная логика также не разработана для формул высших порядков.

Современная информатика и на уровне теории (теория областей данных, выросшая из задачи формализации функционального программирования и далеко перешагнувшая ее пределы), и на уровне практики подошло к моменту, когда возможно создание сентенциального высокоуровневого стиля, стиля вычислений над вычислениями.

Высокоуровневый событийный стиль пока что не подготовлен, но есть некоторые косвенные наметки того, что удачная формализация программирования от событий одновременно даст идеи и для конструкций высших уровней.

Ни один стиль программирования не описывается чисто логически. Это связано прежде всего с наличием в языках программирования фиксированных типов данных, например, чисел.

Соединение хотя бы одного фиксированного типа данных с нетривиальными способами композиции вызывает к жизни все затруднения теоремы Геделя о неполноте и парадокса изобретателя. Итак, первый нетривиальный вывод:

> *Нужно четко разделить композицию понятий и их конструирование. Работу с исходными типами данных нужно помещать в среду суровых ограничений на правила композиции, а там, где требуется создавать сложные структуры понятий либо действий, конкретные данные нужно полностью отбросить, имея лишь косвенные ссылки на них через используемые модули низшего уровня. Итак, вычисления и рассуждения концептуально противоречат друг другу.*

Из этого следует и педагогический вывод: математика, преподаваемая на базе монополии анализа, годится для описания физических, но не годится для описания информационных процессов. Для информатиков намного важнее логика, алгебра, топология, лингвистика и философия. Для них важнее преобразовывать понятия, чем формулы.

[7] Имя взято в кавычки, поскольку оно идет от конкретной реализации данного стиля, и совершенно не отражает его сути.

3. Первоуровневые стили

Проведем анализ развития и состояния существующих стилей.

3.1. Структурное программирование

Этот стиль лучше всего обоснован теоретически (теорема Бема-Джакопини о структурировании схем программ), использует теорию корректно, но даже на нем видно, насколько плохо "практики" воспринимают даже недвусмысленные и легко формулируемые теоретические выводы. Восприняв что-то одно, они уже не могут воспринять другое, связанное с ним, а бросаются напропалую реализовывать первое.

Во-первых, уже Бем и Джакопини показали, что для структурирования произвольных схем часто нужно добавлять дополнительные переменные, по сути дела запоминающие признак того, что поддействие закончилось, а затем многократно проверять их. Д. Кнут показал, что для преодоления этого недостатка достаточно ввести завершители для блоков и иметь право во внутреннем блоке завершить любой объемлющий (т. н. *структурные переходы*). Но нигде (кроме забытого русского языка ЯРМО и автокода Эльбрус) структурные переходы в корректном виде не реализованы, и в результате приходится вводить операторы goto и долго и бессистемно объяснять. когда же их прилично использовать. Уж Java-то могла бы вместо бессистемных синтаксических ограничений на переходы просто воспользоваться великолепно разработанной и совершенно ясной концепцией завершителей! За 30 лет можно было бы чему-то и научиться!

Во-вторых, Дейкстра и Грис ясно показали, что делать программу последовательной с самого начала значит искажать ее логику. Никаких дополнительных усилий не потребовало бы воспринять хотя бы концепцию совместно исполняемых операторов Алгола 68, когда детерминируется лишь то, что необходимо детерминировать! Но в первом приближении удобно, чтобы программа всегда исполнялась однозначно, это и хакерским трюкам способствует! Эта также длящаяся уже более тридцати лет и упорно игнорируемая недоработка приводит к постановке неестественной задачи распараллеливания программ и мешает эффективному использованию параллельных компьютеров, поскольку программы написаны

людьми, привыкшими к последовательному мышлению[8], а средства параллелизма даются как неестественные примочки.

И, наконец, никто не догадался, несмотря на многократные наблюдения (подтверждаемые результатами теории сложности), что циклы и рекурсии плохо смешивать вместе. На самом деле есть два извода структурного программирования. Когда у нас данные также структурированы иерархически, то целесообразно пользоваться циклами, а когда в типах данных есть ссылки на самих себя, целесообразна рекурсия.

Есть еще одна, менее тривиальная, недоработка. Это практическое игнорирование того, что программу не опишешь понятиями, которые заданы внутри ее самой[9]. Давно уже известно, что для полного описания свойств программы необходимы призраки — понятия, в программу не входящие, зачастую просто вредные для вычислений и даже не представимые в машине (например, ординалы), но необходимые для корректного описания системы. Типичным примером призрака является число шагов цикла общего вида. Масса примеров возникновения и использования призраков приведена в [1].

3.2. Программирование от состояний

Структурное программирование практически задавило совершенно другой, применимый в других условиях, стиль: программирование от состояний. **Goto** было безоговорочно объявлено вредным оператором. А оно просто плохо совместимо с оператором присваивания. В последние годы программирование от состояний было возрождено и стало секретным оружием команды петербургских программистов, дважды ставших чемпионами мира (см. [3]).

Программирование от состояний естественно тогда, когда задача описывается как совокупность состояний мира и переходов между ними. Средства для его поддержки уцелели во всех традиционных языках, и оно великолепно реализуется как последовательность проверок условий, вызовов функций, соответствующих состояниям автомата, и переходов в очередное состояние. Функции, реализую-

[8] Интересно, что первая “программа”, вдохновившая Беббиджа на создание первой вычислительной машины, была параллельной. Во время Великой французской революции французские математики организовали работу по пересчету артиллерийских, навигационных, астрономических и геодезических таблиц, запрограммировав с помощью конечных разностей их приближенные вычисления таким образом, что их могла параллельно выполнять рота солдат, каждый из которых делал сложение либо вычитание и передавал результат следующим.

[9] На самом деле это — общая беда всех стилей и методологий программирования.

щие состояния автомата, чаще всего естественно программируются в структурном стиле (и здесь мы видим первый пример того, что *стили не должны ходить в одиночку*).

3.3. Сентенциальное программирование

Два альтернативных (и даже теоретически несовместимых) извода сентенциального программирования — программирование, основанное на возвратах и унификации (Prolog) и программирование, основанное на конкретизации (Рефал). Оба они живы. Первый вариант лучше подходит для моделирования логики решений, когда эффективность не критична и явное выписывание условий утомительно (задачи поиска). Второй вариант лучше подходит для синтаксических преобразований иерархических структур.

Исключительно интересен пример сентенциального программирования в связи с глобализацией. Глобализация усиливает монополизацию, в том числе и в интеллектуальной схеме. Этот стиль возник независимо в СССР и на Западе, и созданные его конкретизации оказались ортогональными. При множестве общих черт Рефал и Prolog несовместимы по механизму сопоставления поля зрения с образцом и механизму управления. Если Рефал великолепно приспособлен для &-параллелизма, когда параллельно запускается несколько процессов и результат получается обобщением всех полученных частных результатов, то Prolog столь же великолепно приспособлен к $\vee$-параллелизму, когда несколько процессов соревнуются в том, кто быстрее достигнет цели. Ввиду "железного занавеса" обе концепции выжили, встали на ноги, актуальны до сих пор.

Программирование на Prolog традиционно называют логическим. От логики здесь ничего не осталось, и термин вводит в заблуждение (и даже если бы осталось, логика здесь была лишь инструментом, и ставить конкретный инструмент впереди концепции то же самое, что телегу впереди лошади).

Теория была корректно применена и конкретизирована лишь в случае Рефала (но с точки зрения интерфейсов этот язык так и остался недоработанным, хотя с самого начала была ясна его специализированность и, соответственно, целесообразность использования в системах, где другие подзадачи решаются другими методами).

Prolog гораздо лучше сопрягается с другими языками, но безнадежно неэффективен и по ресурсам, и технологически для нетривиальных случаев. Добавленные примочки, которые должны были бы повышать эффективность, полностью развалили концепцию языка

(и здесь ненависть к знаниям выразилась в том, что возможности, которые лучше было бы переложить на модули, написанные на других языках и в других стилях, вставили в сам язык).

Prolog ярко демонстрирует, что происходит, когда теорию используют как заклинание, а не как обоснование. Хорновские дизъюнкты, давшие идею языку, совершенно неадекватны тому. что получилось, когда взятую из логики идею унификации соединили с другой, великолепно сочетающейся с унификацией, идеей: косвенного управления посредством возврата после неудач. В итоге сообщество Prolog упорно держится за жупел логики, что мешает ему осознать реальные достижения подхода.

3.4. Программирование от событий

Программирование от событий возникает тогда, когда действия активизируются в связи с тем. что возникло некоторое состояние системы и не нашлось лучшего кандидата для реакции на это действие. Таки образом, здесь обработчик условий централизован, он определяет, кто готов и кто должен обработать происшедшее событие, например, щелчок мыши (чтобы установить контекст щелчка, нужно порою проанализировать состояние интерфейса всех работающих процессов).

Программирование от событий имеет два извода:

— собственно событийное программирование, когда событие (например, щелчок мыши) снабжает процесс-обработчик важной информацией (например, о месте щелчка);

— программирование от приоритетов: событие состоит в том, что все более приоритетные процессы ничего не могут сделать, и никакой позитивной информации активизируемому процессу не дает.

Это программирование в самостоятельном виде реализовано в некоторых языках скриптов для специализированных процессоров. Но на самом деле это — равноправный стиль, реализуемый в традиционных языках хакерскими приемами.

4. Высшие уровни

4.1. Функциональный стиль

Функциональный стиль программирования, когда программа представляет собой функционал высокого уровня, преобразующий функции, представлен языками LISP ML, Huskell и другими. Если функции типизированы, то этот подход, сохраняя возможности понятий высших уровней исключительно компактно выразить сложную структуру, вдобавок эффективен по использованию ресурсов. Но нынешние реализации функционального программирования не могут удержаться от использования рекурсий и нетипизируемых конструкций типа оператора вычисления произвольного выражения или оператора неподвижной точки, смешивая тем самым один из изводов структурного программирования с функциональным. Смелости избавиться от нижнего уровня и от лишних возможностей никак не хватает. Это создало функциональному программированию репутацию крайне неэффективного стиля, подходящего лишь для прототипов программ.

Более того, в самом LISP четко выделено и приличными программистами используется концептуально целостное ядро, а последующие языки явились украшенным вырождением идеи, поскольку вставляли в нее модные новые возможности без их критического анализа. Дошло до того, что в функциональное программирование для его 'усиления' вставляется концептуально неизмеримо более низкие и излишние в развитом мире высших сущностей элементы объектно-ориентированного программирования.

4.2. Действия

Объектно-ориентированное программирование является четвертьшагом к высокоуровневому программированию, соответствующему функциональному для действий и событий. Объекты показали даже достаточно необразованным людям, что для сложных систем работа с действиями эффективнее работы с данными.

Декларировано, что объекты возникли на базе теории абстрактных типов данных (АТД), но здесь ситуация еще тяжелее, чем в Prolog. *Теория просто не имеет никакого отношения к практике.* В тех случаях, когда употребляются теоретические термины, они употребляются ошибочно. Например, обогащение алгебраической системы называется ее конкретизацией.

С точки зрения практики, это несколько наметок разных стилей. Один из них берет начало скорее в психологии и искусственном интеллекте, и успешно применяется для массового производства программ, где внешняя упаковка важнее сути. Здесь объекты в

разных контекстах могут вести себя совершенно по-разному, так что они соответствуют ролям людей.

В данном случае четко видно, как ссылка на неадекватную сути подхода теорию абстрактных типов данных заставляет выпячивать слабейшие стороны подхода и мешает развитию действительно сильнейших. Программистам бы здесь к психологам обратиться, а не к математикам!

Другой рудиментарный стиль здесь — стиль работы с событиями и сообщениями. Третий — просто стиль программирования от состояний, где состояния изображаются различными классами, выводимыми из исходного. Это дало возможность формально избежать презренных `goto`, но программы от этого более понятными не стали.

5. Общие выводы

Рассмотрение стилей приводит к выводу о губительном влиянии на современную высокоуровневую практику следующих факторов.

Во-первых, это иллюзия универсальности. Пора рассматривать слова 'универсальная система' как нечто подобное философскому камню. Иллюзия универсальности заставляет вводить в систему возможности, губительным образом расширяющие ее.

Переход к следующему уровню понятий невозможен без запрещения многих методов, действовавших на предыдущем уровне. Но в программировании этому препятствует иллюзия универсальности, конкретизирующаяся в предрассудок совместимости, который заставляет при развитии системы тянуть за собой шлейф концептуально несовместимых устаревших понятий. Случаи, когда от этого предрассудка отказались, считаны.

Во-вторых, это игнорирование отрицательных результатов, или, в более общем виде, позитивное мышление и квазирелигия прогресса. Закрывание глаз на теоретические предупреждения никогда ни к чему хорошему не приводит, а их систематическое изгнание из учебных курсов в угоду душевному комфорту и 'позитивности' вредно сказывается на способностях к творческому мышлению, один из приемов которого: превратить минус в плюс[10].

[10] Впрочем, что Вы хотите, если в США принята социологическая программа по мягкому лишению вундеркиндов их способностей. Конечно же, при этом говорится масса хороших слов о том, что стремятся к их счастью, но одна фрейдистская оговорка подчеркивает ее суть: гении подрывают ценности потребительского общества, поскольку получают от творчества гораздо большее наслаждение, чем обычный человек от секса. Более того, это наслаждение недоступно большинству, и тем самым его не купишь за деньги!

В-третьих, это дурно понимаемый прагматизм. Из теории выхватывается что-либо одно, и как можно быстрее оформляется в виде практической системы.

Наблюдение за состоянием дел в современном "искусственном интеллекте" и информатике приводит к следующему закону экологии искусственных систем:

> *Первая система, декларировавшая успех*[11] *в данной экологической нише, захватывает ее и изгаживает таким образом, чтобы никакая другая в ней не могла выжить.*

Еще одной стороной того же прагматизма является ранняя стандартизация. Случайные особенности недозрелой системы фиксируются в стандарте, и, поскольку лучшие специалисты в данной области слишком часто имеют хакерские наклонности и привыкли использовать недоделки в качестве якобы виртуозных приемов, эти недоделки возводятся в ранг священной коровы. Например, такой статус приобрела ошибка в реализации алгоритма унификации в раннем Prolog и конкретный способ реализации возвратов, который был там применен. Здесь нужно было бы в позитивном смысле воспринять Оруэлла. у которого словарь Новояза вплоть до 11 издания был предварительным, поскольку продолжались усилия обеспечить концептуальное единство языка и убрать лишние возможности.

И, наконец, программирование больше похоже на стихи, чем на прозу. Без самоограничения не достигнешь совершенства, и во всех случаях. когда пытались обеспечить программистам «свободу выражения», это приводило лишь к концептуальным противоречиям и открывало настежь двери для хакерства. Так что нужно нацеливать мысль, а не освобождать ее. Свобода для творца не является ценностью, если продуктами его творения будут вынуждены пользоваться другие.

Список литературы

1. *Непейвода Н.Н., Скопин И. Н.* Основания программирования.— Ижевск–Москва: РХД, 2003.
2. *Непейвода Н.Н.* Квазиискусственные объекты. // Современная логика-2002. — СПб.: Наука, 2002.— С. 28–30.
3. *Шалыто А.А.* SWITCH-технология. СПб.: Наука, 1998.

[11] Насколько этот успех реален, обычно второй вопрос, которым не задаются (по причине позитивного мышления), пока не станет слишком поздно.

Programming styles as foundation of a notion system for informatics

Nikolai N. Nepejvoda
(Izhevsk)

Abstract. There are no really universal systems. Freedom of expression for programmers is the worst enemy of users. Extra features added to good system usually make it worse.

Conceptual unity is the best property of a program. Thus it is necessary to classify most general classes of programming problems and connect them with good tools. There is a rough classification of programming styles.

Actions	Conditions	
Local	Local	Structured Programming
Global	Local	Automata programming
Local	Global	Events Programming
Global	Global	Sentential Programming

The unique mature style of high-level programming (functional programming, LISP and ancestors) corresponds to structured programming. Three other styles are not developed yet. OOP is the first step to Automata high-level programming.

Keywords: Theory of Programming, Education in Informatics, Information Technologies, Industrial Programming.